The Amulet Programming Language

CoderPuppy, demhydraz, SquidDev

October 14, 2016

Abstract

Amulet is an experimental, purely-functional programming language borrowing features from OCaml, Haskell, PureScript and Idris.

Contents

1. Semantics

1.1 Summary of Amulet

- A purely functional language
- Uses the Extensible Effects monad[1] to represent side effects
- Complex type system supporting type classes and row types
- Strict by default, though laziness is encompassed in the type system

1.2 A basic calculus

Amulet is composed of multiple layers of lambda calculus, each layer describing the layer below it. There are two primary levels of calculus: expressions (e) and types (τ).

Every object in both the expression and type calculus has a type. In the case of expressions it is of type τ. Types also have the type τ, though lifted one level higher: τ_n is of type τ_{n+1}. This is known as a "cumulative hierarchy". Basic types can be considered to be on τ_0, types describing types (on τ_1) are known as "kinds".

These layers are best demonstrated by an example.

- The root layer consists of expressions and values. The calculus here describes the operation of your program. We will use the basic expression `show`.
- Types help ensure that the program is well formed. This prevents you from adding a number to a string. The type of this expression is $\forall a.$`Show` $a \Rightarrow a \to$ `String`. This type is composed of several key components:
  - $\forall a.\tau$ creates an environment for the type τ containing the type variable a.
  - `Show` $a \Rightarrow \tau$ states that the type constraint `Show` a must exist for the type τ to exist.
  - $a \to$ `String` is a function that maps the type variable a to the type `String`.
- Most types are of type `Type` (often abbreviated to $*$), for example `String` or $a \to$ `String` . However `Show` is of kind $* \to$ `Constraint`. This indicates a kind which maps a type to a `Constraint`.

[1] Kiselyov, Sabry, and Swords 2013.

1.2.1 A calculus

The calculus is defined in terms of expressions, types and kinds.

$v =$	variable
$\mid x : \tau$	`term variable`
$\mid \alpha : \tau$	`type variable`
$\mid \Gamma : \tau \sim \tau$	`coercion variable`
$e =$	expressions
$\mid v$	`variable`
$\mid \texttt{let } v = e_1 \texttt{ in } e_2$	`let binding`
$\mid \texttt{letrec } (v_n = e_n) + \texttt{ in } e_{n+1}$	`letrec binding`
$\mid \lambda x.e$	`term lambda`
$\mid \Lambda \alpha.e$	`type lambda`
$\mid \Lambda(\Gamma : \tau_1 \sim \tau_2).e$	`coercion lambda`
$\mid e_1\ e_2$	`term application`
$\mid e\ \tau$	`type application`
$\mid e\ \Gamma$	`coercion application`
$\mid \texttt{match } e_1 \texttt{ with } \mid p \to e_2$	`pattern matching`
$\mid e \rhd \Gamma$	`type-safe cast`
$\tau =$	types
$\mid \alpha$	`type variable`
$\mid \tau_1\ \tau_2$	`type application`
$\mid \forall \alpha.\tau$	`for all`
$\mid \tau_1 \Rightarrow \tau_2$	`constraint`
$\mid \tau_1 \to \tau_2$	`arrow`

1.2.2 Kinds

As discussed above, all types are described by a type.

- `Type`: A fully constructed type. This is the base case for all types: its type is $\texttt{Type}_{n+1}$.
- `Constraint` : $*$: A constraint over a type.
- `Coercion` : $*$: A coercion between two types.
- `Row` (#): map of string keys to kinds (like a disjoint union but with strings). As this is polymorphic over kinds it must have a kind applied to it first ($\square \to \square$).
- `Union` (%): a unique, unordered set of a given kind. These are also polymorphic and so must be given a kind.

1.3 Types

Types in Amulet are composed of functions, tuples and primitives. You can store multiple types together in a tagged union (or sum type). A tagged union with one entry is a useful way of boxing a type to ensure that it cannot be accidentally used as an object of the same type but different semantics.

Types can also be generic over one or more parameters. When inferring the type for an expression, the most generic type will always be used. Constraints can be put on generic parameters by specifying type class implementations that must exist for this set of parameters. There can be multiple constraints, each applying to multiple parameters.

There is no subtyping in Amulet, *however* type constraints can be loosened through the upcast operator: this converts an existing value to a more general type: for instance `[a]` can be upcast to `Show b`.

1.3.1 Types and kinds

> A kind is the type of a type constructor or, less commonly, the type of a higher-order type operator. A kind system is essentially a simply typed lambda calculus "one level up", endowed with a primitive type, denoted $*$ and called `Type`, which is the kind of any data type which does not need any type parameters.[2]

The base lambda calculus in Amulet is that which operates on values, namely the input program. Every value must have a corresponding type. 0 might have type `int` and the empty list type `[a]`. These have kinds $*$ and $* \to *$ respectively, where $*$ is a shorthand notation for `Type`.

However $*$ is not the only kind in Amulet. Constraints, such as `Show Int`, also have a kind called, imaginatively, `Constraint`. However type classes generally take one or more parameters and so have kind $* \to$ `Constraint`.

Unions and rows

In order to build more complex types Amulet adds row and union kinds. These are polymorphic over kinds and so have the signature $\square \to \square$. Here the $\square$ symbol represents all kinds such as types, constraints, rows and unions. These can then be used to create a compound type: `Row` $\to *$

Union kinds represent a unique, unordered set of a given kind. They are denoted by `Union` or $\%$. Unions are written using basic set notation: $\{$`Int`, `Bool`$\}$ is a set containing the `Int` and `Boolean` types.

Row kinds represent a map of string keys to kinds (like a disjoint union but with strings). This is denoted by `Row` or $\#$. Rows are written with the key value pairs within parenthesis: (`name` : `String`, `age` : `Int`).

Unions and rows share similar semantics as they allow being combined with another object of the same kind. For instance you can add `String` to the union `a` with $\{$`String`$|$`a`$\}$. This also highlights the fact that unions and rows can be polymorphic (this case is `Row` $\to$ `Row`). This simple fact allows emulating subtyping of these kinds. When creating a union of row objects where both sides have the same key the value must be unifiable.

It is also possible to take the union of two polymorphic kinds such as $\{a|b\}$ (however, type inference is not guaranteed).

1.3.2 Universal quantification

Universal quantification, or $\forall$ is a way of marking a type that can exists for all types matching its criteria. For instance the identity function would be universally quantified as any type can be applied to it.

