Cogent Language Report

Zilin Chen

30 January 2017

1 Surface Syntax

1.1 Grammar

Note: The syntax of Cogent is not restricted to regular expression, thus the following EBNF definition is incomplete and is not strictly formal. In particular, indentation rules are specified in a semi-formal manner. Superscript (c) means the associated construct starts at column c.

variable / field	v	::=	`[a-z][A-Za-z1-9'_]*` | `_[A-Za-z1-9'_]+`
type construtor / tag	T	::=	`[A-Z][A-Za-z1-9'_]*`
binding	b	::=	p_i `(:` τ`)?` `=` e `(!`v`)*`
record matching	r_m	::=	v `(=` p_i`)?`
strict record matchings	$\overline{r_m}^!$	::=	r_m`(,` r_m`)*`
record matchings	$\overline{r_m}$	::=	`(`$\overline{r_m}^!$`,)?` `..`
record assignment	r_a	::=	v `(=` $e^{(1)}$`)?`
strict record assignments	$\overline{r_a}^!$	::=	r_a`(,` r_a`)*`
record assignments	$\overline{r_a}$	::=	`(`$\overline{r_a}^!$`,)?` `..`
irrefutable pattern	p_i	::=	v `({`$\overline{r}$`})?`
		|	`(`p_i`(,` p_i`)*)?`
		|	`#{`$\overline{r}$`}`
		|	`_`
pattern	p	::=	`True` | `False`
		|	T p_i | p_i
		|	*integer* | *'char'*
term	ε	::=	`(inline)?` v `([`τ`(,` τ`)*])?`
		|	`True` | `False`
		|	*integer*
		|	*'char'*
		|	*"string"*
		|	T ε
		|	`(`$e^{(1)}$`(,` $e^{(1)}$`)*)?`
		|	`#{`$\overline{r}^!$`}`
basic expression	$e_b{}^{(c)}$	::=	e'_b | e'_b`;` $e^{(c)}$
	e'_b		*operator precedence descending*
		::=	ε`.`v
		|	`(complement|not)` ε
		|	ε ε
		|	ε `{`$\overline{r}$`}`
		|	ε `(*|/|%)` ε
		|	ε `(+|-)` ε
		|	ε `(>=|>|==|/=|<|<=)` ε
		|	ε `.&.` ε *bitwise-and*
		|	ε `.^.` ε *bitwise-xor*
		|	ε `.|.` ε *bitwise-or*
		|	ε `(>>|<<)` ε
		|	ε `&&` ε
		|	ε `||` ε
exression matching	$m^{(c)}$	::=	$e_b{}^{(c)}$ `((!`v`)*` $a^{(>c)+}$`)?` a^+ *are aligned*
expression	$e^{(c)}$	::=	`let` b `(and` b`)*` `in` $e^{(c)}$
		|	`if` $e^{(c)}$ `(!`v`)*` `then` $e^{(c)}$ `else` $e^{(c)}$
		|	$m^{(c)}$
kind	κ	::=	`[D|S|E]+`
kind signature	s	::=	v `(:<` κ`)?`

kind signatures	$\overline{s}$	::=	$s\texttt{(}s\texttt{(, }s\texttt{)}^*\texttt{)}$
alternative	$a^{(c)}$	::=	$\texttt{| }p\texttt{ (=>|->|~>) }e^{(c)}$
atomtype	τ_a	::=	$\texttt{(}\tau\texttt{(, }\tau\texttt{)}^*\texttt{)}$
		|	$\texttt{\{}v\texttt{ : }\tau\texttt{(, }v\texttt{ : }\tau\texttt{)}^*\texttt{\}}$
		|	$\texttt{<}T\ \tau_{A2}\ \texttt{(| }T\ \tau_{A2}\texttt{)}^*\texttt{>}$
		|	v
		|	T
type A2	τ_{A2}	::=	$\texttt{\#}\tau_a$
		|	$\tau_a\texttt{!}^?$
type A1	τ_{A1}	::=	$T\ \tau_{A2}^*$
		|	$\tau_{A2}\ \texttt{((take|put) (}v\texttt{|((}v\texttt{(, }v\texttt{)}^*\texttt{|..))))}^?$
mono-type	τ	::=	$\tau_{A1}\ \texttt{(->}\ \tau_{A1}\texttt{)}^?$
poly-type	σ	::=	$\texttt{(all }\overline{s}\texttt{.)}^?\ \tau$
function signature	f_σ	::=	$v\texttt{ : }\sigma$
function definition	f	::=	$v\ (a^{(c)^+}\texttt{|}v\texttt{ = }e^{(1)})$
type definition	t	::=	$\texttt{type }T\ v^*\ \texttt{(= }\tau\texttt{)}^?$
include	i	::=	$\texttt{include "}filepath\texttt{"}$
program	p	::=	$(f_\sigma\|f_\sigma\ f\|t\|i)^{(0)^+}$

