A Cogent Manual

Gunnar Teege
gunnar.teege@unibw.de

Version 0.9

I wrote this manual when I had my first contact with Cogent and wanted to get a clear view of its syntax and its concepts. For me the best way to do so is by writing it down in an organized way. Since I found that there was no similar documentation, I decided to do so in a form that it may be usable (and hopefully useful) for others as well.

Zilin Chen has read the manual very carefully and has pointed me to many issues, so I was able to correct and improve it extensively. Many thanks to him for his commitment.

This manual is not intended as a tutorial for programming in Cogent or to prove properties of Cogent programs. The examples are not chosen to be realistic, they are only used for illustrating the syntax. The manual only describes the Cogent "surface syntax" which is the interface for the programmer. Note that most publications about Cogent refer to the more concise "core syntax" which is created from the surface syntax by applying the "desugaring rules".

Chapter 1

Lexical Syntax

The basic lexical items in Cogent are comments (including document blocks and pragmas), identifiers (including reserved words), symbols and literals. Additionally in a Cogent program the usual Haskell preprocessor directives can be used, which are similar to the C preprocessor directives.

1.1 Comments

Comments have the same form as in Haskell. A comment may either be a line comment starting with the symbol `--` and ending at the end of the line, or it may be a block comment enclosed in `{-` and `-}`.

Examples of comments are:

```
f: U8 -> U8 -- this is a line comment after a function signature
f x {- x is the function argument -} = x+1
  -- function f returns its incremented argument
```

Block comments may occur in a Cogent source between all other lexical entities. Block comments can be nested, the closing brace of the inner comment does not end the outer comment:

```
{- This is a comment with a nested {- inner comment. -}
   After it the outer comment continues. -}
```

A special form of comments are document blocks. They have a similar form like line comments but start with the symbol `@`. Document blocks can be used to generate an HTML documentation from a Cogent source:

```
@@ # Heading
@@ This is a standalone documentation block

@ Documentation for the following function
f: U8 -> U8
f x = x+1
```

Another special form of comments are pragmas, they have the form

```
{-# ... #-}
```

Pragmas are used to optimise Cogent programs and to interface external C components. The details of pragmas are not (yet?) covered by this manual.

1.2 Identifiers

Identifiers are used to name items in a program. As usual in programming languages, they consist of a sequence of letters and digits beginning with a letter.

Cogent syntactically distinguishes between lowercase identifiers and capitalized identifiers.

LowercaseID: informal
  A sequence of letters, digits, underscore symbols, and single quotes starting with a lowercase letter

CapitalizedID: informal
  A sequence of letters, digits, underscore symbols, and single quotes starting with an uppercase letter

The underscore symbol `_` and the single quote `'` may appear in identifiers but not at the beginning. Examples for valid identifiers are `v1`, `very_long_identifier`, `CamelCase`, `T`, `W_`, and `v'`.

Lowercase identifiers are used for record field names and for term variables and type variables. Capitalized identifiers are used for type constructors and data constructors.

There are some *reserved words* in Cogent wich syntactically are identifiers but cannot be used as identifiers. The reserved words are in alphabetical order:

```
all and complement else False if in include let not put
take then True type
```

1.3 Literals

There are four kinds of literals in Cogent.

1.3.1 Boolean Literals

The boolean literals are the reserved words `True` and `False`.

BooleanLiteral: one of
  `True False`

1.3.2 Integer Literals

Integer literals are digit sequences. They can be written in decimal, hexadecimal. or octal form.

IntegerLiteral:
  DecDigits
  `0x` *HexDigits*
  `0X` *HexDigits*
  `0o` *OctDigits*
  `0O` *OctDigits*

DecDigits: informal
  A sequence of decimal digits 0-9.

HexDigits: informal
  A sequence of hexadecimal digits 0-9, A-F.

OctDigits: informal
  A sequence of octal digits 0-7.

1.3.3 Character Literals

A character literal consists of a quoted character.

CharacterLiteral: informal
  A character enclosed in single quotes.

The type of a character literal is `U8` (see below), which corresponds to a single byte. Syntactically, a character literal can be specified as in Haskell (see the Haskell Report), i.e., full Unicode and several escape sequences (such as `n`) are supported. However, a valid character literal in Cogent must always correspond to a code value in the range 0..255.

Examples for valid character literals are `'h'`, `'8'`, and `'/'`. The quoted character `'\300'` is not a legal character literal since it is mapped to code 300.

1.3.4 String Literals

A string literal consists of a quoted character sequence.

StringLiteral: informal
  A sequence of characters enclosed in double quotes.

Syntactically a string literal can be specified as in Haskell (see the Haskell Report). The same escape sequences as for character literals are supported for specifying every character. For a valid Cogent string literal every character must be mapped to a code in the range 0..255.

An example for a valid string literal is the string `"This is a string literal\n"`. Again, the string `"String containing a \300 glyph"` is not legal, since it contains a character mapped to code 300.

Chapter 2

Types

The Cogent language is a strongly typed language, where every term and variable in a program has a specific type. Like some other strongly typed languages, such as Scala and several functional languages types are often automatically inferred and need not be specified explicitly. Although possible in many cases, Cogent never infers types for toplevel definitions (see Section 4.2), they must always be specified explicitly.

In most typed programming languages a type only determines a set of values and the operations which can be applied to these values. As a main feature of Cogent, types are extended to also represent the way how values can be used in a program.

2.1 Type Basics

We will first look at the basic features of the Cogent type system, which are similar to those of types in most other programming languages. There are some predefined *primitive* types and there are ways to construct *composite* types from existing types.

Note that in Cogent types can always be specified (e.g. for variables or function arguments) by arbitrarily complex *type expressions.* It is possible to use a *type definition* to give a type a name, but it is not necessary to do so. In particular, types are matched by structural equality, hence if the same type expression is specified in different places it means the same type.

The general syntactical levels of type expressions are as follows:

MonoType:
    TypeA1
    ...

TypeA1:
    TypeA2
    ...

TypeA2:
    AtomType
    ...

AtomType:
  (*MonoType*)
  . . .

By putting an arbitrary *MonoType* expression in parentheses it can be used wherever an *AtomType* is allowed in a type expression.

2.1.1 Primitive Types

Since Cogent is intended as a system programming language, the predefined primitive types are mainly bitstring types. Additionally, there is a type for boolean values and an auxiliary string type.

Syntactically, a primitive type is specified as a nullary type constructor:

AtomType:
  (*MonoType*)
  TypeConstructor
  . . .

TypeConstructor:
  CapitalizedId

Bitstring Types

The bitstring types are named `U8`, `U16`, `U32`, and `U64`. They denote strings of 8, 16, 32, or 64 bits, respectively.

The usual bitstring operations can be applied to values of the bitstring types, such as bitwise boolean operations and shifting. Alternatively, bitstring values can be interpreted as unsigned binary represented numbers and the corresponding numerical operations can be applied. All numerical operations are done modulo the first value that is no more included in the corresponding type. E.g., numerical operations for values of type `U8` are done modulo $2^8 = 256$.

TODO: application of && and ||?

Other Primitive Types

The other primitive types are `Bool` and `String`. Type `Bool` has the two values `True` and `False` with the boolean operations. Type `String` can only be used for specifying string literals, it supports no operations.

2.1.2 Composite Types

There are the following possibilities to construct composite types: records, tuples, variants, and functions.

Tuple Types

A tuple type represents mathematical tuples, i.e., values with a fixed number of fields specified in a certain order. Every field may have a different type. A tuple type expression has the syntax:

AtomType:
  `(` *MonoType* `)`
  TypeConstructor
  TupleType
  ...

TupleType:
  `()`
  `(` *MonoType* `,` *MonoType* { `,` *MonoType*} `)`

The empty tuple type `()` is also called the *unit* type. It has the empty tuple as its only value.

Note that there is no 1-tuple type, a tuple type must either be the unit type or have at least two fields. Conceptually, a 1-tuple is equivalent to its single field. Syntactically the form `(` *MonoType* `)` is used for grouping.

The type expression `(U8, U16, U16)` is an example for a tuple type with three fields.

Record Types

A record type is similar to a C struct, or a Haskell data type in record syntax. It consist of arbitrary many *fields*, where each field has a name and a type. Accordingly, a record type expression has the following syntax:

AtomType:
  `(` *MonoType* `)`
  TypeConstructor
  TupleType
  RecordType
  ...

RecordType:
  `{` *FieldName* `:` *MonoType* { `,` *FieldName* `:` *MonoType*} `}`

FieldName:
  LowercaseId

The fields in a record type are order-sensitive. Therefore, the type expressions `{a: U8, b: U16}` and `{b: U16, a: U8}` denote different types. A record type must always have atleast one field. Other than for tuples, a record type may have a single field. Therefore, the type expressions `{a: U8}` and `U8` denote different types.

Variant Types

A variant type is similar to a union in C, or an algebraic data type in Haskell. As in Haskell, and unlike in C, a variant type is a *discriminated* union: each alternative value set is tagged with the variant it belongs to.

A variant type specifies for every variant the tag and the types of the values, hence it has the syntax:

AtomType:
  `(` *MonoType* `)`
  TypeConstructor
  TupleType
  RecordType
  VariantType
  ...

VariantType:
  `<` *DataConstructor {TypeA2}* { `|` *DataConstructor {TypeA2}*} `>`

DataConstructor:
  CapitalizedId

The tags are given by the *DataConstructor* elements. Each variant has a sequence of values, hence the ordering of the *TypeA2* matters.

The type expression `<Small U8 | Large U32>` is an example for a variant type with two variants. Typical applications of variant types are for modelling error cases, such as in

```
<Ok U16 U32 U8 | Error U8>
```

or for modelling optional values, such as in

```
<Some U16 | None>
```

Although *DataConstructor*s and *TypeConstructor*s have the same syntax, they constitute different namespaces. A *CapitalizedId* can be used to denote a *DataConstructor* and a *TypeConstructor* in the same context. In the example

```
<Int U32 | Bool U8>
```

the name of the predefined primitive type `Bool` is also used as a tag in a variant type.