Constraints

For universal quantification to be useful, constraints must be added to the type. Constraints come in two forms: a type class constraint, or a unification requirement.

The former requires that a particular type class instance exists: $\forall a.$`Num` $a \Rightarrow a \to a$ states that `Num` a must exist. It is worth noting that any number of variables (or constant types) can be give n to the type class:

[2] Wikipedia 2016.

$\forall a.\texttt{Num Int} \Rightarrow a \to a$ is equally valid, though meaningless as the only valid implementation would be the identity function.

A unification requirement states that two types must unify for this type to exist. An simple example might be $\forall a.a \sim \texttt{Int} \Rightarrow a$. This type is pointless as the only value of a which will satisfy this requirement is `Int`. However, this can be useful with more complex types such as rows or unions.

1.3.3 Record types

Record types express a way of handling key-value pairs where the keys are known at compile time. Records are implemented using rows, with a hidden polymorphic argument, allowing for converting records with many fields to records with less. Strictly records are defined as:

```
data Record p = forall a . Record ( a |  p )
```

1.3.4 Tagged unions

Tagged unions, also known as *sum-of-product types* or *Algebraic Data Types*, are data structures that can take on several variations. Only one of these variants may be in use at once, and a tag field explicitly indicates so.

Tagged unions are represented using a row, where each entry is the type of the corresponding entry. For example:

```
data Expr = Var Name
          | Abs Name Expr
          | App Expr Expr
```

could be represented as

```
type ExprTotal = (
    Var : Name -> Expr,
    Abs : Name -> Expr -> Expr,
    App : Expr -> Expr -> Expr,
)

data Expr = Expr (Union ExprTotal)
```

`Union` is an internal Amulet type which is understood by the compiler to form a union type.

Using the union kind It would be nice to replace this with a union as it doesn't *need* to have the types of the constructor. However this is hard as the union is over types rather than strings.

Narrowed functions

The main benefit of using rows to represent ADTs is that we can define functions which only take a subset of the values: for instance ‘head‘ could only accept non-nil lists.

For example, require a function which accepts `Abs` or `App` but not `Var`. Lets us consider the type `ExprPartial` p which represents a subset of `Expr`. We can define these requirements for this function:

You should be able to pass in anything of the form:

- `ExprPartial (App :  _, Abs :  _)`

- `ExprPartial (App : _)`
- `ExprPartial (Abs : _)`

But not of types:

- `ExprPartial ()`
- `ExprPartial (Var : _| p)`

Namely the arguments passed to this required function must be a subset of `(App : _, Abs : _)` and not an empty list. The first is trivial to solve: we can define a constraint `RowSubset` pa which requires that $a \subset p$.

```
type RowSubset p a = (a | p) ~ p
```

More difficult is defining a type which refuses the empty row. This could either be done with a non-empty-row constraint, or a more general "does not unify" requirement. For now we will consider the former and call it `NonEmpty` a.

We can then give the definition of `ExprPartial` p and of our function:

```
data ExprPartial a = ExprPartial (RowSubset ExprTotal a)

type Partial p = forall a . NonEmpty a => RowSubset p a => p
func : ExprPartial (Partial (App : _, Abs : _))
```

Of course `Expr` can now be generalized to:

```
type Expr = ExprPartial ExprTotal
```

Pattern matching

This form of narrowing can also be used in pattern matching. Each case can remove one of the possible values of ‘p‘ from ‘ExprPartial‘:

```
case (x : Expr) of
| Abs name _ = name
| y = ... -- y : ExprPartial (Var : _, App : _) as it cannot be Abs if you hit this point
```

GADTs

It is also possible to define GADTs in the same way, though obviously the resulting type is not just `Expr`. For instance, the GADT

```
data Expr a where
    I   : Int  -> Expr Int
    B   : Bool -> Expr Bool
    Add : Expr Int -> Expr Int -> Expr Int
```

would be represented:

```
type ExprTotal a = (
    I   : Int -> Expr Int
    B   : Bool -> Expr Bool
    Add : Expr Int -> Expr Int -> Expr Int
)
```

1.3.5 Type classes

Type classes are a way to ensure different types have a consistent interface, and to constrain polymorphic types on having an instance of a type class. A fully instantiated type class resolves to a type of `Constraint`.

A constraint section on a type is given as a fat-arrow behind the type. For example, the following snippet says that "for all types for which there is an instance of `a`, there is a function `show` transforming it into a `string`."

```
show : forall a. Show a => a -> String
```

The type-class mechanism poses a way to do name overloading based on types. In contrast to, say, Java interfaces, type classes allow the user to be parametric on the return type, not only the parameter types.

```
class Read a where
  read : String -> a
```

When resolving type classes the type checking algorithm will unify the current type across all instances. If more than one instance matches then an error will be created.

Type classes can also be named and a specific implementation can be used at compile time. Implementations are passed in to the type lambda. For instance using the above show definitions we get:

$$\texttt{show} = \Lambda(a : \texttt{Type})\ (x : \texttt{Num}\ a) \to \lambda(x : a)\ (y : a) \to function\ body$$

Type classes can take have multiple parameters. In the case where some parameters define others, functional dependencies[3] can be used to prevent conflicting definitions.

Some type classes will be auto-generated by the compiler (such as `Eq` or `Show`) if the criteria are matched (for instance all child types also implement `Show`). Auto-generated instances can safely be overridden or masked: there will not be a compile error if both a auto-generated and normal instance match.

Named implementations All type class instances require an explicit name. This allows us to explicitly specific implementations to functions.

Type families Type classes can also include type definitions[4], allowing for type aliases and data definitions to exist and be instantiated within a type class.

1.4 Modules

Modules allow splitting components into separate components. Unlike Haskell, exports are defined at the definition level. By default variables are exported, though you can also specify other levels:

- Public: export this variable, allowing access anywhere.
- Internal: export this variable, but only allowing access inside this compilation unit.
- Private: do not export this method

It is also possible to re-export entries from other modules.

If a variable's signature consumes an object which is has a more restrictive export level then an error would occur. For instance:

[3] Wiki 2016a.
[4] Wiki 2016b.

```
data private X = X Int

(* An error occurs as f is public but X is private and so isn't exported *)
let public f (X a) : Int = a
```

In the case of sum types, it is possible to export the type but not its constructors. This means it is trivial to provide a interface without exposing the implementation.

All definitions are accessible from any module when compilation occurs. This allows cross-module interaction and optimisation. However export checks occur after Amulet is parsed but before this exposing occurs so it is not possible to access inaccessible objects.