a^+ are aligned, c is the first column of a^+

1.2 Comments

They are exactly the same as those of Haskell. Single-line comments start with `--`, block comments are surrounded by `{-` and `-}`.

1.3 Pragmas

Cogent supports several pragmas, or instructions to the compiler placed in the source code. Pragmas don't normally affect the meaning of the program, but they might affect the efficiency of the generated code.

Pragmas all take the form `{-# word ... #-}` where *word* indicates the type of pragma, and is followed optionally by information specific to that type of pragma. Case is ignored in *word*. The various values for *word* that Cogent understands are described in Section 6.5. The layout rule does NOT apply in pragmas.

2 Abstract Syntax

irref. pattern	*irrp*	::=	pv
		|	$\texttt{(}\overline{irrp_i}\texttt{)}$
		|	$\texttt{\#\{}\overline{pfield_i}\texttt{\}}$
		|	`_`
		|	`()`
		|	$pv\ \texttt{\{}\overline{pfield_i}\texttt{\}}$
pattern field	*pfield*	::=	`..`
		|	$f = irrp$
pattern	*pat*	::=	$T\ \overline{irrp_i}$
		|	n
		|	b
		|	c
		|	*irrp*
alternative	*alt*	::=	$pat \Rightarrow e$
opt. ty. ann.	$\cdot^\tau$	::=	$\cdot : \tau$
		|	$\cdot$
binding	b	::=	$irrp^\tau = e\ \texttt{!}\overline{v_j}$
expression	e	::=	$op\ \overline{e_i}$
		|	v
		|	$\texttt{case }e\ \texttt{!}\overline{v_j}\ \texttt{of }\overline{alt_i}$
		|	$v\ [\overline{\tau_i}]$
		|	$T\ \overline{e_i}$
		|	$e\texttt{; }e$
		|	$e\ e$
		|	$\texttt{if }e\ \texttt{!}\overline{v_j}\ \texttt{then }e\ \texttt{else }e$
		|	$e\texttt{.}f$
		|	`()`
		|	ℓ_n
		|	ℓ_b
		|	ℓ_c

		|	ℓ_s
		|	$(\overline{e_i})$
		|	$\texttt{\#\{}\overline{f_i = e_i}\texttt{\}}$
		|	$\texttt{let } \overline{b_i} \texttt{ in } e$
		|	$v\ \texttt{\{}\overline{field_i}\texttt{\}}$
field	$field$	::=	`..`
		|	$f = e$
type	τ	::=	τ_{NF}
		|	$\texttt{\#}\tau$
		|	$\tau\texttt{!}$
		|	$\tau \texttt{ take } \overline{f_i}$
		|	$\tau \texttt{ take ..}$
		|	$\tau \texttt{ put } \overline{f_i}$
		|	$\tau \texttt{ put ..}$
nor. type	τ_{NF}	::=	$T\ \overline{\tau_i}\ \varsigma$
		|	$t\texttt{!} \mid t$
		|	$\tau \to \tau$
		|	$\texttt{\{}\overline{f_i : {\tau_i}^?}\texttt{\}}$
		|	$\langle \overline{T_i \mapsto \overline{\tau_{ij}}} \rangle$
		|	$(\overline{\tau_i})$
poly-type	σ	::=	$\overline{t_i <: \kappa_i}\ \tau$
top level	top	::=	$\texttt{include } s$
		|	$T\ \overline{t_i} = \tau$
		|	$T\ \overline{t_i}$
		|	$v : \sigma$
		|	$v : \sigma = \overline{alt_i}$
		|	$v : \tau = e$

3 Surface Type Inference

4 Desugaring

Note: :: is meta-lang.