Function Types

A function type corresponds to the the usual concept of function types in functional programming languages, as it is even available in C. A function type has the syntax:

MonoType:
  TypeA1
  FunctionType

FunctionType:
  TypeA1 `->` *TypeA1*

A function with type `U8 -> U16` maps values of type `U8` to values of type `U16`.

Note, that a *TypeA1* cannot be a function type. Hence, to specify a higher order function type in Cogent, which takes a function as argument or returns a functions as result, the argument or result type must be put in parentheses.

In particular, the type expression `U8 -> U8 -> U16`, which is the usual way of specifying the type of a binary function in Haskell through currying, is illegal in Cogent. Strictly speaking, function types always describe unary functions. To specify the corresponding type in Cogent use `U8 -> (U8 -> U16)`. Alternatively, a type expression for a binary function can be specified as `(U8,U8) -> U16` in Cogent, which is a different type.

2.1.3 Type Definitions

Although all types in Cogent could be denoted by type expressions, types can be named by specifying a *type definition*. In the simplest case, a type definition introduces a name for a type expression, such as in the following example:

```
type Fract = { num: U32, denom: U32 }
```

Syntactically a type name is a *TypeConstructor* in the same way as the primitive types. Hence, the primitive types can be considered to be specific "predefined" type names.

A type name defined in a type definition may be used in type expressions after the definition but also in type expressions occurring *before* the type definition. In this way type definitions are "global", the defined type names can be used everywhere in the Cogent program, also in and from included files.

An important restriction of Cogent is that type definitions may not be recursive, i.e., the type name may not occur in the type expression on the right-hand side. Thus the following type definition is illegal

```
type Numbers = <Single U32 | Sequ (U32, Numbers)>
```

because the defined type name `Numbers` occurs in the type expression. Also there may not be an indirect recursion, where type definitions refer to each other cyclically.

Generic Types

In a type definition it is also possible to define a *TypeConstructor* which takes one or more *type parameters*. Such a *TypeConstructor* is called a *generic* type. An example would be

```
type Pair a = (a,a)
```

Here, the *TypeConstructor* `Pair` is generic, it has the single type parameter `a`.

In fact, a generic type like `Pair` is not really a type, it is a type *constructor*. Only when it is applied to type *arguments*, such as in `Pair U32`, it yields a type. Such a type is called a *parameterized type*. Every generic type has a fixed *arity*, which is its number of type parameters and specifies the number of arguments required in parameterized types constructed from it.

A *TypeConstructor* is non-generic, if it has arity 0. In this special case, the *TypeConstructor* itself already denotes a type.

Generic types in Cogent are known in Haskell as "polymorphic types" and similar concepts can be found in several other programming languages. In Java, a generic class definition has the form `class Pair<A> {...}`, it defines the generic class `Pair` with its type parameter `A`. In C++ a similar concept is supported by "templates".

The syntax for a type definition in Cogent supports both generic and non-generic types:

TypeDefinition:
  `type` *TypeConstructor {TypeVariable}* `=` *MonoType*
  ...

TypeVariable:
  LowercaseId

A *TypeConstructor* defined this way is also called a *type synonym*, since as a type expression it is strictly equivalent to the expression on the right-hand side in the definition. A type synonym with arity 0 is called a *type name*.

In the definition of a generic type, the type parameters may occur in the *MonoType* on the right-hand side. There they are called *type variables* and a type expression containing type variables is called a *polymorphic type*. To support polymorphic type expressions, the syntax allows type variables as *AtomType*:

AtomType:
  TypeConstructor
  TupleType
  RecordType
  VariantType
  TypeVariable

As in Haskell there is no syntactic difference between type variables and normal (term) variables. However, type variables are syntactically different from type constructors, since the latter are capitalized identifiers, whereas variables begin with a lowercase letter.

Since type variables are allowed as *AtomType*, they can occur in a polymorphic type expression in all places where a type is allowed.

Note that in the definition of a generic type, all type variables occurring in the type expression on the right-hand side must be type parameters, declared on the left-hand side, i.e., they must all be bound in the type definition. The other way round, a type parameter need not occur as type variable in the type expression. In Haskell, this is called a "phantom type". Other than in Haskell in Cogent these types are not checked by the type checker, hence for

```
type A a = U8
```

the types `A U16` and `A Bool` are equivalent.

Parameterized types are simply denoted by the generic type constructor followed by the required number of type expressions as arguments, such as in

```
Pair U32
```

They can be used in type expressions as *TypeA1*:

TypeA1:
  TypeA2
  ParameterizedType
  ...

ParameterizedType:
  TypeConstructor {TypeA2}

Note that parameterized types must be put in parentheses if they are nested (used as argument of another parameterized type).

Expanding Type Expressions

We call a parameterized type with a type synonym as *TypeConstructor* a *parameterized type synonym*.

Since type definitions may not be recursive, type synonyms can always be eliminated from type expressions by substituting the defining type expression for them, putting it in parentheses if necessary.

In the case of a parameterized type synonym also the type variables are substituted by the actual type arguments. We call the result of eliminating (transitively) all type synonyms from a type expression the *expansion* of the type expression.

Abstract Types

An *abstract* type is similar to a type synonym without a definition. The idea of abstract types in Cogent is to provide the actual definition externally in accompanying C code. Hence abstract types are the Cogent way of interfacing C type definitions. However, since abstract types are used in Cogent in an opaque way, it is not necessary to know the external C definition for working with an abstract type in Cogent. Note that abstract types are not meant to be used as interfaces to or abstractions of other Cogent types.

Abstract types can be generic, i.e., they may have type parameters. The names of these type parameters are irrelevant, since there is no definition where they could occur as type variables. They are only used to specify the arity of the generic abstract type.

The syntax for defining abstract types is the same as for normal type definitions, with the defining type expression omitted:

TypeDefinition:
  `type` *TypeConstructor {TypeVariable}* `=` *MonoType*
  AbstractTypeDefinition

AbstractTypeDefinition:
  `type` *TypeConstructor {TypeVariable}*

The following examples define two abstract types. Type `Buffer` is non-generic, type `Array` is generic with arity 1:

```
type Buffer
type Array a
```

Like generic type synonyms, generic abstract types can be used to construct parameterized types:

```
Array U16
```

We call a parameterized type with an abstract type as its *TypeConstructor* a *parameterized abstract type*. Note that abstract types cannot be eliminated by expanding a type expression, since they have no definition.

2.2 Restricted Types

A type semantically determines a set of values as its extension. In most other typed programming languages the main consequence is that the type of a value restricts the functions which can be applied to it.

A specific feature of Cogent is that the type may impose additional restrictions on the ways a value can be used in the program, in particular, how *often* it may be used. This concept is known as *linear types*, it is also present in some other special programming languages, e.g., in Rust.

Many types in Cogent do not impose additional restrictions, they behave like types in other programming languages, we call them *regular types.* Types with additional restrictions are called *restricted types.*

2.2.1 Linear Types

One kind of restricted types are *linear types.* A linear type has the specific property, that its values must be used *exactly once* in the program. What this means is explained in Section 3.3. Here it is only relevant that a type may be linear or not.

Linearity is an inherent property of type expressions. Type expressions as they have been described until now can either be linear or regular. To determine whether a type expression is linear or regular its expansion is inspected using the following rules:

- Primitive types are regular.
- Record types are linear.
- Tuple types are linear if they contain at least one field with a linear type.
- Variant types are linear if they contain at least one variant with a linear type.
- Function types are regular.
- Parameterized and non-generic abstract types are linear.

Together, a type is linear when, after expanding all type synonyms, it has a component of a record or abstract type which does not appear as part of a function type.

2.2.2 Boxed and Unboxed Types

In order to decouple the property of linearity somewhat from the way how types are composed, the concept of *unboxed types* is used. Record types and abstract types, which may cause a type to be linear, are called *boxed types*, the other types (primitive, tuple, variant, and function) are called *unboxed types.*

The type system is expanded by introducing the unbox type operator `#`. For boxed types it produces an unboxed version. By applying the unbox type operator to all record types and abstract types in a type expression, the type expression becomes regular.

The operator `#` is applied to a type expression as a prefix. To simplify the syntax it is allowed to be applied to arbitrary *AtomType* expressions:

TypeA2:
  AtomType
  `#` *AtomType*
  ...

By putting an arbitrary *MonoType* in parentheses, the unbox operator can be applied to it, as in `#(Array U8)`.

If the unbox operator is applied to an *AtomType* which is already unboxed, it has no effect. Hence, the type expressions `(U8,U16)` and `#(U8,U16)` denote the same type, whereas `{fld1:U8,fld2:U16}` and `#{fld1:U8,fld2:U16}` denote different types.

When applied to a record, the unbox operator affects only the record itself, not its fields. Hence, an unboxed record is still linear, if it has linear fields. The additional linearity rules for types resulting from applying the unbox operator are

- Unboxed record types are linear if they contain at least one field with a linear type.
- Unboxed non-generic or parameterized abstract types are regular.
- For all other cases, an unboxed type is linear or regular according to the linearity of the type expression to which the unbox operator is applied.

As an example, if `A` is a non-generic abstract type, the type expression `#(U8,A)` is linear, since the linear second field makes the type expression `(U8,A)` linear.

2.2.3 Partial Record Types

Since record types are linear, their values must be used exactly once, which also uses all their linear fields. To support more flexibility, Cogent allows using linear record fields independently from the record itself, although each of them must still be used exactly once. This is done by separating the linear field's value from the rest of the record. The fact that the field value is no more present in the remaining record is reflected by the remaining record having a different type. These types are called *partial record types.* A record field for which the value is not present is called a *taken* field.

A partial record type is denoted by specifying a record type together with the names of the taken fields using the following syntax:

TypeA1:
  TypeA2
  TypeConstructor {TypeA2}
  PartialRecordType

PartialRecordType:
  TypeA2 TakePut TakePutFields

TakePut: one of
  `take put`

TakePutFields:
  FieldName
  (*[FieldName { , FieldName}]*)
  (..)

Thus `take` and `put` together with field names constitute type operators. The result of applying these type operators is usually a partial record type.