Implicit modules

Some constructs automatically create child modules which are imported into the primary module. These are generally used for types, and so you can index the type name to

Type classes All methods defined in a type class are automatically added to a submodule of the same name. This means you can use `Show.show` in addition to `Show`.

Tagged Unions All constructors and destructors defined in a tagged union are added to a submodule of the same name. This means you can use `Expr.Var` as both a constructor and in pattern matching.

2. Syntax

2.1 Basic syntax

The syntax of Amulet is primarily a mixture of OCaml and Haskell. It uses indentation to determine blocks of code.

2.1.1 Identifiers and Naming Conventions

Identifiers can be composed of most Unicode character. There are some restrictions:

- Identifiers cannot contain `[](),;"\`'`.
- Identifiers cannot start with a digit.
- Identifiers cannot contain the "Other" and "Separator"[1][2] character groups.
- The symbols `+-/\*.,:|` cannot be mixed with symbols not in that set.

You can also use double backticks to have a multi-word variable name: ``` ``This is a fancy variable name`` ```.

A period can be used to access child variables in modules.

⟨*start char*⟩ ::= a | b | c | ...

⟨*later char*⟩ ::= ⟨*start char*⟩ | 0 | 1 | 2 | ...

⟨*ident*⟩ ::= ⟨*start char*⟩ { ⟨*later char*⟩ }
| { '+' | '-' | ... }

⟨*name*⟩ ::= ⟨*ident*⟩ { '.' ⟨*ident*⟩ }

Amulet has several conventions for names:

- Modules, types and type constructors should be written using PascalCase.
- Variables, function names, type parameters and record field names should be lowercase.

2.1.2 Comments

Comments come in two forms: line comments (comments which last from the delimiter until a line break) and block comments (comments which last until the terminating block).

Line comments are started with `;`. Block comments are started with `(*` and finished with `*)`. These can be nested to allow easier commenting out of code.

You can document a binding using `;;` or `(** ... *)`. Documentation comment can be exported from a file to HTML with the contents rendered as Markdown.

[1] FileFormat.info 2016.
[2] Consortium 2016.

2.2 Types

2.2.1 Primary types

Types are composed of several key expressions:

⟨*type*⟩ ::= ⟨*name*⟩
| ⟨*type application*⟩
| ⟨*forall*⟩
| ⟨*constraint*⟩
| ⟨*type function*⟩
| ⟨*type tuple*⟩
| '`(`' ⟨*type*⟩ '`)`'
| '`_`'
| '`[`' ⟨*type*⟩ '`]`'
| '`[|`' ⟨*type*⟩ '`|]`'
| ⟨*record*⟩
| ⟨*row*⟩
| ⟨*union*⟩
| ⟨*unifies*⟩

⟨*var*⟩ ::= ⟨*ident*⟩
| '`(`' ⟨*ident*⟩ : ⟨*type*⟩ '`)`'

⟨*type application*⟩ ::= ⟨*type*⟩ ⟨*type*⟩
| ⟨*type*⟩ ⟨*name*⟩ ⟨*type*⟩
| ⟨*type*⟩ '`` `' ⟨*name*⟩ '`` `' ⟨*type*⟩

⟨*forall*⟩ ::= '`forall`' { ⟨*var*⟩ } '`.`' ⟨*type*⟩

⟨*constraint*⟩ ::= ⟨*type*⟩ '`=>`' ⟨*type*⟩

⟨*function type*⟩ ::= ⟨*type*⟩ '`->`' ⟨*type*⟩

⟨*tuple type*⟩ ::= ⟨*type*⟩ { '`*`' ⟨*type*⟩ }

⟨*unifies*⟩ ::= ⟨*type*⟩ '∼' ⟨*type*⟩

The *type application*, *forall*, *constraint* and *type function* rules are right associative, whilst *unifies* is non-associative.

What is the precedence of these?

2.2.2 Type annotations

- `:` is used to annotate a variable or expression having a type.
- `_` can be used to mark a type wildcard (not a type variable, just something you don't want to explicitly write). So you could have `[_]` to constrain it to be a list of something.

Rows and Unions

Rows are composed of a series of key-type pairs or a union of two rows:

⟨*row body*⟩ ::= {⟨*ident*⟩ '`:`' ⟨*type*⟩ '`,`'}
| ⟨*ident*⟩

⟨*row union*⟩ ::= ‘’
| ⟨*row_ body*⟩ { ‘|’ ⟨*row_ body*⟩ }

⟨*row*⟩ ::= ‘(’ ⟨*row union*⟩ ‘)’

What syntax are we using for union types? Braces are taken for record syntax.

Records

Record definitions are composed of a series of identifier keys to type pairs separated by commas within braces.

⟨*record*⟩ ::= ‘{’ ⟨*row union*⟩ ‘}’

2.2.3 Type declarations

Types can only be declared in a module. There are two forms of type declarations:

- Aliases: allowing referencing a longer type name as a shorter one (or renaming it if it conflicts or any other reason)
- Type definitions: used for creating sum and product types.

Type names

Type names are defined as an identifier followed by a series of free type variables and/or concrete types. The parameters that are specified within the RHS of the definition *must* exist on the LHS.

Type aliases

The `type` keyword is used to define a type alias:

⟨*type name def*⟩ ::= ⟨*ident*⟩ { ⟨*var*⟩ }
| ‘`op`’ (‘`left`’ | ‘`right`’) [⟨*digit*⟩] ⟨*var*⟩ ⟨*ident*⟩ ⟨*var*⟩
| ⟨*var*⟩ ‘`` ` ``’ ⟨*ident*⟩ ‘`` ` ``’ ⟨*var*⟩

⟨*type name*⟩ ::= ⟨*type name*⟩ [‘`:`’ ⟨*type*⟩]

⟨*type def*⟩ ::= ‘`type`’ ⟨*type name*⟩ ‘`=`’ ⟨*type*⟩
| ‘`type`’ ‘`foreign`’ ⟨*type name*⟩

Sum type definitions

Sum types are composed of one or more type-constructors, each followed by the types the product type is composed of.