$$\boxed{\Gamma \vdash e \rightsquigarrow e \quad (e \text{ is core syntax. See [1]})}$$

$$\frac{\Gamma \vdash op \rightsquigarrow o \qquad \text{for each } i\text{: } \Gamma \vdash e_i \rightsquigarrow e_i}{\Gamma \vdash op\ \overline{e_i} \rightsquigarrow o(\overline{e})}$$

$$\frac{v \in \Gamma}{\Gamma \vdash v \rightsquigarrow v} \qquad \frac{(v \mapsto e) \in \Gamma \qquad \Gamma \vdash e \rightsquigarrow e}{\Gamma \vdash v \rightsquigarrow e}$$

$$\frac{\Gamma \vdash \texttt{case } e \texttt{ of } \overline{alt_i} \rightsquigarrow_{a^*} e}{\Gamma \vdash \texttt{case } e \texttt{ of } \overline{alt_i} \rightsquigarrow e} \qquad \frac{\Gamma \vdash e \rightsquigarrow e \qquad v \text{ fresh} \qquad \Gamma, v \vdash \texttt{case } v :: \tau \texttt{ of } \overline{alt_i} \rightsquigarrow_{a^*} e'}{\Gamma \vdash \texttt{case } e :: \tau\ \texttt{!}\overline{v_j} \texttt{ of } \overline{alt_i} \rightsquigarrow \textbf{let!}(\overline{v_j})\ v = e\ \textbf{in}\ e'}$$

$$\frac{\text{for each } i\text{: } \Gamma \vdash \tau_i \rightsquigarrow \tau_i}{\Gamma \vdash v\ [\overline{\tau_i}] \rightsquigarrow v[\overline{\tau_i}]}$$

$$\frac{}{\Gamma \vdash T\ \rightsquigarrow \mathsf{T}} \qquad \frac{\Gamma \vdash e \rightsquigarrow e}{\Gamma \vdash T\ e \rightsquigarrow \mathsf{T}\ e} \qquad \frac{\Gamma \vdash (\overline{e_i}) :: \tau \rightsquigarrow e}{\Gamma \vdash T\ \overline{e_i} ::_N \langle T \mapsto \tau, \cdots \rangle \rightsquigarrow \mathsf{T}\ e}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1 \qquad \Gamma, v \vdash e_2 \rightsquigarrow e_2 \qquad v \text{ fresh}}{\Gamma \vdash e_1\texttt{;}\ e_2 \rightsquigarrow \textbf{let}\ v = e_1\ \textbf{in}\ e_2}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1\ e_2}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2 \qquad \Gamma \vdash e_3 \rightsquigarrow e_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1 \qquad v \text{ fresh} \qquad \Gamma, v \vdash e_2 \rightsquigarrow e_2 \qquad \Gamma, v \vdash e_3 \rightsquigarrow e_3}{\Gamma \vdash \texttt{if } e_1\ \texttt{!}\overline{v_j} \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \textbf{let!}(\overline{v_j})\ v = e_1\ \textbf{in if } v \textbf{ then } e_2 \textbf{ else } e_3}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e}{\Gamma \vdash e\texttt{.}f \rightsquigarrow e\texttt{.f}}$$

$$\frac{}{\Gamma \vdash \texttt{()} \rightsquigarrow \texttt{()}} \qquad \frac{\Gamma \vdash \ell \rightsquigarrow_\ell \ell}{\Gamma \vdash \ell \rightsquigarrow \ell}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2}{\texttt{-fno-tuples-as-sugars}}\ \text{(tuples are not in Core)} \qquad \frac{\text{for each } i\text{: } \Gamma \vdash e_i \rightsquigarrow e_i}{\Gamma \vdash (\overline{e_i}) \rightsquigarrow \texttt{\{p}\overline{i\ = e_i}\texttt{\}}}$$

$$\frac{\text{for each } i\text{: } \Gamma \vdash e_i \rightsquigarrow e_i}{\Gamma \vdash \texttt{\#\{}\overline{f_i = e_i}\texttt{\}} \rightsquigarrow \{\overline{\mathrm{f}_i \ = e_i}\}}$$

$$\frac{\Gamma \vdash \texttt{let } irrp = e' \texttt{ in } e \rightsquigarrow_a e}{\Gamma \vdash \texttt{let } irrp^{\tau} = e' \texttt{ in } e \rightsquigarrow e} \qquad \frac{\Gamma \vdash e' \rightsquigarrow e' \qquad \Gamma, pv \vdash e \rightsquigarrow e}{\Gamma \vdash \texttt{let } pv^{\tau} = e' \texttt{ !}\overline{v_j} \texttt{ in } e \rightsquigarrow \textbf{let!}(\overline{v_j})\ pv = e' \textbf{ in } e}$$