When applied to a type R the operator `take (v,w)` produces the record type where at least fields `v` and `w` are taken, in addition to the fields that have already been taken in R. If the fields `v` and `w` are already taken in R, the compiler produces a warning. If R has no such fields then applying the take operator is illegal.

The operator `put (v,w)` is dual to the take operator, it produces the record type where at least the fields `v` and `w` are *not* taken, in addition to the fields that have not been taken in R. If the fields `v` and `w` are not taken in R, the compiler produces a warning. If R has no such fields then applying the put operator is illegal.

The operator `take ( .. )` produces a record type where all fields are taken, the operator `put ( .. )` produces the record type where no field is taken. Applying it to a type which is not a (boxed or unboxed) record type is illegal.

If a take or put operator is applied to a boxed record type the result is again boxed, if applied to an unboxed record type the result is unboxed.

Consider the following examples:

```
type A
type B
type C
type R1 = {fld1: A, fld2: U8, fld3: B, fld4: C}
type R2 = R1 take fld1
type R3 = R1 take ( .. )
```

Types `R1, R2, R3` are all boxed and thus linear. The type expressions

```
R1 take (fld1, fld2)
R2 take (fld1, fld2)
R2 take fld2
R3 put (fld3, fld4)
```

are all equivalent. The type expressions `R3 put ( .. )` and `R2 put ( .. )` are both equivalent to type `R1`.

An unboxed record type without linear fields is regular. The same holds for unboxed partial record types if all linear fields are taken. Thus the additional linearity rules for partial record types are

- Partial boxed record types are linear.
- Partial unboxed record types are linear if they contain at least one non-taken field with a linear type.

2.2.4 Readonly Types

Since the restrictions for using values of a linear type are rather strong, Cogent supports an additional kind of types, the *readonly types.* The use of values of a readonly type is also restricted, however, in a different way: they can be used any number of times but they may not be modified. Again, the meaning of this is explained in Section 3.3.

The bang Operator

All type expressions defined until now are not readonly. The only way to construct a readonly type is by applying the type operator `!`, which is pronounced "bang". This operator may be applied to an *AtomType* in postfix notation:

TypeA2:
   AtomType
   `#` *AtomType*
   AtomType `!`

By putting an arbitrary *MonoType* in parentheses the bang operator can be applied to it.

Readonly types are considered as an alternative to linear types, hence regular types are never readonly: If the bang operator is applied to a regular type A the resulting type is equivalent to A. Only if the bang operator is applied to a linear type a readonly type may result.

Unlike the unbox operator the bang operator also affects subexpressions such as record fields and abstract types. If in type A a field has type F then in type A! the same field has type F!. An exception are function types: if a bang operator is applied to a function type it is not applied to argument and result types. As a result of this recursive application of the bang operator, it turns every linear type into a non-linear type.

Escape-restricted Types

A concept related to readonly types are *escape-restricted types.* A type is escape-restricted if it is readonly or if it has an escape-restricted component. This definition implies, that readonly types are always escape-restricted. The opposite is not true, there are escape-restricted types which are not readonly. An example is the type

```
#{fld1: U8, fld2: {f1: U16}! }
```

It is not readonly since the bang operator is not applied to it. However, it has the field `fld2` with a readonly type, therefore it is escape-restricted.

We call a type which is not escape-restricted an "escapable" type.

A linear type always is a boxed record or abstract type or it contains a component of such a type. When the bang operator is applied to the linear type, it will recursively be applied to that component, turning it into a component of readonly type. Therefore, the result of applying the bang operator to a linear type will always be an escape-restricted type which is not linear.

There are even types which are linear and escape-restricted, such as the boxed record type

```
{fld1: U8, fld2: {f1: U16}! }
```

or the unboxed record with a field of linear type and a field of readonly type:

```
#{fld1: {f1: U16}, fld2: {f1: U16}! }
```

If all escape-restricted fields are taken from a record, the resulting partial record type is escapable. An example is the type

```
{fld1: U8, fld2: {f1: U16}! } take (fld2)
```

As the other restricted types, escape-restricted types impose additional restrictions on the use of their values: they may not "escape" from certain context. Again, the meaning of this is explained in Section 3.3.

Together we have the following properties for type expressions: A type expression can be regular or restricted. If it is restricted it can be linear, escape-restricted, or both. A readonly type is always escape-restricted but never linear.

Chapter 3

Working with Values

The main part of a Cogent program is usually about specifying values, typically the result values of functions, depending on argument values.

3.1 Patterns

Functional programming languages typically use the concept of *pattern matching*, which covers several concepts from imperative or object oriented languages, such as binding values to variables, accessing components of a value, or testing for alternatives. In Cogent patterns are the most important language construct for working with composite values.

A pattern is a syntactical language construct which can be *matched* against values. A pattern may contain *variables*, then matching it with a value has the effect of *binding* the contained variables to components of the matched value. In Cogent, as in languages like Haskell or Scala, a variable may occur atmost once in a pattern. Hence it is not possible to construct patterns which restrict matching values to have some parts which are equal to each other.

A pattern *conforms* to a type, if it matches at least one value of the type. A pattern can conform to several different types. A pattern is *irrefutable*, if it matches all values of all its conforming types. Irrefutable patterns cannot be used to discriminate between different sets of values, they can only be used to bind contained variables. If a pattern is matched against a value, the match must always be exhaustive, i.e. alternative patterns must be specified which together cover the value's type.

The conforming types of a pattern can always be inferred from the syntactical structure of the pattern. Therefore type expressions are not needed as part of patterns.

Syntactically, patterns may be put in parentheses for grouping:

Pattern:
    (*Pattern*)
    . . .

A pattern in parentheses is equivalent to the pattern itself.

3.1.1 Simple Patterns

The simplest patterns consist of only one part. They may be irrefutable or not.

Irrefutable Simple Patterns

A simple pattern can be a single variable or the special symbol _, which is called the *wildcard* pattern:

```
Pattern:
    ( Pattern )
    IrrefutablePattern
    ...

IrrefutablePattern:
    Variable
    WildcardPattern
    ...

Variable:
    LowercaseId

WildcardPattern:
    _
```

Both patterns conform to all types and are irrefutable, hence they match every possible value which may occur in Cogent. If a variable `x` is matched with a value, the value is bound to `x`. This means that in a certain *scope*, the value can be referenced by denoting the variable `x`.

The *WildcardPattern* _ contains no variable, therefore no variable can be bound when the pattern is matched with a value. The *WildcardPattern* is used when, for some reason, a value must be matched with a pattern but need not be referenced afterwards.

As for type and data constructors, the syntactically equal *Variable*s, *FieldName*s, and *TypeVariables* constitute three different namespaces. The same lowercase identifier can be used to denote a term variable, a type variable, and a record field in the same context without imposing any relation among them.

Refutable Simple Patterns

Refutable simple patterns consist of a single literal:

```
Pattern:
    ( Pattern )
    IrrefutablePattern
    LiteralPattern
    ...

LiteralPattern:
    BooleanLiteral
    IntegerLiteral
    CharacterLiteral
```

A *LiteralPattern* matches exactly one value, the value which is denoted by the literal. A *BooleanLiteral* conforms only to type `Bool`, a *CharacterLiteral* conforms only to type `U8`.

An *IntegerLiteral* conforms to every bitstring type which includes the value denoted by the literal. For example, the literal 100000 conforms to types `U32` and `U64` but not to `U16` or `U8`.

Since a *LiteralPattern* contains no variables, no variable can be bound when it is matched with a value. *LiteralPattern*s are used for discriminating the value from other values, not for binding variables.

Note that you cannot use a value bound to a variable like a literal in a pattern to match just that value. If a variable occurs in a pattern it is always used for a new binding which shadows any value already bound to it. In particular, this applies to variables bound in a topevel value definition (see Section 4.2.1).

3.1.2 Composite Patterns

Composite patterns conform to composite types. However, there are no patterns which conform to function types.

Tuple Patterns

A tuple pattern is syntactically denoted by a tuple of patterns:

IrrefutablePattern:
  Variable
  WildcardPattern
  TuplePattern
  . . .

TuplePattern:
  `()`
  `(` *IrrefutablePattern* `,` *IrrefutablePattern* { `,` *IrrefutablePattern*} `)`

The subpatterns in a tuple pattern must all be irrefutable. As a consequence, tuple patterns are also irrefutable. Even the tupel pattern `()` is irrefutable, although it matches only a single value. Since it conforms only to the unit type which has only this single value, it satisfies the requirements for an irrefutable pattern.

Note that, as for tuple types, there is no tuple pattern with only one subpattern, the corresponding syntactical construct like `(v)` is a pattern in parentheses and conforms to all types the inner pattern conforms to, not only to tuple types.

A tuple pattern `(`p_1`, ... ,` p_n`)` with $n \neq 1$ conforms to every tuple type with n fields where each subpattern p_i conforms to the type of the ith field.

If a tuple pattern is matched with a value, the subpatterns are matched with the corresponding fields of the value.

A useful case is a tuple pattern where all subpatterns are (distinct) variables. Such a pattern can be used to bind all fields of a tuple value to variables for subsequent access.

Here are some examples for tuple patterns:

```
(v1, v2, v3)
(v1, (v21, v22), _)
()
```

The first pattern conforms to all tupel types with three fields. The second pattern conforms to all tuple types with three fields where the second field has a tuple type with two fields. The third pattern only conforms to the unit type.

Record Patterns

Patterns for record values exist in two syntactical variants, depending on whether the record is boxed or unboxed:

IrrefutablePattern:
  Variable
  WildcardPattern
  TuplePattern
  RecordPattern

RecordPattern:
  Variable `{` *RecordMatchings* `}`
  `# {` *RecordMatchings* `}`

The main part *RecordMatchings* of a record pattern is used to match the fields and has the following syntax:

RecordMatchings:
  RecordMatching { `,` *RecordMatching}*

RecordMatching:
  FieldName [= IrrefutablePattern]

The basic case is a sequence of field names with associated subpatterns, such as in

```
fld1 = v1, fld2 = (v21, v22), fld3 = _
```

A record pattern with these *RecordMatchings* conforms to all record types which have atleast three fields named `fld1`, `fld2`, and `fld3`, and where `fld2` has a tuple type with two fields. More general, a record pattern where the *Record-Matchings* consist of pairs of field names and subpatterns conforms to all record types which have atleast the named (untaken) fields and every subpattern conforms to the corresponding field type. Since all subpatterns must be irrefutable, the record pattern is irrefutable as well.