⟨*product name*⟩ ::= ⟨*ident*⟩ {⟨*type*⟩}
| ⟨*type*⟩ ‘`(`’ ⟨*ident*⟩ ‘`)`’ ⟨*type*⟩
| ⟨*type*⟩ ‘`` ` ``’ ⟨*ident*⟩ ‘`` ` ``’ ⟨*type*⟩

⟨*product type*⟩ ::= ⟨*product name*⟩ [‘`:`’ ⟨*type*⟩]

⟨*sum type*⟩ ::= ‘`data`’ ⟨*type name*⟩ ‘`=`’ {‘`|`’ ⟨*product type*⟩ }
| ‘`data`’ ‘`foreign`’ ⟨*sum name*⟩

If the declaration starts on a new line it should have a leading ‘|’, otherwise not.

```
data X = A | B

data Y =
    | C Int
    | D
```

GADTs GADTs are defined as so:

```
data Foo a = Foo1 Int : Foo Int
           | Foo2     : Foo a
```

Type variables defined on the RHS are accessible on the left:

```
data Bar a = Bar1 a b : forall b . Foo a
           | Bar2     : Foo a
```

Type classes

Type classes are marked with the `class` keyword. You can then specify a series of constraints on various parameters before specifying the actual type. All constraints' parameters must appear in the type class's definition.

⟨*class constraint*⟩ ::= ⟨*type*⟩ ‘`=>`’

⟨*functional dep*⟩ ::= { ⟨*ident*⟩ } ‘`->`’ { ⟨*ident*⟩ }

⟨*binding*⟩ ::= ⟨*lhs*⟩ [‘`:`’ ⟨*type*⟩] [‘`=`’ ⟨*expr*⟩]

⟨*bindings*⟩ ::= ⟨*binding*⟩
| ⟨*binding*⟩ { ‘`and`’ ⟨*binding*⟩ }

⟨*class body*⟩ ::= ‘`type`’ ⟨*type name*⟩ [‘`=`’ ⟨*type*⟩]
| ‘`data`’ ⟨*type name*⟩ [‘`=`’ { ’|’ ⟨*product type*⟩ }]
| ‘`let`’ [‘`rec`’] ⟨*bindings*⟩

⟨*class*⟩ ::= ‘`class`’ { ⟨*class constraint*⟩ } ⟨*type*⟩ [‘`|`’ ⟨*functional dep*⟩ { ‘`,`‘ `<functional dep>` }] ‘`where`’
⟨*class body*⟩

⟨*instance*⟩ ::= ‘`impl`’ ⟨*ident*⟩ { ⟨*var*⟩ } ‘`=`’ ⟨*type*⟩ ‘`where`’ ⟨*class body*⟩

```
class Show a where -- or class Show a =
    let show : a -> String

impl X = Show String where
    let show x = "!" ++ x ++ "!"
```

The beginning of the type class's body is marked with the `where` keyword and an indent. The body contains a series of variables with type annotations with optional definitions. It is possible for all definitions to be filled and depend on one another: when creating an implementation Amulet will determine if you have provided sufficient information for a complete definition.

Type class implementations Implementations share a similar syntax to type class definitions, using `impl` instead of `class`. However, instead of following the type class with type parameters you can use any type expression.

The body of the implementation uses an identical syntax to that of type classes, though empty definitions are not allowed.

2.3 Expressions

2.3.1 Literals

Numeric literals

Numeric literals can be written as base 2, 8, 10 or 16 numbers.

Binary numbers are prefixed by 0b, hexadecimal numbers are prefixed with 0x, and octal numbers are prefixed with `0o`. None of the previous can contain an exponent or decimal place.

Base 10 numbers can contain a decimal point and an exponent.

A number can have infinitely repeated decimals after the decimal point (possibly directly) prefixed by a "~".

All numbers can be prefixed with a positive or negative sign, as well as allowing _between any two digits. This allows breaking a number up into nibbles or thousands.

Numbers with a decimal point are inferred to be of a type with an instance of `Coerce Rational a`, while numbers without a decimal point (though optionally with an exponent) are inferred to be types with an instance of `Coerce Integer a`.

Character literals

Characters are composed of a single character code between single quotes (`'`). A character code can be composed of:

- A single character (such as `a`)
- An escape character (`\'`, `\n`, `\r`, `\t`, `\"`)
- A decimal character code: `\123`
- A hex character code: `\xAB`
- The character can be prefixed with `u` to convert it to a unicode character. This also allows writing UTF8 literals:
- A UTF8 literal `\uABCD` This is a hexadecimal value: as many hex characters are consumed as possible. If a semicolon (`;`) then that will be consumed and the sequence exited.

String literals

Strings are written as a series of character codes between double quotes (`"`). Prefixing with a `u` will convert it to a unicode string, also allowing UTF8 literals.

Unit literal

An empty pair of parenthesis `()` are considered a unit value.

2.3.2 Primary expressions

⟨*expr*⟩ ::= ⟨*literal*⟩
| ⟨*expr*⟩ ‘`:`’ ⟨*type*⟩
| ⟨*apply*⟩
| ⟨*type application*⟩
| ⟨*lambda*⟩
| ⟨*list*⟩
| ⟨*vector*⟩
| ⟨*tuple*⟩
| ⟨*parens*⟩
| ⟨*record*⟩
| ⟨*if*⟩
| ⟨*case*⟩
| ⟨*match*⟩
| ⟨*let*⟩
| ⟨*do*⟩

Function application

Any two expressions following another are considered a function application. If functions are applied to a lesser number of arguments than they expect, this is partial application (which also means that functions are automatically curried).

Function application associates to the left. That is, `a b c d` is interpreted as `(((a b) c) d)`.

⟨*op*⟩ ::= ⟨*name*⟩
| ‘`` ` ``’ ⟨*name*⟩ ‘`` ` ``’

⟨*apply*⟩ ::= ⟨*expr*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| ‘`(`’ ⟨*expr*⟩ ⟨*op*⟩ ‘`)`’
| ‘`(`’ ⟨*op*⟩ ⟨*expr*⟩ ‘`)`’

Infix operators

Infix (binary) operators are implemented as functions which take two arguments. By convention a binary operator should exclusively be composed of symbols.

You can convert arbitrary functions of type `a -> b -> c` to infix operators by surrounding their name with backticks (`` `foo` ``).

Operator sections Operator sections are partially-applied binary operators, written in infix form for clarity. An operator can either be applied to their left or right operand, which gives rise to the two categories of operator sections: either *left* or *right*.

A right operator section is an operator applied to their left operand, as in `(e +)`, which is fully equivalent to `(λx. e + x)`. A left operator section is applied to their right operand, as in `(+ e)`, which is fully equivalent to `(λx. x + e)`.

Explicit type application

It is possible to apply a type to a function which takes type parameters (such as constraints for `forall`s). This allows explicitly specifying a particular type class implementation.