$$\frac{v \text{ fresh} \qquad \Gamma \vdash \texttt{let } pv = e' \texttt{ !}\overline{v_j}, irrp^{\tau} = pv \texttt{ in } e \rightsquigarrow e}{\Gamma \vdash \texttt{let } irrp^{\tau} = e' \texttt{ !}\overline{v_j} \texttt{ in } e \rightsquigarrow e} \qquad \frac{\Gamma \vdash \texttt{let } b_0 \texttt{ in let } b_i \texttt{ in } e \rightsquigarrow e}{\Gamma \vdash \texttt{let } b_0, \overline{b_i} \texttt{ in } e \rightsquigarrow e}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e}{\Gamma \vdash e\ \texttt{\{\}} \rightsquigarrow e} \qquad \frac{\Gamma \vdash e \rightsquigarrow e \qquad \Gamma \vdash e' \rightsquigarrow e'}{\Gamma \vdash e\ \texttt{\{}f = e'\texttt{\}} \rightsquigarrow \textbf{put } e.\mathrm{f} := e'} \qquad \frac{\Gamma \vdash ((e :: \tau \texttt{ take } \overline{f_i})\ \texttt{\{}f_0 = e_0\texttt{\}})\ \texttt{\{}\overline{f_i = e_i}\texttt{\}} \rightsquigarrow e}{\Gamma \vdash e\ \texttt{\{}f_0 = e_0, \overline{f_i = e_i}\texttt{\}} ::_N \tau \rightsquigarrow e}$$

$$\boxed{\Gamma \vdash \texttt{case } e' \texttt{ of } \overline{alt_i} \rightsquigarrow_{a^*} e}$$

$$\frac{\Gamma \vdash \texttt{case } e' \texttt{ of } alt \rightsquigarrow_a e}{\Gamma \vdash \texttt{case } e' \texttt{ of } alt \rightsquigarrow_{a^*} e}$$

$$\frac{\Gamma, pv \vdash e_0 \rightsquigarrow e_0 \qquad v' \text{ fresh} \qquad \Gamma, v' \vdash \texttt{case } v' :: \langle \overline{T_j \mapsto \tau_j} \rangle \texttt{ of } \overline{alt_i} \rightsquigarrow_{a^*} e}{\Gamma \vdash \texttt{case } v ::_N \langle T \mapsto \tau, \overline{T_j \mapsto \tau_j} \rangle \texttt{ of } T\ pv \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} \textbf{case } v \textbf{ of } \mathsf{T}\ pv \to e_0 \textbf{ else } v' \to e}$$

$$\frac{v' \text{ fresh} \qquad \Gamma \vdash \texttt{case } v \texttt{ of } T\ v' :: \tau \Rightarrow \texttt{let } irrp = v' \texttt{ in } e_0, \overline{alt_i} \rightsquigarrow_{a^*} e}{\Gamma \vdash \texttt{case } v ::_N \langle T \mapsto \tau, \cdots \rangle \texttt{ of } T\ irrp \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} e}$$

$$\frac{\Gamma \vdash \texttt{case } v :: \tau \texttt{ of } T\ (\overline{irrp_m}) \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} e}{\Gamma \vdash \texttt{case } v ::_N \tau \texttt{ of } T\ \overline{irrp_m} \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} e}$$

$$\frac{\Gamma \vdash v \rightsquigarrow e' \qquad \Gamma \vdash e_0 \rightsquigarrow e_0 \qquad \Gamma \vdash \texttt{case } e_0 \texttt{ of } \overline{alt_i} \rightsquigarrow_{a^*} e}{\Gamma \vdash \texttt{case } v \texttt{ of } \ell \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} \textbf{if } e' \texttt{==} \ell \textbf{ then } e_0 \textbf{ else } e}$$

$$\frac{pv \text{ fresh} \qquad \Gamma \vdash \texttt{let } pv = e' \texttt{ in case } pv \texttt{ of } \overline{alt_i} \rightsquigarrow e}{\Gamma \vdash \texttt{case } e' \texttt{ of } pat \Rightarrow e_0, \overline{alt_i} \rightsquigarrow_{a^*} e}$$

5 Types

5.1 Base Types

Primitive types In Cogent, we have a few built-in types, they are `U8`, `U16`, `U32`, `U64`, `Bool`, `Char` and `String`. For example, `U8` is unsigned 8-bit integer types, which correspond to `uint8_t` in C. Integer literals can be written in octal (e.g. `0072`, `0o24`; read "zero-oh"), decimal or hexadecimal (e.g. `0XA2C`, `0x1fd`). `Bool` type in Cogent has only two values, `True` and `False`. `Char` type is defined as a synonym to `U8` (this may change in the future). `String` is a type which does not relate to `Char`. This type is mainly used in debugging functions (`printf` alike), as there are no `String`-specific operations that you can work on them.