A special application of a record pattern is to bind field values to local variables which have the same name as the field itself. The effect is to make the fields of a record value locally accessible using their field names. This can be accomplished for a specific field by matching a record pattern with a *RecordMatching* of the form `fldi = fldi`. Such a *RecordMatching* can be abbreviated by simply specifying the field name alone: `fldi`, for example in the *RecordMatchings*

```
fld1, fld2 = (v21, v22), fld3, fld4
```

Note that since the field name as a variable conforms to all types, the corresponding record patterns conform to all record types which have a (untaken) field named `fldi`, irrespective of the field type.

A record pattern starting with `#` conforms only to unboxed record types. When matched with a value, for every field according to the value's type a subpattern must be present in the *RecordMatchings* and is matched to the corresponding field value.

A record pattern starting with a *Variable* conforms to boxed and unboxed record types. When matched with a value this variable is bound to the remaining record after matching the subpatterns in the *RecordMatchings*. This "remaining" record has as its type the type of the matched value with all fields taken which are matched in the *RecordMatchings*. Matching a pattern of this kind with a value is called a "take operation".

The rationale for this is that boxed record types are linear and their values must be used exactly once. Matching only some fields would only use these fields and not the rest, which is not allowed. Hence the remaining record must also be matched so that it can be used as well. Even when all linear fields are matched the remaining record itself is still linear and must be preserved.

If value `val` has type

```
{fld1: U8, fld2: U16, fld3: U32}
```

an example take operation would be to match the pattern

```
v {fld1 = v1, fld3 = v3}
```

with `val`. This will bind `v1` to the value of the first field, `v3` to the value of the third field, and `v` to the remaining record of type

```
{fld1: U8, fld2: U16, fld3: U32} take (fld1,fld3)
```

where only the second field is still present.

Although the ordering of fields is relevant in a record type expressionm, it is irrelevant in a record pattern. Therefore the record pattern `#{fld1 = v1, fld2 = v2}` conforms to the types

```
#{fld1: U8, fld2: U16}
#{fld2: U32, fld1: U32}
```

and all other unboxed record types which have two fields named `fld1` and `fld2`.

When a field of non-linear type is taken from a (boxed or unboxed) record value, a copy of it could remain in the record and could be taken again. Cogent does not allow this, non-linear fields can also be taken only once. This way it is possible to represent uninitialized fields in a record by specifying the record type with the corresponding fields being taken.

Variant Patterns

A variant pattern consists of a data constructor and a subpattern for every value in the corresponding variant:

Pattern:
  (*Pattern*)
  IrrefutablePattern
  LiteralPattern
  VariantPattern

VariantPattern:
  DataConstructor {IrrefutablePattern}

A variant pattern conforms to every variant type which has atleast a variant with the *DataConstructor* as its tag. Although a variant pattern matches all values of the type having only that variant, this is not true for all other conforming types. For those types the pattern only matches the subset of value sequences which have been constructed with the *DataConstructor* as its discriminating tag. Therefore variant patterns are always refutable. As usual, when matched with a value, the match must be exhaustive, specifying a pattern for every variant.

When a variant pattern is (successfully) matched with a value, the subpatterns are matched with the remaining values after removing the discriminating tag.

The following is an exmple for a variant pattern:

```
TwoDim x, y
```

It conforms, e.g., to the variant type

```
<TwoDim U32 U32 | ThreeDim U32 U32 U32>
```

and generally to every type with a variant tagged with `TwoDim` and having two values. When it is matched with a value tagged with `TwoDim` the first value is bound to `x` and the second value is bound to `y`.

3.2 Expressions

As usual in programming languages, an *expression* denotes a way how to calculate a value. The actual calculation of a value according to an expression is called an *evaluation* of the expression. Since an expression may contain variables which are not bound in the expression itself ("free variables"), the value obtained by evaluating an expression may depend on the context in which the free variables are bound.

Usually, when an expression occurs in a Cogent program, a type may be *inferred* for it. There are several ways to infer an expression's type. The most basic way is to infer its type from its syntactical structure, although there are cases where that is not possible. If an expression has an inferred type, the value resulting from evaluating the expression always belongs to this type.

The general syntactical levels of expressions are as follows:

Expression:
    BasicExpression
    ...

BasicExpression:
    BasExpr
    ...

BasExpr:
    Term
    ...

Term:
    `(` *Expression* `)`
    ...

Every *Expression* can be used wherever a *Term* is allowed by putting it in parentheses.

3.2.1 Terms

The simplest expressions are called *terms*. A term specifies a value directly or, for a composite value, by specifying its parts.

A term can be a single variable, denoting the value which has been bound to the variable in the context.

Term:
    `(` *Expression* `)`
    Variable
    ...

From the variable alone no type can be inferred. However, a type may be inferred when the variable is bound. Then this type is also inferred for every occurence of the variable as a term in its scope.

Literal Terms

Terms for values of primitive types are simply the literals:

Term:
    `(` *Expression* `)`
    Variable
    LiteralTerm
    ...

LiteralTerm:
    BooleanLiteral
    IntegerLiteral
    CharacterLiteral
    StringLiteral

The inferred type for a *BooleanLiteral*, a *CharacterLiteral*, or a *StringLiteral* is `Bool`, `U8`, or `String`, respectively. The inferred type for a *IntegerLiteral* is the smallest bitstring type covering the value, thus the literal `200` has inferred type `U8`, whereas the literal `300` has inferred type `U16` and `100000` has inferred type `U32`.

Terms for Tuple Values

Terms for tuple values are written as in most other programming languages supporting tuples:

Term:
    `(` *Expression* `)`
    Variable
    LiteralTerm
    TupleTerm
    ...

TupleTerm:
    `()`
    `(` *Expression* `,` *Expression* { `,` *Expression*} `)`

Again, as for tuple types and patterns, a single *Expression* in parentheses is not a tuple term but is only syntactically grouped.

An example tuple term is

```
(15, 'x', 42, ("hello", 1024))
```

which specifies 4 subexpressions for the fields, separated by commas.

The type inferred from the structure of a tuple term is the tuple type with the same number of fields as are present in the term, where the field types are the types inferred for the subexpressions. If one of the subexpressions does not have an inferred type then no type can be inferred from the tuple term's structure.

Terms for Record Values

Cogent only suppoprts terms for unboxed record values. Boxed record values cannot be specified directly, they must always be created externally in a C program part and passed to Cogent as (part of) a function argument or result.

The syntax for terms for unboxed values specifies all field values together with the field names:

Term:
    `(` *Expression* `)`
    Variable
    LiteralTerm
    TupleTerm
    RecordTerm
    . . .

RecordTerm:
    `#` `{` *RecordAssignments* `}`

RecordAssignments:
    RecordAssignment { `,` *RecordAssignment*}

RecordAssignment:
    FieldName [`=` *Expression*]

An example is the record term

```
#{fld1 = 15, fld2 = 'x', fld3 = 42, fld4 = ("hello", 1024)}
```

which specifies 4 subexpressions for the fields, separated by commas. The field names must be pairwise different. As for record types, but other than for record patterns, the order of the field specifications is significant. Hence the term

```
#{fld2 = 'x', fld3 = 42, fld1 = 15, fld4 = ("hello", 1024)}
```

evaluates to a different value than the first example term.

The type inferred from a record type's structure is the unboxed record type with the same number of fields in the same order as are present in the expression, named according to the names given in the term. The field types are the types inferred for the subexpressions. If a subexpression has no inferred type, no type can be inferred from the record term's structure.

Terms for Values of Variant Types

A term for a value of a variant type specifies the discriminating tag and the actual values:

Term:
  (*Expression*)
  Variable
  LiteralTerm
  TupleTerm
  RecordTerm
  VariantTerm
  ...

VariantTerm:
  DataConstructor {Term}

Examples for such terms are

```
Small 42
TwoDim 3 15
```

For a *VariantTerm* it is not possible to infer a type from its structure, since there may be several variant types using the same *DataConstructor*. The Cogent compiler even does not infer the type if there is only one variant type using the *DataConstructor* as tag.

Terms for Values of Function Types

A term for a value of a function type is, as usual, called a *lambda expression.* Often in other programming languages, a lambda expression consists of a body expression and a variable for every argument. In Cogent all functions take only one argument, therefore only one variable is needed. However, more general than a variable, an irrefutable pattern may be used. Every application of such a function is evaluated by first matching the pattern against the argument value, thus binding all variables contained in the pattern. Then the body expression is evaluated in the context of the bound variables to yield the result.

The syntax for lambda expressions is:

Term:
  (*Expression*)
  Variable
  LiteralTerm
  TupleTerm
  RecordTerm
  VariantTerm
  LambdaTerm
  ...

LambdaTerm:
    \ *IrrefutablePattern [*: *MonoType]* => *Expression*

Optionally, the argument type may be specified explicitly after the pattern. If no unique conforming type can be inferred for the pattern, the argument type is mandatory.

Examples for lambda terms are

```
\x => (x,x)
\(x,y,z) (U8, U8, Bool) => #{fld1 = y, fld2 = (x,z)}
\(x,y) : (U32,U32) => TwoDim y x
```

In the first case the argument type must be known from the context by knowing an inferred type for the lambda term, for example the type `U8 -> (U8,U8)`. In the third case the result type must be known from the context by knowing an inferred type for the lambda term, for example the type

```
(U32,U32) -> <TwoDim U32 U32 | Error U8>
```

The body expression in a lambda term is restricted to not contain any free non-global variables. Non-global variables are variables bound by pattern matching in contrast to *global* variables which are bound by a toplevel definition (see Section 4.2).