⟨*type application*⟩ ::= ⟨*expr*⟩ ‘`#`’ ⟨*type*⟩

Lambda expressions

Lambda expressions form the base of any functional language worth its salt. In Amulet, there are two ways to express a lambda abstraction: Either `\`x_1 `...` x_2 `->` `e` or λx_1 `...` x_2 `->` `e`.

Lambda expressions have the highest precedence of any expression.

⟨*lambda*⟩ ::= ('\' | 'λ') { ⟨*var*⟩ } '`->`' ⟨*expr*⟩

List literals

Linked lists are represented in Amulet through repeated usage of the `(::)` value constructor, which can get tiring. To aid in the creation of lists, a comma-separated list of expressions delimited by square brackets (`[]`) is interpreted as a list.

The methods `(::)` and `[]` are both variables within the `List` *h t r* class. This means additional forms of lists (such as HLists) can be constructed from the same syntax.

We need two type classes, one for cons, one for nil. Otherwise you'd have to implement `[]` for `List a (Vect n a) (Vect (S n) a)` which makes no sense. -cpup

Also related to list literals are vector literals. Vectors are, in Amulet, semantically equivalent to a Lua table or a C array; They contain a sequence of values in contiguous memory without pointers to the following/previous element. These, however, are delimited by square brackets with vertical bars on the *inner* side: While `[1, 2, 3]` is a list, `[| 1, 2, 3 |]` is a vector.

⟨*list contents*⟩ ::= ''
| ⟨*expr*⟩ { '`,`' ⟨*expr*⟩ } ['`,`']

⟨*list*⟩ ::= '`[`' ⟨*list contents*⟩ '`]`'

⟨*vector*⟩ ::= '`[|`' ⟨*vector contents*⟩ '`|]`'

Tuple literals

Tuples are composed of two or more values within a pair of parenthesis (0 values are considered a unit, 1 value is just for grouping expressions). Each expression within a tuple is separated with a comma.

⟨*unit*⟩ ::= '`(`' '`)`'

⟨*parens*⟩ ::= '`(`' ⟨*expr*⟩ '`)`'

⟨*tuple*⟩ ::= '`(`' [⟨*expr*⟩] '`,`' [⟨*expr*⟩] { '`,`' [⟨*expr*⟩] } '`)`'

Tuples can be sectioned by omitting values (leaving the commas), this pulls the value out to a lambda around the tuple: $(, v) : a \to (a, b)$ where $v : b$.

Record field access

Record's fields can be accessed using `record.field`.

⟨*record access*⟩ ::= '`.`' ⟨*ident*⟩
| '`(`' '`.`' ⟨*ident*⟩ '`)`'

Record literals

Record literals are composed of a series of comma separated, key-value pairs between matching braces.

```
let a = { ident1 = expr1, ident2 = expr2, ident3 = expr3 }
```

⟨*record field*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩

⟨*record fields*⟩ ::= ⟨*record field*⟩ { '`,`' ⟨*record field*⟩ } ['`,`']

⟨*record*⟩ ::= '`{`' '`}`'
| '`{`' ⟨*record fields*⟩ '`}`'

Record updates

Record updates let you create a new record based on an existing record with some updates.

```
let b = { a | ident1 = expr4, -ident2, ident3 $= expr5, ident4 <- expr6 }
```

Their syntax is:

⟨*record update*⟩ ::= '`{`' ⟨*expr*⟩ '`|`' ⟨*record update updates*⟩ '`}`'
| '`{`' '`|`' ⟨*record update updates*⟩ '`}`'
| '`{`' ⟨*record update updates*⟩ '`|}`'

⟨*record update updates*⟩ ::= ⟨*record update update*⟩ { '`,`' ⟨*record update update*⟩ } ['`,`']

⟨*record update update*⟩ ::= ⟨*record update remove*⟩
| ⟨*record update add*⟩
| ⟨*record update replace*⟩
| ⟨*record update modify*⟩

There are 4 types of updates:

Remove These remove a key from the record: $\{\, r \mid -x \,\} : \texttt{Record}\ p$ where $r : \texttt{Record}\ (x : a \mid p)$.

⟨*record update remove*⟩ ::= '`-`' ⟨*ident*⟩

Add These add a key to the record: $\{\, r \mid x \texttt{<-}\ v \,\} : \texttt{Record}\ (x : a \mid p)$ where $r : \texttt{Record}\ p$ and $v : a$.

⟨*record update add*⟩ ::= ⟨*ident*⟩ '`<-`' [⟨*expr*⟩]

Replace These change the value of an existing key: $\{\, r \mid x = v \,\} : \texttt{Record}\ (x : a \mid p)$ where $r : \texttt{Record}\ (x : b \mid p)$ and $v : a$.

⟨*record update replace*⟩ ::= ⟨*ident*⟩ '`=`' [⟨*expr*⟩]

Modify These modify the value of an existing key using a function: $\{\, r \mid x\ \$= f \,\} : \texttt{Record}\ (x : a \mid p)$ where $r : \texttt{Record}\ (x : b \mid p)$ and $f : b \to a$.

⟨*record update modify*⟩ ::= ⟨*ident*⟩ '`$=`' [⟨*expr*⟩]

For any of the updates that take an expression you can "section" them by omitting the expression, this pulls the expression out as another parameter to the block: $\{\, r \mid +x = \,\} \equiv \lambda v.\{\, r \mid +x = v \,\}$.

There are also two syntaxes for sectioning out the parent record: $\{|\ <\ updates\ >\} \equiv \{<\ updates\ >\ |\} \equiv \lambda r.\{r|\ <\ updates\ >\}$.

2.3.3 Control flow

If expressions

If expressions allow different expressions to be evaluated depending on the value of a condition. The condition *must* be of type `Bool`.

If statements are started with `if`, followed by a condition, the `then` keyword then the expression to be evaluated on a `True` value. This should be followed by the `else`

This expression can either appear on the same line or on a new line followed by an indent. Each expression's indentation are independent, so one can use a complex multiline expression and the other a single variable. This allows if expressions to be chained.

All branches' types must be unifiable.

```
-- Simple single line
print $ if a then b else c

-- Multiple line if statements
print $ if a then b
        else
           let c = doSomethingFancy a
               d = moreFancyThings
             in c d d

-- Chaining
print $ if a then b
        else if c then d
        else e
```

⟨*if*⟩ ::= '`if`' ⟨*expr*⟩ '`then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩

Cond expression

Amulet provides a way to simplify the writing of long `if-else if` chains. These expressions are started with the word `cond` (coming from the Lisp macro of the same name[3]) . This keyword is then followed by multiple lines of the form `| condition -> expression`.