Abstract types Apart from these primitive types, users can declare their own types, as building blocks, in Cogent. They are abstract in the language and their definitions are implemented in C directly. For example, one can do:

```
1  type T
```

Now we have a type `T` in our program.

5.2 Linear type system

One major feature of Cogent is its linear type system. For detailed explanations, see [2] and our publications. Generally speaking, *linear* objects are dynamically and explicitly allocated on the heap. By contrast, anything else are statically and implicitly allocated on the stack. For heap-allocated objects, we need to manage the references to them, which is where linear types help, to ensure that each object on the heap is pointed by **exactly** one reference.

Kinds In order to classify linearity properties of each type of objects, we introduce kinds. In Cogent, kinds are denoted by a combination of three letters – `D` (discardable), `S` (shareable) and `E` (escapable). It's obvious to see that for any linear object, it's type must be non-`D` and non-`S`. We cannot throw away a linear reference, nor can we duplicate (share) it. In Cogent, `D` and `S` must be present/absent in a well-formed kind together, although in the formal semantics, we usually only focus on one side of it. In the following text, we just write `DS` instead of having them separately. We now have at most 4 different combinations. Getting simpler, right?

In some cases, we want to create read-only "copies" of an object (say, give them to other functions which only need to inspect them). It in fact doesn't copy it in memory (sure, because we don't modify it anyway), so it would be more accurate to say "create read-only copies of the reference". In Cogent's purely functional setting, we just don't have to distinguish references and the actual stored objects. For a linear object (kinded without `DS`), we can temporarily put it

in a context where it becomes read-only. In this context, this linear object can be freely discarded or duplicated (for the reason we explained earlier), thus the kind of whose type changes, from linear to read-only, obtaining `DS`. But obviously, we cannot simply allow these read-only copies to escape this temporary context, otherwise we may have no or multiple references (one read-write, and arbitrarily many read-only) outside. For example, consider the following expression:

```
1  let obj_ro = let (obj_ro1, obj_ro2) = dup obj !obj
2               in obj_ro1
3   in foo (obj_ro, obj)
```

In this expression, `obj` is a linear object. `!obj` creates a context inside which `obj` becomes read-only. [1] The scope of read-only `obj` is in the binding (the inner `let-in` expression). The code in the binding is totally valid, as we can duplicate the read-only `obj` and discard one of them (`obj_ro2`). The problem of this whole expression is that `obj_ro1` escapes its context, where our type system tells us that it cannot escape. As we can see that in the body of the outer binding, `obj_ro` and `obj` are referencing to the same thing, which violates linear type system.

We use `E` to encode this property. A normal linear type has `E`-property. After coming into a read-only context, it becomes `DS`-kinded, losing `E`. For non-linear types, they always have all `DSE`, as we don't put any constraints on the usage of them. A type without any of `DSE` is useless, so we don't consider them.

To summarise, we have 3 kinds in Cogent, namely 0-kind (linear but read-only, `{DS}`), 1-kind (linear, `{E}`), and 2-kind (non-linear, `{DSE}`).

5.3 Composite types

Using the aforementioned base types, with the help of type constructors and modifiers, we can build up composite types. We generally have three different kinds of composite types — record types, variant types and tuples.

Record types Similar to C structs, each record consists of more than 1 field. Fields are order-sensitive. For example, `{f1:T1, f2:T2, f3:T3}` is an intensionally different type to `{f2:T2, f1:T1, f3:T3}`. Records (or sometimes to disambiguate we call them *boxed* records) is a data structure allocated on the heap, explicitly, by abstract Cogent functions. In pure Cogent (without the help of abstract functions), there is no way to allocate/create a boxed record, even though you have all the fields at hand, nor a way to free one. Each boxed record comes with one reference (to the "box" itself because the "box" is on the heap) regardless of its fields (which are unnecessarily on the heap, depending on each field's type), so every boxed record is linear, even though all its fields are non-linear or taken. A boxed record is analogous to a box with one linear reference, which is capable of encapsulating all (if any) the references to its linear fields. So from outside a boxed record, we have no access to its linear fields. We will introduce term level operations for field access in later sections.