If the body expression of a lambda term has inferred type T2 and the argument type is explicitly specified as T1 then the type inferred from the structure of the *LambdaTerm* is T1 -> T2.

3.2.2 Basic Expressions

Basic expressions are constructed from terms in several ways, which all correspond semantically to a function application.

Plain Function Application

As is typical for functional programming languages, a value in Cogent can be a function and it can be applied to arguments.

As we have seen with function types, in Cogent all functions have only one argument. Hence, an expression for a function application consists of a term for the function and a second term for the argument:

BasExpr:
    Term
    FunctionApplication
    ...

FunctionApplication:
    BasExpr BasExpr

The argument Expression is simply put after the Expression for the function. This is common in functional programming languages, whereas in imperative and object oriented languages (and in mathematics) the argument is usually put in parantheses like in $f(x)$. In Cogent this is allowed, since a *BasExpr* may be an expression in parentheses, but it is not necessary.

The syntax here is ambiguous. Several *BasExpr* in a row are interpreted as left associative. Therefore the following two *BasExpr* are equivalent:

```
f 42 17 4
((f 42) 17) 4
```

If the first *BasExpr* in a *FunctionApplication* has an inferred type it must be a function type T1 -> T2. If the second *BasExpr* has an inferred type it must be equal to T1. The type inferred from the *FunctionApplication*'s structure is type T2.

As an example, if the variable `f` is bound to a function of type `U8 -> U16` then the basic expression

```
f 42
```

is a *FunctionApplication* with a result of type `U16`.

Operator Application

In Cogent there is a fixed set of predefined functions. These functions are denoted by *operator symbols* which are syntactically different from variables. In contrast to normal functions, predefined functions may be binary, i.e. take two arguments. Binary operator applications are written in infix notation:

BasExpr:
    Term
    FunctionApplication
    OperatorApplication
    ...

OperatorApplication:
    UnaryOP BasExpr
    BasExpr BinaryOp BasExpr

UnaryOp: one of
    `upcast complement not`

BinaryOp: one of
    `o * / % + - >= > == /= < <= .&.  .^.  .|.  >> << && || $`

As usual in most programming languages, the syntax here is ambiguous and operator precedence rules are used for disambiguation. The precedence levels ordered from stronger to weaker binding are:

```
upcast complement not <plain function application>
o
* / %
+ -
< > <= >= == /=
.&.
.^.
.|.
<< >>
&&
||
$
```

Note that plain function application is treated like a binary invisible operator, where the first argument is the applied function and the second argument is the argument to which the function is applied.

When binary operators on the same level are combined they are usually left associative, with the exception of `o`, `&&`, `||` and `$` which are right associative and `<, >, <=, >=, ==, /=` which cannot be combined.

TODO: describe all operation semantics and inferred types

Put Expressions

A common function used in functional programming languages is the record update function. It takes a record value and returns a new record value where one or more field values differ. In Cogent the application of this function is restricted: if a field has a linear type, it cannot be replaced, since then its old value would be discarded without being used. In this case the field can only be replaced, when it has been taken in the old value. For this reason the record update function is called the "put function" in Cogent. For non-linear fields the put function may either put a value into a taken field or replace the value of an untaken field.

Cogent supports a *PutExpression* as specific syntax for applying the put function. It specifies the old record value and a sequence of new field values together with the corresponding field names:

BasExpr:
  Term
  FunctionApplication
  OperatorApplication
  PutExpression
  ...

PutExpression:
  BasExpr { RecordAssignments }

As an operator the *RecordAssignments* have the same precedence as plain function application and the unary operators.

If a type T is inferred for the leading *BasExpr* in a *PutExpression*, T must satisfy the following conditions: it must be a (boxed or unboxed) record type having all fields occuring in the *RecordAssignments.* If such a field has a linear type it must be taken in T. The type inferred from the structure of the *PutExpression* then is

  T `put` (fld1,. . . ,fldn)

where fld1,. . . ,fldn are all fields occurring in the *RecordAssignments.*

Unlike in a record term, the field order in a *PutExpression* is not significant.

If the variable `r` is bound to a value of type `R` where

```
typedef A
typedef R = {fld1: A, fld2: U32, fld3: (Bool,U8), fld4: A}
            take (fld3,fld4)
```

and variable `a` is bound to a value of type `A`, then the following are valid put expressions:

```
r {fld2 = 55, fld3 = (True, 17)}
r {fld4 = a, fld2 = 10000}
```

The first expression has inferred type `R put (fld2,fld3)` which is equal to the type

```
{fld1: A, fld2: U32, fld3: (Bool,U8), fld4: A} take (fld4)
```

The expression `r {fld1 = a}` is invalid since field `fld1` is untaken and has linear type.

Member Access

A second function commonly provided for records is *member access* or projection, often denoted by a separating dot in programming languages. Cogent provides the same syntax for member access:

BasExpr:
  Term
  FunctionApplication
  OperatorApplication
  PutExpression
  MemberAccess

MemberAccess:
  BasExpr `.` *FieldName*

Here, the *BasExpr* specifies the record value and the *FieldName* specifies the name of the field to be accessed. As an operator, the dot in a *MemberAccess* has the highest precedence, higher than the unary operators.

Again, in Cogent the use of member access is restricted. The type inferred for the leading *BasExpr* in a *MemberAccess* must be either an unboxed record type or a readonly boxed record type. Then it is possible to use the value of only one field without caring about the other fields. Moreover, also the type of the accessed field must be non-linear, since in addition to being accessed, its value also remains in the record, hence it could be used twice.

The type inferred from the *MemberAccess* expression structure is the type of the field named by the *FieldName*.

If types `A` and `R` are defined as in Section 3.2.2 and `r` is bound to a value of type `R!` then the basic expression `r.fld2` is a valid *MemberAccess*. The basic expression `r.fld3` is invalid since field `fld3` is taken in `R!`, the basic expression `r.fld1` is valid since field `fld1` has type `A!` in `R!` (due to recursive application of the bang operator). If `r` is bound to a value of type `R` then also the basic expression `r.fld2` is invalid since type `R` is linear.

3.2.3 General Expressions

In Cogent the most general concept for specifying a calculation as an expression is *matching*. All other forms of general expressions can be understood as specific variants of matching.

Matching Expressions

A *MatchingExpression* matches a value against one (irrefutable) pattern or several (refutable) patterns. For every pattern a subexpression is specified for the result:

> *Expression:*
>
> *BasicExpression*
> *MatchingExpression*
> ...
>
> *MatchingExpression:*
>
> *ObservableBasicExpression Alternative {Alternative}*
>
> *ObservableBasicExpression:*
>
> *BasicExpression*
> ...
>
> *Alternative:*
>
> `|` *Pattern PArr Expression*
>
> *PArr: one of*
>
> `-> => ~>`

All *Expressions* in the *Alternatives* must have equal inferred types, this is also the type inferred from the *MatchingExpression*'s structure.

For every *Alternative* the *Expression* is called the *scope* of the variables occurring in the *Pattern*.

All *Patterns* in the *Alternatives* must conform to the type T inferred for the leading expression. The *Patterns* together must be exhaustive for T, that means, every value of type T must match one of them. This may be accomplished by using an exhaustive set of refutable patterns, such as one for every variant in a variant type, or by optionally specifying some refutable patterns followed by a final alternative with an irrefutable pattern.

The order in which alternatives are specified is irrelevant. The pattern syntax in Cogent guarantees that different refutable patterns cannot partially overlap, i.e. the sets of matching values are disjunct or equal. Moreover, a refutable pattern may be specified in at most one alternative. Together, every value matches at most one of the refutable patterns, there is no need to resolve conflicts. An irrefutable pattern is only used when no refutable pattern matches.

If the variable `x` is bound to a value of type `U8` an example for a *Matching-Expression* is

```
x + 7 | 20 -> "too much"
      | 10 -> "too few"
      | _  -> "unknown"
```

It has the inferred type `String`.

If the variable `v` is bound to a value of the variant type

```
< TwoDim U32 U32 | ThreeDim U32 U32 U32 | Error U8 >
```

then the following is a valid *MatchingExpression* with inferred type `U32`:

```
v | TwoDim x y -> x+y
  | ThreeDim x y z -> x+y+z
  | Error code -> 0
```

whereas

```
v | TwoDim x y -> x+y
  | ThreeDim x y z -> x+y+z
```

is invalid since it is not exhaustive for the type of `v`.

TODO: Using layout for Alternative grouping

Alternatively to the separator `->` the separators `=>` and `~>` can be used in an *Alternative*. Semantically they have the same meaning, however they may allow for some code optimization when the first is used for "likely" alternatives and the second for "unlikely" alternatives.

Binding Variables

If the only intention for using a *MatchingExpression* is binding variables, the simpler *LetExpression* syntax can be used:

Expression:
  BasicExpression
  MatchingExpression
  LetExpression
  ...

LetExpression:
  `let` *Binding {*`and` *Binding}* `in` *Expression*

Binding:
  IrrefutablePattern [: *MonoType]* = *ObservableExpression*

ObservableExpression:
  Expression
  ...

A simple *LetExpression* is equivalent to a *MatchingExpression* with one *Alternative*:
  `let` IP = E `in` F
is semantically equivalent with
  E | IP -> F

From this it follows that pattern IP must conform to the type inferred for E and the type inferred from the *LetExpression*'s structure is that inferred for F. The expression F is also called the "body" of the *LetExpression*, it is the scope of the variables in IP.

The *LetExpression*

```
let x = y + 5 in (True, x)
```

binds the variable `x` to the result of evaluating the expression `y + 5` and evaluates to a tuple where the bound value is used as the second field value. The tuple expression is the scope of variable `x`.

If types `A` and `R` are defined as in Section 3.2.2 and `r` is bound to a value of type `R` then the *LetExpression*

```
let s {fld1 = x, fld2} = r in (x, fld2 + 5, s)
```

binds the variables `s`, `x`, and `fld2` by matching the pattern against the value bound to `r` as described in Section 3.1.2. Then it uses them in their scope which is a tuple term. The type inferred for the *LetExpression* is

```
(A, U32, R take (fld1, fld2))
```

In a *Binding* optionally a *MonoType* may be specified:
    IP `:` T `=` E
If neither for E nor the pattern IP a type can be inferred the type specification is mandatory.