All branches' types must be unifiable.

If Amulet cannot prove that the `cond` expression is total then a warning will be emitted and an fallback branch will be emitted which produces an error.

⟨*cond branch*⟩ ::= '`|`' ⟨*expr*⟩ '`->`' ⟨*expr*⟩

⟨*cond*⟩ ::= '`cond`' { ⟨*cond branch*⟩ }

Match expressions

Match expressions provide a way to branch based off of a series of patterns. They follow a similar syntax to `cond` expressions. They start with `match <expr> of` and are followed by multiple lines of the form `| pattern -> expression`. An optional guard can be placed after the condition (which must evaluate to a `Bool`).

All branches' types must be unifiable.

[3] Lispworks 2016.

If Amulet cannot prove that the `match` expression is total then a warning will be emitted and fallback branch will be emitted which produces an error.

⟨*match pattern*⟩ ::= '`|`' ⟨*pattern*⟩

⟨*match branch*⟩ ::= { ⟨*match pattern*⟩ } '`->`' ⟨*expr*⟩
| { ⟨*match pattern*⟩ } '`if`' ⟨*expr*⟩ '`->`' ⟨*expr*⟩

⟨*match*⟩ ::= '`match`' ⟨*expr*⟩ '`of`' { ⟨*match branch*⟩ }

2.3.4 Let bindings

Let bindings are used to create a scope with one new variable. These expressions are composed of the binding and the expression to evaluate with this new variable. This expression can either appear after an explicit `in` or on a new line with the same indentation level as the let expression.

Let bindings come in three key forms (destructing assignments, function creation and recursive function(s) creation). Let bindings are non-recursive by default: this requires you to think about dependencies more which makes you try to decouple things.

⟨*lhs*⟩ ::= ⟨*pattern*⟩
| '`op`' ('`left`' | '`right`') [⟨*digit*⟩] ⟨*pattern*⟩ ⟨*ident*⟩ ⟨*pattern*⟩
| ⟨*pattern*⟩ '`` ` ``' ⟨*ident*⟩ '`` ` ``' ⟨*pattern*⟩
| ⟨*ident*⟩ { ⟨*pattern*⟩ }
| '`(`' ⟨*ident*⟩ '`)`' { ⟨*pattern*⟩ }

⟨*binding*⟩ ::= ⟨*lhs*⟩ ['`:`' ⟨*type*⟩] '`=`' ⟨*expr*⟩
| '`foreign`' ⟨*ident*⟩ '`:`' ⟨*type*⟩

⟨*bindings*⟩ ::= ⟨*binding*⟩
| ⟨*binding*⟩ { '`and`' ⟨*binding*⟩ }

⟨*continue*⟩ ::= '`in`' ⟨*expr*⟩
| ⟨*newline*⟩ ⟨*expr*⟩

⟨*let*⟩ ::= '`let`' ['`rec`'] ⟨*bindings*⟩ ⟨*continue*⟩

Destructuring bind

A destructuring bind is simply composed of a pattern followed by an equals sign. This is simply sugar for a pattern matching expression:

```
let (x, y) = f
g $ x y
```

is equivalent to

```
match f with
| (x, y) -> g $ x y
```

As with `match` expressions a warning will be produced if the expression isn't total.

Function creation

Let expressions can be used to create functions. These can either be written in normal or infix form.

If in normal form then the function name is given followed by a series of arguments. This is followed by an equals sign and the function body. If the function is an operator then it must be surrounded by parentheses.

Infix form is similar, but the function name is given between the two arguments. If the function name is not an infix operator then it must be surrounded in backticks to convert it into one.

Recursive binds

The syntax for function creation can be expanded to allow recursive functions using the `rec` keyword. The body of the function will be defined in the scope including the variable.

Mutually recursive functions can be defined by separating functions with the `and` keyword.

Some examples

```
let op left x |> f = f x

let (a, b) = undefined

let a = 1
    b = b
1 + 2

let a = 1
    b = 2 in 1 + 2

let f x =
        let y = x + 1
            z = x - 1
        y - z
    g x = (+2)
f 2 + g 3
```

2.3.5 `do` notation

Do notation is simply sugar for `>>=` and `pure` over an instance of `Monad` α. All existing expressions are valid (though lets are changed) and an additional bind syntax is added.

`do` blocks are started with the `do` keyword. This can optionally be followed with a explicit type-class application specifying which implementation of `Monad`α to use.

⟨*do expr*⟩ ::= ⟨*pattern*⟩ '`<-`' ⟨*expr*⟩
| '`let`' ⟨*bindings*⟩
| ⟨*expr*⟩

⟨*do*⟩ ::= '`do`' ['`#`' ⟨*type*⟩] { ⟨*do expr*⟩ }

2.4 Patterns

Patterns are used in the left-hand-side of where and let bindings, in the clauses of matches and in the declarations of functions.

There are several different patterns that can occur in an Amulet program. Though this is a fixed set, they can be combined to match against more complicated data.

⟨*pattern*⟩ ::= ⟨*literal*⟩
| '`(`' ⟨*pattern*⟩ '`)`'
| '`_`'
| ⟨*var*⟩
| ⟨*var*⟩ '@' ⟨*pattern*⟩
| ⟨*constructor pattern*⟩
| ⟨*list pattern*⟩
| ⟨*vector pattern*⟩
| ⟨*tuple pattern*⟩
| ⟨*record pattern*⟩

2.4.1 Wildcards

Denoted by `_`, this pattern matches anything and does no binding. As an example, `let _= x in e` discards the value x (and is optimized to `e`). Formally, the set of free variables of an expression in the scope created by a `_` binder is the same as the set of free variables of the expression.

2.4.2 Literal Patterns

These match against a given literal, that is, a number, string, or character, and bind nothing. Matching on a string is equivalent to matching on a vector of characters. These patterns, however, are not free, as they entail an `Eq` constraint on the parameter.

2.4.3 Capturing patterns

These can be any valid identifier, matching anything and binding them to the given name. These are semantically equivalent to `name@_`.

2.4.4 At-patterns

At-patterns are a way to give an otherwise nameless pattern a name. An at-pattern will bind its name to the match of the pattern it was created from in the resulting scope. These take the form of `name@pattern`.