Unboxed records In contrast to boxed records, we also have unboxed records. The major difference is that the record itself is allocated on the stack, implicitly by the program. That said, the linearity of an unboxed record solely depends on its fields. If any field is linear, then the whole is linear. In Cogent we can construct or destruct an unboxed records cheaply, supplying or obtaining all it fields (especially linear ones) respectively. An unboxed record is written as `#{f1:T1, f2:T2}`, for example, with a leading `#`. It's worth mentioning, though, the linear fields of an unboxed record are still allocated on the heap.

Tuples Tuples, written as `(T1,T2)` can be deemed as unnamed unboxed records. It is also order-sensitive. In the core calculus of Cogent, tuples are nested pairs, grouping from the right. For example, `(a,b,c,d)` is equal to `(a,(b,(c,d)))`. Other than that, it is pretty much the same as an unboxed record.

5.4 Type Operators

Cogent provides several type operators. `take` and `put` operators can be applied to record types (be it boxed or not). They will be introduced in detail in Section 6.2. Here, we focus on the other two operators: `#` and `!`.

Unbox `#` is the sigil for unboxed data types. It works for abstract types and records. For an abstract type `A`, `#A` is the unboxed version of it. `#` has to be put at its use site. For example, if we have code:

```
1  type #A
2  foo : A -> Bool
3  bar : #A -> Bool
```

Note that the `A` in line 2 is still the boxed version, the `#A` in line 3 is an unboxed type. We can tell that type `#A` is of 2-kind.

If we apply `#` to records, they become unboxed records as described earlier. If the record has a synonyms, then the operator can be put either at definition site or use site. For example,

[1] More about the term level operation to create the read-only context, see Section 6.5.

```
type R = {f1 : A, f2 : B}
foo : #R -> ()
type S = #{f1 : A, f2 : B}
bar : S -> ()
baz : #S -> ()
```

In this program, `#R == S == #S`, because the types are purely structural and `#` is idempotent. Keep in mind that a type synonym is just a "macro", so it is totally valid to put any type operators that works for records to synonyms that are indeed records. We will not repeat this point in the following text.

Bang The `!` suffix turns any linear types into read-only ones. It is also idempotent and has no effect on 2-kinded types. The underlying C types for a linear type and its banged variant are identical.

6 Patterns and Expressions

We have seen a large portion of Cogent's type system. On term level, similar to most mainstream functional language, we can also categorise things into patterns and expressions, which reside on LHS and RHS of `=`s respectively (roughly speaking).

Patterns In Cogent, pattern matching is not as strong as some commercial level languages. When we deal with nested patterns, inner patterns have to be irrefutable, which can be seen from the syntax (see Section 1.1).

Expressions They are quite standard and the semantics of them can be derived straightforwardly from the syntax. In the following paragraphs we are just going to highlight some of the "unusual" features.

6.1 Literals and Variables

See Section 5.1.

6.2 Dealing with records

We first look at boxed records. Assume that we have an record `r` of type `{f1:T1, f2:T2, f3:U32}` (for brevity, both in a Cogent program and in this documentation, we would like to define it as `R` which is intensionally the same), where both `T1` and `T2` are boxed abstract types. Generally, there are three operations relating to record fields — *member*, *take* and *put*. Member operation behaves uniformly for linear fields and non-linear ones. The precondition is that the record which contains the field has to be shareable (i.e. with kind `S`). In the case of `r` (of type `R`), it is not shareable as it contains linear fields `f1` and `f2`. In order to temporarily turn it into a shareable one, we use `!` operator, which is introduced later in Section 6.5. By doing `(... r.f1 ...) !r` (the ellipses are meta-syntax, indicating the context in which `r` is read-only and consequently shareable). It's worth stressing that by doing member extraction, what you get is merely a *value*. It does not correspond to any memory locations on the heap.

To access the memory locations corresponding to fields, we need take. Take operation is in the form of pattern matching in the surface language, written `r' {f1 = f1_obj} = r` (of course, more than 1 field can be taken at once). It only works for records that are not read-only as a whole (fields do not matter) as we are destructing the record. This pattern matching does the following:

1. it takes field `f1` out of record `r`, and binds it to name `f1_obj`;
2. it gives the "new" record `r'`, with field `f1` taken.