If E is an *IntegerLiteral* of type U and T is a bitstring type which is a superset of U then the value of E is automatically widened to type T before matching it against IP. Therefore the *LetExpression*

```
let x: U32 = 5 in (True, x)
```

has inferred type `(Bool, U32)`, although the literal `5` has type `U8`.

A *LetExpression* of the form
    `let` B1 `and` B2 `in` F
is simply an abbreviation for the nested *LetExpression*
    `let` B1 `in` `let` B2 `in` F

A *LetExpression* which uses the wildcard pattern
    `let _ =` E `in` F
can be abbreviated to
    E `;` F
using the following syntax:

BasicExpression:
    BasExpr
    BasExpr `;` *Expression*

Since a *LetExpression* is only used to bind variables occurring in the pattern and there is no variable in the wildcard pattern this case seems to be useless. Its only use is when expression E has side effects. Note that functions which are completely defined in Cogent do not have side effects, however, functions defined externally can have side effects.

An example usage whould be an externally defined function of type `String -> ()` which is bound to the variable `print` and prints its *String* argument to a display. Then the expression

```
v | TwoDim x y -> print "flat"; x+y
  | ThreeDim x y z -> print "space"; x+y+z
  | Error code -> print "crash"; 0
```

would print one of the strings to the display whenever it is evaluated.

Conditional Expressions

If the only intention for using a *MatchingExpression* is discrimination between two cases the *ConditionalExpression* can be used which is nearly omnipresent in programming languages. It has the usual syntax:

Expression:
  BasicExpression
  MatchingExpression
  LetExpression
  ConditionalExpression

ConditionalExpression:
  `if` *ObservableExpression* `then` *Expression* `else` *Expression*

The *ConditionalExpression*
  `if` C `then` E `else` F
is equivalent to the *MatchingExpression*

```
C | True -> E
  | False -> F
```

From this it follows that C must have the inferred type `Bool` and E and F must have the same inferred type which is the type inferred from the *ConditionalExpression*'s structure.

If a *MatchingExpression* discriminates among more than two cases, as usual a nested *ConditionalExpression* can be used instead.

TODO: Using layout to disambiguate nested ConditionalExpressions

An example for a *ConditionalExpression* is

```
if x > 5 then (True, "sufficient") else (False, "insufficient)
```

It has the inferred type `(Bool, String)`.

Observing Variables

At some places variables can be "observed" in an expression. Observing a variable means replacing its bound value with a copy of readonly type. Observing variables is the only way how values of readonly types can be produced in Cogent.

When a variable should be observed, an expression must be specified as scope of the observation. The readonly value may be freely used in this scope, but it may not escape from it. Syntactically, an expression which may be the scope of a variable observation is called an *observable expression.* The syntax for variable observation is as follows:

ObservableBasicExpression:
  BasicExpression
  BasicExpression {`!` *Variable}*

ObservableExpression:
  Expression
  Expression {`!` *Variable}*

In both cases one or more observed variables are specified at the *end* of the observation scope using the "bang" operator as a prefix. Examples for *ObservableExpression*s are

```
if isok #{fld1=x, fld2=x, fld3=z} then 5 else 0 !x !y
let v1 = x and v2 = x and v3 = z in (1, 2, 3) !x !z
```

If there is at least one banged variable in an observable expression, then the inferred type of the scope may not be an escape-restricted type.

The *ObservableExpression*

```
E !V
```

is conceptually equivalent to a *LetExpression* of the form

```
let V = readonly V in E
```

where `readonly` would be an operator which produces a readonly copy from a value. An important effect of this form is that the variable used for the readonly copy has the same name as the variable containing the original value. Therefore the former variable shadows the latter in its scope, making the original value unaccessible there.

The operator `readonly` does not actually exist in Cogent, hence expressions of the second form cannot be used to bind readonly copies. This guarantees that the variable for the readonly copy *always* shadows the original value in its scope.

Observable expressions may only occur in three places: As the leading expression in a *MatchingExpression* and in the corresponding position in the more specific forms, which is the right-hand side of a *Binding* in a *LetExpression* and the condition in a *ConditionalExpression*.

3.3 Expression Usage Rules

Cogent's linear type system implies additional restrictions on expression usage over the usual restriction that the type of a function argument must be compatible to the parameter type. The additional rules are described in this section.

3.3.1 Using Values of Linear Types

The basic rule for linear types is that their values must be used exactly once. For observing this rule it must be specified in more detail, what it means to use a value.

Sharing a Value

In a Cogent program, values are always denoted by expressions. If an expression is a *Term* for a tuple, a record, or a variant type, or if it is a *BasExpr* representing the application of a function or operator, or if it is a *MatchingExpression* or one of its specific variants, the value is created by evaluating the expression. Then it can only be used atmost once: at the position where the expression syntactically occurs in the program. In the remaining cases the expression is either a single variable or a *MemberAccess* (values of literals are never linear). A value bound to a variable can be used more than once: it is used at all places where it is referenced by the variable name in its scope. The value of a record field can be used more than once by accessing the field several times. In both cases we say the value may be "shared".

When a record field is accessed its value is not taken from the record, hence it is already shared between the record and the access result upon a single member access. As a consequence, record fields of linear type may not be accessed using a *MemberAccess* expression.

Hence the rule for using values of linear types not more than once is only relevant for variables: if a variable has a bound value of a linear type, the value must be used atmost once by referencing it, it may not be shared. However, as can be seen for the variable `v` in the example

```
if x == 5 then f v else g v
```

the number of uses of the value is not simply the number of occurrences of the variable name in its scope. Instead, the rule is that a variable of linear type must occur atmost once in all possible paths of an execution. Thus, for a *ConditionalExpression* it must either occur once in the condition, or in each branch. For a *MatchingExpression* it must either occur once in the leading *ObservableBasicExpression*, or in each *Alternative*.

Note that the field names in a *RecordTerm*, a *PutExpression*, a *RecordPattern* or a *MemberAccess* are irrelevant, even if a field is present with the same name as the variable. Moreover, only free occurrences count. If a variable of the same name is bound in the scope, the binding and its usages are irrelevant for the original variable. Variables are bound by *LetExpression*s, *ObservableExpression*s, *ObservableBasicExpression*s, and *LambdaTerm*s.

Discarding a Value

If a variable is never used in its scope its value is "discarded". Values of linear type may not be discarded. This is guaranteed for values bound to a variable, if it is used in every possible path of execution.

Although the value of an expression other than a variable or member access cannot be used more than once, it can be discarded by matching the expression with a pattern other than a variable or a boxed record pattern. In the case of the wildcard pattern as in

```
let _ = someExpression
```

the expression `someExpression` may have a linear type, then this matching would be illegal. In the case of a *LiteralPattern* the expression must always have a primitive type which is never linear. The same holds for an expression which occurs as condition in a *ConditionalExpression*.

In the case of a *TuplePattern*, a *VariantPattern* or an unboxed *RecordPattern* the expression only has a linear type if it has components of a linear type. Then it is no problem to discard the value as long as no component of a linear type is discarded, as in

```
let (a, #{fld1= _, fld2=b}, c) = someExpression
```

In this case the `fld1` of the second field of the value is discarded which would be illegal if it has linear type.

A record field is also discarded if it is replaced in a *PutExpression*. Therefore in a *PutExpression* the leading *BasExpr* must not have linear fields which are put, if there are linear fields they must have been taken.

The value of an expression is discarded when the expression is used as the *BasExpr* in a *MemberAccess*.

Together, linear values could be discarded by binding them to a variable which is never used in its scope, by matching them with the wildcard pattern, by replacing them in a *PutExpression*, or by using them as the record in a *MemberAccess*. All these cases are not allowed for values of linear type in Cogent.

However, there are two other cases which specifically apply to values of a boxed record type. If such an expression is used as the leading expression in a *PutExpression* or if it is matched against a *RecordPattern*, it is discarded as well. These two cases are allowed in Cogent. Note that in both cases a new value of the same type is created, in the first case it becomes the result of the *PutExpression*, in the second case it is bound to the leading variable of the *RecordPattern*.

The Result of Using a Value

What happens to a value after it has been used? "Using" here only means a *syntactical* usage, it does not mean that the value is dismissed afterwards. Depending on the context of usage there are three possibilities: the value may immediately be used in the context, it may become a part of another value (its "container" value), or it may be bound to a variable.

If the value results from evaluating an expression E in an *Alternative*, in a branch of a *ConditionalExpression*, or in the body of a *LetExpression*, then the value becomes the evaluation result of the expression containing E and is immediately used in the context.

If expression E occurs as subexpression in a tuple term, a record term, or a variant term, or in a *RecordAssignment* of a *PutExpression*, its evaluation result becomes a part of its container value created by the term or *PutExpression*, respectively. Since a value of linear type may be used only once, it is always the part of atmost one container value. The container value, since it has a part of linear type is also of linear type and behaves in the same way.

Whenever a container value is used, it is used with all its parts. A linear part can be separated from its container by matching the container value with a complex pattern which binds the part to a variable and dismisses the container. If the container is a boxed record, a new container will be created where the part is taken. Thus, after binding the part to a variable it is not a part of its container anymore.

If expression E is the leading expression in a *MatchingExpression*, or occurs in a *Binding* of a *LetExpression*, or is the argument in a *FunctionApplication*, then it is matched against a pattern. If the pattern is a variable, the evaluation result is bound to the variable. It remains bound to it until the evaluation of its scope ends. However, if the value is of linear type, it cannot be referenced by the variable after its first use, hence thereafter the binding is irrelevant.

Note that the body expression in a *LambdaTerm* is not evaluated when evaluating the *LambdaTerm* to yield a function. The body will only be evaluated when the function is applied to an argument.