2.4.5 List and Vector Patterns

List pattern matching syntax exists as syntactic sugar for repeated matching against the `(::)` constructor. They match against a list of exact size. `[x,y,z]` is equivalent to `x::y::z::[]`, where `x`, `y` and `z` are patterns. For matching against a list of at least N elements, the syntax `[x,y,z] :- xs` is allowed, where `x`, `y`, `z`, and `xs` are patterns. This is equivalent to `x::y::z::xs`.

Like list constructing, the destructor is polymorphic across all types within the `List` *h t r* class, allowing matching in the same style as constructing.

The same sugar exists for vectors, but this is builtin rather than sugar. `[| x, y, z |]` matches against a 3-element vector, while `[| x, y, z |] :- xs` matches against a vector of *at least* 3 elements. The typical way of matching against the head/tail of a vector is `[|x|] :- xs`.

⟨*list pattern contents*⟩ ::= ''
| ⟨*pattern*⟩ { '`,`' `<pattern >` } ['`,`']

⟨*list pattern*⟩ ::= '`[`' ⟨*list pattern contents*⟩ '`]`'

⟨*vector pattern*⟩ ::= '`[|`' ⟨*vector pattern contents*⟩ '`|]`'
| ⟨*pattern*⟩ '`:-`' ⟨*pattern*⟩

2.4.6 Tuple patterns

Tuple patterns are used to extract multiple variables from tuples

⟨*tuple pattern*⟩ ::= '`(`' ⟨*pattern*⟩ { '`,`' ⟨*pattern*⟩ } ['`,`'] '`)`'

2.4.7 Constructor Patterns

Constructor patterns match against a value constructor, and thus are used for destructuring abstract data, such as lists. Obviously constructor patterns can be used to match against the multiple cases of sum types.

The basic form is `Name`. This matches against a nullary constructor. For constructors with more fields, the form is `(Name` p_1 `...` p_n`)`. This binds everything that the set of p_1 `...` p_n binds. For example, `(Ratio _ _)` doesn't bind anything.

As with functions, constructors can be matched (and applied) in infix form. For example, `(_ ` `` `Ratio` `` ` _)` is equivalent to `(Ratio _ _)`

⟨*constructor pattern*⟩ ::= ⟨*name*⟩ { ⟨*pattern*⟩ }
| ⟨*pattern*⟩ ⟨*name*⟩ ⟨*pattern*⟩
| ⟨*pattern*⟩ '`` ` ``' ⟨*name*⟩ '`` ` ``' ⟨*pattern*⟩

Record Patterns

Record patterns are used to match over records. They share the same syntax as record constructors, though with pattern on the RHS rather than expressions:

⟨*record field ptrn*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*expr*⟩

⟨*record fields ptrn*⟩ ::= ⟨*record field ptrn*⟩ { '`,`' ⟨*record field ptrn*⟩ } ['`,`']

⟨*record ptrn*⟩ ::= '`{`' '`}`'
| '`{`' ⟨*record fields ptrn*⟩ '`}`'
| '`{`' ⟨*ptrn*⟩ '`|`' ⟨*record fields ptrn*⟩ '`}`'

It is worth noting that this pattern requires that the specified fields exist on the matchee. For instance the following would not typecheck:

```
match { x = 1 } with
| { x = a, y = b } -> sqrt $ a^2 + b^2
| _ -> 0
```

As the `y` field does not appear on the matchee.

2.5 Modules

2.5.1 Importing modules

Modules can be imported with the `open` keyword. It is possible to partially import modules and import them to a child variable.

⟨*open*⟩ ::= '`open`' ⟨*name*⟩
| '`open`' ⟨*name*⟩ '`as`' ⟨*ident*⟩
| '`open`' ⟨*name*⟩ '`with`' '`(`' { ⟨*ident*⟩ ['`as`' ⟨*ident*⟩] }'`)`'

2.5.2 Exporting definitions

All definitions can have have their access modifier placed after the declaration "type" but before the main definition. An example would be for let bindings:

```
let private x = 2
```

It is also possible to re-export imported modules with the `export` keyword.

⟨*access modifier*⟩ ::= '`public`' | '`internal`' | '`private`'

⟨*export*⟩ ::= '`export`' [⟨*accces modifier*⟩] ['`type`' | '`data`'] ⟨*name*⟩

Or put access modifier on the import keyword? We need a way of only exporting some entries: maybe allowing `as` and `with`?
Also worth looking at Idris' way of doing things - `http://docs.idris-lang.org/en/latest/tutorial/modules.html` Especially the `export`/`public export` distinction.

2.5.3 Defining modules

Modules are defined with the `module` keyword, followed by the module name and its body. Normally the module's body should be indented. However if the module definition is the first expression in the file then no indentation is required.

⟨*module*⟩ ::= '`module`' ⟨*name*⟩ '`=`' ⟨*module_ body*⟩

⟨*module body*⟩ ::= ⟨*type def*⟩
| ⟨*sum def*⟩
| ⟨*module*⟩
| ⟨*class*⟩
| ⟨*instance*⟩
| ⟨*expr*⟩

3. Runtime

3.1 Run-time and Compile-time Semantics

3.1.1 The Runtime System

Since Amulet is a pure language, there are many things that are fundamental to writing programs that could not be implemented within it, the most important of which being I/O. Seeing as there is no way of expressing I/O actions in a pure language, we expose untyped bindings from the runtime system, and have *interface modules* link against those.

The runtime system, or *RTS* for short, provides, along with handle-based I/O,

- Automatic memory management (garbage collection) that can handle cyclic data structures.
- An efficient linear data structure for high-performance code where linked lists are inappropriate (The `Vector` α type).
- Bindings to various big number frameworks (bigint, GMP, etc...), for arbitrary precision numbers.
- Operations on unboxed numbers (integers and doubles).
- Miscellaneous functions for management of thunks.
- Support for infinite recursion.

The compiler knows about RTS functions, and will insert invocations to them where appropriate. Such applications are marked with the compiler pragma `RTS`, and can only be present in modules not marked `Safe`, as they are intrinsically unsafe.

Though `Safe` modules can't use RTS functions, there are bindings to all of the functions which may be considered safe, such as `force#` (which evaluates a thunk unconditionally), in safe modules. The binding for `force#` is `force` : `Lazy` $\alpha \to \alpha$.

3.1.2 Laziness

Amulet, though strict by default, has standard library functions and compiler extensions to deal with laziness.

Notation

Before introducing the distinction between `Lazy` and `Strict` modules, we present a useful notation for strictness analysis.