After doing it, the record `r'` no more contains the reference to object `f1_obj` (via its field `f`), which is now a standalone reference. To interpret it, we can:

- functionally, the old value `r` *becomes* two new values, one of type `T1`, and the other of type `R take f1` (more detail in a second);
- and imperatively, you can consider it as assigning the pointer to `r -> f1` a new pointer `f1_obj` and `r -> f1` becomes `NULL` in the record.

We can see that taken fields are reflected in the type level. Apart from "complete" record types introduced earlier, we can also have "partial" record types, which are first class as well. These "partial" record types have some (or all) of its fields taken. These types are denoted as `R take f1`, for example. If more than one fields are taken, then we need to put them in parentheses, separated by commas (e.g. `R take (f1,f2)`). If all are taken, a syntactic sugar is `R take (..)`. Dually, type `R put f1` means that only `f1` is *not* taken. It is just for convenience, when you have more fields taken than not. It is obvious that for `R`, we have `R take f1 == R put (f2,f3)`.

The dual operation of take (the one in expressions) is, as you might have guessed, put. Note that the type operator `put` is merely a syntactic sugar for type-level `take`, whereas put operation (ironically not spelt out as `take` in the surface language) is a real thing in expressions, even in the core language.

Put, in contrast to take, appears in expressions (as opposed to patterns) in Cogent. Continuing on our running example, if we want to put object `f1_obj` back to `r'`, we can do `r' {f1 = f1_obj}`. You can see that the syntax is nearly identical to take, so do not get confused. Note that Cogent is a functional language, so this is an expression, not a

command/statement. It does not have any effect on the state. Linear type system requires you bind the put-expression to a new binder, say `r"`. So at this stage, `r == r"`, and `r'` and `f1_obj` no longer exist!

Now, some syntactic sugar for take and put. If the field name is the same as the binder name, we can use name punning. E.g. instead of `let r' {f1 = f1, f2 = f2} = r in ...`, we can write `let r' {f1,f2} = r in ...`. If we want to take all (yet untaken) fields (if at all), then we can write `let r' {..} = r in ...`. [2] Both of them apply to put as well, as expected. Once again, `take` and `put` are type-level syntax, and take and put in term-level (as patterns and expressions) have similar syntax using curly brackets.

The aforementioned take operation applies to both linear fields and non-linear ones, although it turns out to be overweighted for non-linears. In order to "take" a non-linear field, the first choice is usually the member operation. To "put" a non-linear field, namely to update it destructively, we can do it by `r' = r {f1 = new_f1}`. This is just the normal put syntax. For non-linear field, a special feature is, the field can be put in a row more than once (but can only be taken once, as usual). In theory, because the field is store on the stack, multiple takes is also valid. We forbid it for the purpose of modelling uninitialised field. When allocating a new record, we generally leave its non-linear fields taken as well to indicate that these fields are not yet initialised.

6.3 Control Flow and Pattern Matching

Control flow in Cogent is simple. We basically support `if-then-else` expressions and case distinctions. The conditionals are standard. Just be reminded of the functional semantics of it. Conditionals can be nested and users should consult Section 1.1 about where to put parenthesis. Case distinctions are in the form of guards. Semantically, conditionals can always be represented equivalently by cases. The patterns following each guard matches the value of the scrutinee. Pattern matching has to be exhaustive. For more examples, please see files in directory `cogent/tests`. Since each Cogent function allows at most one definition binding, to pattern match argument, we can write a program like:

```
1  type A
2
3  bar : <A U8 | B Bool | C #A> -> ()
4  bar | A a -> ()
5      | B b -> ()
6      | c   -> ()
```

instead of

```
1  type A
2
3  bar : <A U8 | B Bool | C #A> -> ()
4  bar arg = arg | A a -> ()
5                | B b -> ()
6                | c   -> ()
```

6.4 Unboxed Records and Tuples

They have similar semantics. Each Cogent function takes exactly one argument and returns exactly one resultant. One way to give multiple parameters is to use unboxed records or tuples, which can be pattern matched directly. One reason to use unboxed records is, by name punning, we can achieve a C-style function. For instance,

```
1  foo : #{f1 : A, f2 : B} -> ()
2  foo #{..} = ... bar (f1, f2) ...
3  -- foo #{f1 = f1_var, f2 = f2_var} = ...
```

using the record wildcard syntax, we introduce names `f1` and `f2` into scope. Syntax `#{...}` (the `...` is meta-syntax) is pattern matching the argument. The commented line is the general syntax to match an unboxed record. The downside of it, however, is that the record is passed by value instead of by reference.