Taking it all together, the usage rules imply that a linear value in a pure Cogent program is always either bound to exactly one variable which has not yet been used or it is a part of exactly one container value which also is linear. In a Cogent program linear values are only dismissed and created in *PutExpression*s and by matching boxed *RecordPatterns*. In both cases a boxed record value is dismissed and a value of the same type is created.

These properties are exploited by Cogent in the following way. Whenever a boxed record is dismissed it is “reused” to create the new value. Since the new value only differs from the old value by some fields having a different value, the old value is *modified* by replacing these field values. As a consequence, linear values are *never* created or destroyed in a Cogent program, they are only passed around as a single copy, possibly being modified on their way. Creating or destroying linear values must be accomplished externally implemented in C.

3.3.2 Using Values of Readonly Types

The basic rule for readonly types is that their values may not be modified. Of course, since Cogent is a functional language, values are conceptually never modified. However we have seen that value modification occurs in Cogent as an optimization for linear values, although semantically this modification can never be observed.

Modifying a Value

The only way to modify a value in Cogent is by changing the value of a field in a boxed record. This can be achieved with the help of a *PutExpression* where a new value is specified for a field. It can also be achieved with the help of a take operation by matching a *RecordPattern* with a boxed record value.

Therefore the following rules apply to values of readonly types:

- a value of readonly type may not be used as the leading *BasExpr* in a *PutExpression*,
- a value of readonly type may not be matched against a record pattern.

When taking a field from a readonly record it is irrelevant whether the field has linear type or not. In both cases the record would be modified which is not allowed. If the field has non-linear type, the taken value could remain in the record. However, Cogent implements taking fields always by removing the field value from the record, thus modifying the record.

Creating readonly Values

The only way to create a value of readonly type is to apply the bang operator to a variable in an *ObservableExpression* or *ObservableBasicExpression.* This creates a readonly copy of the bound value and binds it to the same variable, using the subexpression before the first banged variable as scope for this binding. We call this subexpression a banged scope. If the previosly bound value had the linear type T, the readonly copy has type T! which is readonly or contains readonly parts.

Note that the original binding is shadowed in the banged scope, hence the linear value cannot be referenced there, in particular, it cannot be modified. This is exploited by Cogent in the following way. The original value is actually not copied at all, it remains bound to its variable. Only its type as seen through the variable is changed to T! in the banged scope.

In the banged scope the readonly copies can be freely duplicated, bound to any number of variables and inserted as parts in any number of container values.

Preventing Values from Escaping

When execution leaves the banged context the shadowing ends and original value of linear type may be accessed again and may be modified. Although all copies are still of readonly type, they would be modified as well, since actually they have not been copied. This problem is solved by Cogent by preventing the copies to "escape" from the banged scope. Then they cannot be referenced and observed outside the scope and modifications to the original value are no problem.

If a readonly copy is bound to a variable, the scope of this binding must be syntactically enclosed in the banged scope and cannot be referenced outside. The only way a value can escape from the banged scope is if it is the result value the banged scope evaluates to or a part of it. This must be prevented by Cogent.

It seems that to achieve this Cogent has to "track" all readonly copies and prevent them to become a part of the result value. However, it is impossible to do this statically, since a copy can be passed to an externally defined function which may return it as part of its result without Cogent knowing this. Therefore a simpler but much more radical approach is used, by preventing *all* values with an escape-restricted type from escaping from *any* banged scope, irrespective whether it is related to the value or type of the banged variable. This safely also prevents the readonly copies from escaping.

This approach can be implemented with the help of type checking. The rule to apply is that the type inferred for a banged scope in an *ObservableExpression* or *ObservableBasicExpression* must not be escape-restricted.

This rule implies that even readonly values which existed outside of the banged scope cannot be used as part of its result. Normally this is not a problem since they are available outside the banged scope anyways. However, if the value's type is both escape-restricted and linear, the situation is different. Due to the linearity, the value must not be discarded in the banged scope, it must leave it, which is not allowed either. The solution here is to separate all escape-restricted parts from the rest, discard them in the banged scope and let the rest escape.

Chapter 4

Programs

A Cogent program is a sequence of toplevel definitions and include statements. There is no main program, it must always be implemented externally in C.

> *Program:*
>
> *TopLevelUnit {TopLevelUnit}*

4.1 Including Files

For modularization purpose a Cogent program may be distributed among several files using include statements. Like in many other programming languages, an include statement is replaced by the content of the included file.

The syntax for an include statement is:

> *TopLevelUnit:*
>
> *Include*
>
> ...
>
> *Include:*
>
> `include` *StringLiteral*
>
> `include` *SystemFile*
>
> *Systemfile: informal*
>
> A file pathname enclosed in `<` and `>`.

The *StringLiteral* specifies the pathname of the file to be included, either as an absolute path or as a path relative to the directory where the file containing the include statement resides. A *SystemFile*, like in C, specifies a file which is searched at standard places by the Cogent compiler.

Include statements are transitive, if an included file contains include statements they are executed as well.

However, every file is included only once. If several include statements specify the same file, it is only include by the first statement seen when processing the Cogent source file, all other inclusions of the file are ignored. This is also true for transitive includes, in particular, circular includes do no harm. The effect is the same that is usually achieved in C by #DEFINEing a flag in an include file and including the file body only if the flag is not yet set.

4.2 Toplevel Definitions

The only syntactical constructs which may occur as toplevel units in a Cogent source program are *definitions*.

> *TopLevelUnit:*
>   *Include*
>   *Definition*
>
> *Definition:*
>   *TypeDefinition*
>   ...

A definition may be a type definition, as described in Section 2.1.3.

4.2.1 Value Definitions

A definition may also be a *value definition*. It has the following syntax:

> *Definition:*
>   *TypeDefinition*
>   *ValueDefinition*
>   ...
>
> *ValueDefinition:*
>   *Signature Variable* = *Expression*
>
> *Signature:*
>   *Variable* : *PolyType*
>
> *PolyType:*
>   *MonoType*
>   ...

A value definition is conceptually mainly a syntactical variant for a *Let-Expression* which binds a single variable. However, there are the following differences:

- the variable bound by a definition is a *global* variable which can be referenced in *LambdaExpression*s (see Section 3.2.1),
- the scope of the variable consists of the whole Cogent program after *and before* the definition,
- the type specification is mandatory and in the case of a function type, instead of a *MonoType* it may be a more general *PolyType* (see Section 4.2.4).

A *ValueDefinition* of the form

    V : T V = E

is conceptually equivalent with

    let V : T = E in F

where F is the whole Cogent program around the definition. This equivalence is only conceptual, syntactically it is not correct, since F is not an *Expression* and cannot be expressed as one.

Note that the variable V has to be specified twice. It is an error if two different variables are used in a value definition.

Like type definitions, value definitions in Cogent are restricted to be not recursive: the variable V may not occur freely in the expression E and there may be no cyclic references between different value definitions.

The *Expression* E may only contain free occurrences of global variables which have been bound in other value definitions. The *Expression* E and in the *Poly-Type* T may contain all type synonyms which are defined in a type definition before or after the value definition.

An example for a value definition is

```
maxSize: U16
maxSize = 42
```

TODO: layout rules for value definitions

4.2.2 Function Definitions

A function definition is a special case of a value definition, where the value has a function type. This could be achieved with a normal value definition using a lambda expression to specify the value to be bound. However, for function definitions additional syntactical forms are supported in Cogent:

Definition:
  TypeDefinition
  ValueDefinition
  FunctionDefinition
  ...

FunctionDefinition:
  Signature Variable IrrefutablePattern = *Expression*
  Signature Variable Alternative {Alternative}

A *FunctionDefinition* of the form

```
V : T
V IP = E
```

is semantically equivalent with

```
V : T
V = \ IP => E
```

In a function definition the type T must of course be a function type.

An example for this kind of function definition is

```
f: (U32, U32) -> #{sum: U32, dif: U32}
f v = let (x,y) = v in #{sum=x+y, dif=x-y}
```

where the variable `v` is used to reference the function argument. Note that by using a pattern instead of a single variable, it is possible to directly access the argument components according to the argument type:

```
f: (U32, U32) -> #{sum: U32, dif: U32}
f (x,y) = #{sum=x+y, dif=x-y}
```

The second form of a function definition is intended for the case that the argument is not matched against a single irrefutable pattern but instead against several exhaustive refutable patterns. Then the *FunctionDefinition* of the form

> V : T
> V A1 . . . An

is semantically equivalent with

> V : T
> V `arg` = `arg` A1 . . . An

where `arg` is a new variable not occurring elsewhere.

Examples are the function definitions

```
f: <TwoDim U32 U32 | ThreeDim U32 U32 U32 | Error U8> -> (U32, U32)
f | TwoDim x y -> (y,x)
  | ThreeDim x y z -> (y,z)
  | Error _ -> (0,0)

g: U8 -> U8
g | 0 -> 'a'
  | 1 -> 'b'
  | 2 -> 'c'
  | _ -> 'd'
```

TODO: layout rules

4.2.3 Abstract Definitions

An *abstract* definition only specifies the type of a value bound to a variable but not the value itself. Abstract definitions are only allowed if the bound value has a function type. The syntax is a normal value definition reduced to its signature:

Definition:
 TypeDefinition
 ValueDefinition
 FunctionDefinition
 AbstractDefinition
 . . .

AbstractDefinition:
 Signature

The purpose of abstract definitions is to define functions which are implemented externally as C functions.

A collection of abstract definitions together with corresponding type definitions is often called an "abstract data type" ("ADT"). Typically an abstract data type consists of one or more abstract type definitions and abstract definitions for functions working with values of these types, where both types and functions are externally defined in C.

4.2.4 Polymorphic Definitions

Function values bound by toplevel definitions may be *polymorphic* which means that their type is not specified uniquely. This is achieved by allowing free type variables in the value's type as specified in the definition. A type expression which may contain free type variables is called a *PolyType* in Cogent. Syntactically *PolyType*s must be closed by binding the free type variables by an "all-quantification". The syntax is as follows:

> *PolyType:*
>   *MonoType*
>   `all` *PermSignatures* `.` *MonoType*
>
> *PermSignatures:*
>   *PermSignature*
>   `(` *PermSignature* { `,` *PermSignature*} `)`
>
> *PermSignature:*
>   *TypeVariable*
>   ...