Strictness analysis mostly follows reduction rules that involve divergent terms, which are denoted by $\bot$. These such terms will halt evaluation with an error message, as is the case with `error` or `undefined`, or reduce into themselves, as is the case with the paradoxical ω combinator, $(\lambda x.x\ x)\ (\lambda x.x\ x)$. Though the ω combinator is not well-typed, similar looping terms are, such as `let rec` $x\ =\ x$ `in`x.

A function can be *strict* in its argument. This is indicated by the following equivalence, which says that f *applied to* $\bot$ *is* $\bot$.

$$f\ \bot\ =\ \bot$$

However, if a function is *lazy* in its argument, when that function is applied to $\bot$, a concrete result is returned. For example, given the function `f` $\tilde{\texttt{x}} = 1$, then $f\ \bot \equiv 1$.

Strict and Lazy Modules

Amulet modules can be annotated through the use of compiler pragmas, such as the `LAZY` or `STRICT` pragmas, which control strictness.

In a module that is strict by default (that is, every module without an annotation), then all lambda expressions are automatically strict unless marked with a lazy pattern (this includes top-level combinators, as strictness is handled in the desugared form). Formally,

$$(\lambda p.e)\ \bot \mapsto \bot\ \forall\ p \text{ where } p \text{ is not lazy}$$

In lazy modules, however, all patterns are automatically assumed lazy, unless explicitly marked strict (by use of a bang-pattern, for example). This also has the consequence that any type variable α is converted to a lazy type, such as `Lazy` α. This maintains the laziness properties across module boundaries, and has no observable side effects, since `Lazy` $\alpha \sim \alpha$.

Thunks

Thunks are the representation of a suspended - in other words, *lazy* - computation. A thunk represents a value that is yet to be evaluated. Since laziness is marked in a type (A lazy value doesn't have type α, instead having the type `Lazy` α), there is no added overhead checking if a value is a thunk.

The Amulet runtime takes advantage of the fact that thunks are uniformly represented as pointers to do *thunk sharing*, which reduces duplication of work. While a naïve compiler might generate two separate thunks for two expressions that are semantically equivalent, the Amulet compiler will not. Since these thunks are shared, they'll only be evaluated once.

As an example of thunk sharing, we present an Amulet combinator representing a computation to be executed twice, and how that combinator would be simplified to share most computations.

$$\texttt{let}\ f\ \_\ =\ (2+2)+(2+2)$$

As the expression $(2+2)$ is used twice in exactly the same terms, it is simple to notice that this work can be shared, by introducing a new, fresh let binding and giving that as the argument to the top-level $(+)$.

$$\texttt{let}\ f\ \_\ = \texttt{let}\ a = (2+2)\ \texttt{in}\ a + a$$

Given this definition, we can see how $(2+2)$ is now only computed once. Thunk sharing is used in Amulet along with *Thunk replacement*, which replaces a thunk for x with the normal form of x.

3.2 Standard Library

- Numbers (Num, Integral, Fractional)
- Debugging (Show/Debug, Read?, debug)
- Iterables (Traversable, Foldable)
- Binary, Text
- IO, Async
- Errors (Maybe, Either)
- Monads (State, Reader, Writer)
- General functions (id, undefined, error)

- Avoid duplication at all costs (‘m (m a) -> m a‘ is better than ‘[[a]] -> [a]‘)
- Monad Transformers or Freer?
- If Lazy: ‘force : Lazy a -> a‘
- Number hierarchy: demhydraz recommends `https://tonyday567.github.io/tower/index.html`
- Operators and their precedence

3.2.1 Strings

```
class Binary a where
  toString : a -> String Byte
  fromStringHead : String Byte -> (a, String Byte)
  fromStringTail : String Byte -> (String Byte, a)
  -- not entirely sure about these deconstructing functions

instance Binary Byte where
  MAGIC

data String a = Binary a => MAGIC
```

So then `Binary` would be `String Byte` and `Text` (or `BinaryString` and `UnicodeString`) would be `String CodePoint` (or maybe `String UTF8`)

> This would require codepoints to be a fixed size right? I haven't fully decided how to handle strings yet: the problem with UTF8 is that indexing them isn't $O(1)$.
>
> No, `toString` can generate any length string and `fromStringHead` just needs to parse the first code point
>
> There is a slight problem with `fromStringTail`: I don't know if you can parse UTF8 in reverse

4. The future

4.1 Extensions to Amulet

This catalogues various extensions which could be added to Amulet on a later date.

4.1.1 Active patterns

One of the nice things about F♯ is the ability to define custom patterns (such as `http://stackoverflow.com/a/2429874`).

Haskell has a similar feature (`https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#view-patterns`) known as View patterns.

4.1.2 Disposable types

F♯ adds the `use` binding. This will automatically call the destructor on the object when it falls out of scope. This would be really nice for things like file handles or sockets, though tricky as it has side effects. It could potentially be emulated by having a `withFile` method which acquires a file handle, applies a function over it, and destroys the handle.

4.1.3 Dependent types

Merge the type and term level to allow applying functions over types and using functions within types.

That's not dependent typing (though it often goes along with it). You just need types that depend on values. like $(x : \tau_1) \to \tau_2$ and $(x : \tau_1 ** \tau_2)$ which are universal quantification and existential quantification respectively. I think technically just merging the type and term namespaces would actually give us dependent types. -cpup

Bibliography

Consortium, Unicode (2016). "Unicode Data". In: URL: http://www.unicode.org/Public/UNIDATA/UnicodeData.txt (visited on 10/05/2016).

FileFormat.info (2016). "Unicode Categories". In: URL: http://www.fileformat.info/info/unicode/category/index.htm (visited on 09/25/2016).

Kiselyov, Oleg, Amr Sabry, and Cameron Swords (2013). "Extensible Effects: An Alternative to Monad Transformers". In: *SIGPLAN Not.* 48.12, pp. 59–70. ISSN: 0362-1340. DOI: 10.1145/2578854.2503791. URL: http://doi.acm.org/10.1145/2578854.2503791.

Lispworks (2016). "Lisp cond Macro documentation". In: URL: http://www.lispworks.com/documentation/HyperSpec/Body/m_cond.htm#cond (visited on 09/28/2016).

Wiki, Haskell (2016a). "Functional Dependencies". In: URL: https://wiki.haskell.org/Functional_dependencies (visited on 09/20/2016).

– (2016b). "Type families". In: URL: https://wiki.haskell.org/Type_families (visited on 10/13/2016).

Wikipedia (2016). "Kind (type theory)". In: URL: https://en.wikipedia.org/wiki/Kind_(type_theory) (visited on 09/25/2016).