Let's take a look at another example:

```
1  bar : (A, B, C) -> ()
2  bar (a, bc) = let (b, c) = bc in ...
3  -- bar (a, b, c) = ...
```

Line 1 is one way to pattern match the argument, since tuples are right associative. Line 2 is the flattened way to match.

Another usage of unboxed records is construction. We know that there is no way to create a *boxed* record in pure Cogent. But for unboxed ones, we can construct one on the fly, which resembles creation of tuples. The code looks like: `#{f1 = f1_value, f2 = f2_value}`, which creates an unboxed record with fields `f1` and `f2`, bound to value `f1_value` and `f2_value` respectively. The effectively equivalent tuple version is `(f1_value, f2_value)`. We can summarise that, for unboxed records and tuples, their patterns and expressions share the same shape.

[2] It is a bit font overloading in this documentation, `..` is Cogent syntax, and `...` is meta-level text for any omitted source code.

6.5 Let

Cogent `let`-binding is standard. A sequence of `let`-bindings are separated by keyword `and`. When the binder is not important, we can use a wildcard `_` as a placeholder. Alternatively, we can use sequence syntax, separated by `;`. In this case, the binder is unspoken (thus the type of the expression should be trivial in terms of linearity). Each binding can be given a type annotation, written as `let x : t = b in body`.

6.6 !

In accordance to Cogent's linear type system, there is a way to turn a linear object into a read-only one. The essence is to use `!` (read: bang) operator to create a context in which the bang'ed linear variables are read-only. The language allows users to bang several variable in one go. There are 3 constructs in Cogent that can accommodate bangs — `let`, `if` and case distinction. The scope of bang'ed variables are highlighted in the following (trivial) examples.

```
type A

foo : A -> A
foo a = let (a1,a2) = pair a !a in a

sizeA : A! -> U32

bar : A -> A
bar a = if sizeA a > 4 !a then a else a

bar' : A -> A
bar' a = sizeA a > 4 !a
           | True  -> a
           | False -> a
```

6.7 Polymorphic Functions

Cogent supports polymorphic functions. To define one, we need to explicit qualify the type variable in the type signature. E.g.

```
iterate: all (y, r, s, acc, obsv). #{
         gen: Generator y r s acc obsv!,
         cons: Consumer y r s acc obsv!,
         acc: acc,
         obsv: obsv!} -> IterationResult acc r s
```

In this example, `y`, `r`, `s`, `acc` and `obsv` are type variables. In some cases, we would like to constrain the kind of some type variables, which can be done using kind constraints (`:<`). For example,

```
array_create    : all (a :< E  ). (Ex, U32) -> R (Ex, Array     a) Ex
wordarray_create: all (a :< DSE). (Ex, U32) -> R (Ex, WordArray a) Ex
```

Comparing `array_create` and `wordarray_create`, we can see that the former function requires type argument `a`, namely the element type, to be `E`scapable (which means the elements are not read-only), whereas the later requires strictly more, that the element type has to be `D`iscardable, `S`hareable and `E`scapable (which is an approximation of "Word" types).

To call a polymorphic function, since Cogent compiler currently lacks the capability of inferring poly-functions (which is theoretically possible), the user has to explicitly apply a function to type arguments. For instance,

```
let limit = ext2_last_byte (inode, n) - u16_to_u64 (ext2_dir_rec_len (1)) !inode
and ((ex, maybe_dirent, kaddr, _), res) =
      iterate_do [Ext2DirEnt, (), U32, EmptyDirentsInPageAcc, DirentsInPageObsv]
        (#{ init = dirent,
            gen = get_next_dirent_from_page4,
            cons = check_dirent_empty,
            acc = (ex, None (), kaddr, 0),
            obsv = limit })
 in ...
```

The list in square brackets are types which instantiate the type variables respectively.

TODO: How C code is generated? how to use the compiler and its intermediate outputs? what're the must-knows in the compiler?

7 Flags in Cogent Compiler

8 How to optimise Cogent programs?

References

[1] Liam O'Connor-Davis. Definition of Core Cogent, 2016.

[2] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.