Here all type variables which occur free in the *MonoType* must be listed in the *PermSignatures*. An example for a polymorphic value definition is

```
f: all (t, u). (t, u) -> (U32, u, U16, t)
f (x,y) = (200, y, 100, x)
```

Since the types `t` and `u` are unknown, no expressions can be specified for their values other than variables to which the values have been bound. As a consequence, polymorphic values are always polymorphic functions which take the values of the unknown types as (part of) their argument and only pass them around, perhaps placing them in the function result.

A typical example for a polymorphic function works with lists of arbitrary elements. Therefore no specific type shall be specified for the list elements, which is achieved by using a free type variable for it. The corresponding list type can be defined as a generic abstract type:

```
type List e
```

Then the usual functions working on lists can be defined by the following abstract polymorphic function definitions:

```
first: all e. List e -> Option e
rest: all e. List e -> List e
cons: all e. (e, List e) -> List e
```

Together these definitions constitute an abstract data type for lists. Note, that neither the list type nor the list functions can be defined in Cogent since they would require recursion.

Even when a value of an unknown type is only carried around, additional information about the type is needed for doing this correctly: If the type is linear, the value may still be used only once, whereas the value may be freely copied, if the type is non-linear. Therefore it is possible to specify "permissions" for a type variable in the *PermSignatures* using the following syntax:

PermSignature:
  TypeVariable
  TypeVariable :< *Permissions*

Permissions:
  Permission {Permission}

Permission: one of
  `D S E`

The permissions associated with a type variable specify what must be possible for values of that type. Permission `D` means the values can be *discarded*, permission `S` means the values can be *shared*, and permission `E` means that values may *escape* from a banged context. If a type variable has kind `DSE` the actual type must be regular. If a type variable has kind `DS` the actual type must not be linear, it may be regular or escape-restricted. If it has kind `E` the actual type must not be escape-restricted, it may be regular or linear.

If no *Permissions* are specified for a type variable the default permissions `E` apply.

In the example

```
f: all (t, u :< DSE) . (t, u) -> (U32, u, U16, t, u)
f (x,y) = (200, y, 100, x, y)
```

the type `t` has default permissions `E` and is thus required to be escapable. Type `u` is required to be regular and it is correct to use parameter `y` more than once in the body expression.

Whenever a global variable bound by a polymorphic value definition is referenced, actual types must be substituted for the free type variables. These types can be explicitly specified using the following syntax:

Term:
  `(` *Expression* `)`
  Variable
  LiteralTerm
  TupleTerm
  RecordTerm
  VariantTerm
  LambdaTerm
  PolyVariable

PolyVariable:
  Variable `[` *OptMonoType* { `,` *OptMonoType*} `]`

OptMonoType:
  MonoType
  `_`

If the types are not specified or if some types are specified by `_`, the compiler tries to infer them. If the compiler is unable to infer the types, then they must be explicitly specified. For example, if the compiler has difficulty with the last type argument, instead of `f`

$$U8, Char, < AU8 || BU16 >$$

, we can write `f`

$$\_, \_, < AU8 \| BU16 >$$

.

If `f` has been bound by the polymorphic definition above, example references are

```
f[{fld1: U8, fld2: U8},U32]
f[U16,{fld1: U8, fld2: U8}]
```

where the second reference is illegal since the second type variable `t` is substituted by type `{fld1:  U8, fld2:  U8}` which is not regular.

Chapter 5

Grammar

Here we use a grammar notation which is similar to that used in the Java language specifications. The meta constructs have the following meaning:

- The italic brackets *[]* make their content optional.
- The italic braces *{}* make their content repeatable (and optional).

Nonterminals are denoted in *italics*, literal code is denoted in `typewriter` font.

Productions are structured by indenting the right-hand side, every single line is one alternative. There are special forms of productions for selecting among a set of terminals and for specifying the syntax od a nonterminal informally.

Program:
  TopLevelUnit {TopLevelUnit}

TopLevelUnit:
  Include
  Definition

Include:
  `include` *StringLiteral*
  `include` *SystemFile*

Systemfile: informal
  A file pathname enclosed in `<` and `>`.

Definition:
  TypeDefinition
  ValueDefinition
  FunctionDefinition
  AbstractDefinition

TypeDefinition:
  `type` *TypeConstructor {TypeVariable}* `=` *MonoType*
  AbstractTypeDefinition

TypeConstructor:
  CapitalizedId

TypeVariable:
    LowercaseId

AbstractTypeDefinition:
    `type` *TypeConstructor {TypeVariable}*

MonoType:
    TypeA1
    FunctionType

FunctionType:
    TypeA1 `->` *TypeA1*

TypeA1:
    TypeA2
    ParameterizedType
    PartialRecordType

ParameterizedType:
    TypeConstructor {TypeA2}

PartialRecordType:
    TypeA2 TakePut TakePutFields

TakePut: one of
    `take` `put`

TakePutFields:
    FieldName
    `(` *[FieldName {* `,` *FieldName}]* `)`
    `( .. )`

FieldName:
    LowercaseId

TypeA2:
    AtomType
    `#` *AtomType*
    AtomType `!`

AtomType:
    `(` *MonoType* `)`
    TypeConstructor
    TupleType
    RecordType
    VariantType
    TypeVariable

TupleType:
    `()`
    `(` *MonoType* `,` *MonoType {* `,` *MonoType}* `)`

RecordType:
    `{` *FieldName* `:` *MonoType {* `,` *FieldName* `:` *MonoType}* `}`

VariantType:
`<` *DataConstructor {TypeA2} {*`|` *DataConstructor {TypeA2}}* `>`

DataConstructor:
CapitalizedId

ValueDefinition:
Signature Variable `=` *Expression*

Variable:
LowercaseId

Signature:
Variable `:` *PolyType*

PolyType:
MonoType
`all` *PermSignatures* `.` *MonoType*

PermSignatures:
PermSignature
`(` *PermSignature {*`,` *PermSignature}* `)`

PermSignature:
TypeVariable
TypeVariable :< Permissions

Permissions:
Permission {Permission}

Permission: one of
`D S E`

FunctionDefinition:
Signature Variable IrrefutablePattern `=` *Expression*
Signature Variable Alternative {Alternative}

AbstractDefinition:
Signature

Pattern:
`(` *Pattern* `)`
IrrefutablePattern
LiteralPattern
VariantPattern

LiteralPattern:
BooleanLiteral
IntegerLiteral
CharacterLiteral

IrrefutablePattern:
Variable
WildcardPattern
TuplePattern
RecordPattern

WildcardPattern:
  `_`

TuplePattern:
  `()`
  `(` *IrrefutablePattern* `,` *IrrefutablePattern* {`,` *IrrefutablePattern*} `)`

RecordPattern:
  Variable `{` *RecordMatchings* `}`
  `#` `{` *RecordMatchings* `}`

RecordMatchings:
  RecordMatching {`,` *RecordMatching*}

RecordMatching:
  FieldName [= *IrrefutablePattern*]

VariantPattern:
  DataConstructor {*IrrefutablePattern*}

Expression:
  BasicExpression
  MatchingExpression
  LetExpression
  ConditionalExpression

BasicExpression:
  BasExpr
  BasExpr `;` *Expression*

MatchingExpression:
  ObservableBasicExpression Alternative {*Alternative*}

ObservableBasicExpression:
  BasicExpression
  BasicExpression {`!` *Variable*}

Alternative:
  `|` *Pattern PArr Expression*

PArr: one of
  `->` `=>` `~>`

LetExpression:
  `let` *Binding* {`and` *Binding*} `in` *Expression*

Binding:
  IrrefutablePattern [`:` *MonoType*] = *ObservableExpression*

ObservableExpression:
  Expression
  Expression {`!` *Variable*}

ConditionalExpression:
  `if` *ObservableExpression* `then` *Expression* `else` *Expression*

BasExpr:
Term
FunctionApplication
OperatorApplication
PutExpression
MemberAccess

FunctionApplication:
BasExpr BasExpr

OperatorApplication:
UnaryOP BasExpr
BasExpr BinaryOp BasExpr

UnaryOp: one of
complement not

BinaryOp: one of
`o * / % + - >= > == /= < <= .&. .^. .|. >> << && || $`

PutExpression:
BasExpr { Record Assignments }

RecordAssignments:
RecordAssignment { , RecordAssignment}

RecordAssignment:
FieldName [= *Expression]*

MemberAccess:
BasExpr `.` *FieldName*

Term:
`(` *Expression* `)`
Variable
LiteralTerm
TupleTerm
RecordTerm
VariantTerm
LambdaTerm
PolyVariable

LiteralTerm:
BooleanLiteral
IntegerLiteral
CharacterLiteral
StringLiteral

BooleanLiteral: one of
`True False`

IntegerLiteral:
: *DecDigits*
: `0x` *HexDigits*
: `0X` *HexDigits*
: `0o` *OctDigits*
: `0O` *OctDigits*

DecDigits: informal
: A sequence of decimal digits 0-9.

HexDigits: informal
: A sequence of hexadecimal digits 0-9, A-F.

OctDigits: informal
: A sequence of octal digits 0-7.

CharacterLiteral: informal
: An ASCII character enclosed in single quotes.

StringLiteral: informal
: A sequence of ASCII characters enclosed in double quotes.

TupleTerm:
: `()`
: `(` *Expression* `,` *Expression {* `,` *Expression}* `)`

RecordTerm:
: `# {` *RecordAssignments* `}`

VariantTerm:
: *DataConstructor {Term}*

LambdaTerm:
: \ *IrrefutablePattern [* `:` *MonoType]* `=>` *Expression*

PolyVariable:
: *Variable* `[` *OptMonoType {* `,` *OptMonoType}* `]`

OptMonoType:
: *MonoType*
: `_`

LowercaseID: informal
: A sequence of letters, digits and underscore symbols starting with a lowercase letter

CapitalizedID: informal
: A sequence of letters, digits and underscore symbols starting with an uppercase letter