Proposal for Dargent Improval

Gunnar Teege

14 November 2019

Main idea: integrate layout specifications into types.

1 Surface Syntax

No layout expressions are used anymore, the syntax for types is extended as shown in the following Cogent surface grammar fragment for type expressions (taken from the Cogent documentation).

in green: correction (the unit type was not covered by the existing grammar, constructors in variants only had a single argument)
in red: additions for supporting layout information.

variable / field	v	::=	$\texttt{[a-z][A-Za-z0-9'\_]}^* \mid \texttt{\_[A-Za-z0-9'\_]}^+$
type constructor / tag	T	::=	$\texttt{[A-Z][A-Za-z0-9'\_]}^*$
bitsize	β	::=	$nat(\texttt{b}\vert\texttt{B})\ (\texttt{+}\ nat(\texttt{b}\vert\texttt{B}))^*$
bitlayout	l	::=	$\texttt{\#}^?(\texttt{sized}\vert\texttt{at})\ \beta$
endianness	ω	::=	$\texttt{endian}\ (\texttt{big}\vert\texttt{little})$
atomtype	τ_a	::=	$\texttt{(}(\tau(\texttt{,}\ \tau)^*)^?\texttt{)}$
		$\vert$	$\texttt{\{}v\ \texttt{:}\ \tau(\texttt{,}\ v\ \texttt{:}\ \tau)^*\texttt{\}}$
		$\vert$	$\texttt{<}(\tau\ \texttt{\vert})^?(nat\ \texttt{->})^?T\ \tau_{A2}^*\ (\texttt{\vert}\ (nat\ \texttt{->})^?T\ \tau_{A2}^*)^*\texttt{>}$
		$\vert$	v
		$\vert$	$T\ \omega^?$
type A2	τ_{A2}	::=	$\texttt{\#}\tau_a$
		$\vert$	$\tau_a\texttt{!}^?l^*$
type A1	τ_{A1}	::=	$T\ \tau_{A2}^*$
		$\vert$	$\tau_{A2}\ ((\texttt{take}\vert\texttt{put})\ (v\vert\texttt{(}(v(\texttt{,}\ v)^*\vert\texttt{..})\texttt{)}))^?$
mono-type	τ	::=	$\tau_{A1}\ (\texttt{->}\ \tau_{A1})^?$

Summary:

- For every atomtype a sequence of bit layouts can be specified.
- For variants the tag field and the mapping from tag values to variants can be specified.
- For primitive types an endianness can be specified.

2 General Semantics

A type now always specifies two views: the concrete view as a bitstring with access operations, and the abstract view as a functional data type (as before).

The concrete view is created from a given type expression by inferring default layout information at all places where no explicit layout information exists (see 6).

The abstract view is created from a given type expression by deleting all explicit layout information (shown in red in the grammar). This corresponds to a "layout erasure" type operator `nolayout` : $\tau \to \tau$.

For the refinement proof it must be proved that for every type used in the Cogent program the concrete view is a refinement (an encoding) of the abstract view.

Advantages of this approach:

- Simpler surface syntax, no redundancy for specifying the type structure in layout specs (such as for `{f1:U32,f2:U32} layout record {f1:4B at 0B, f2:4B at 6B}`, it now becomes `{f1:U32 sized 4B at 0B, f2:U32 sized 4B at 6B}`)
- No need for layout declarations and layout names.

- No need for a predicate / test that a layout matches a type.
- Layout polymorphism can be supported with the existing polymorphism mechanism using type variables.
- More flexibility by supporting partial layout specifications.
- Layout information can automatically be specified for other types than records, as it was planned but not yet implemented for the current approach.
- More expressiveness than for the current approach (e.g., layout for unboxed abstract types, layout for type variables).
- Support for endianness, including mixed endianness.

3 Full Support of Existing Approach

The proposal is semantically an extension of the existing approach: all specifications possible in the existing approach can be represented in the proposed approach (but see 9). Here I refer to the Dargent grammar as specified in `cogent/src/Cogent/Dargent/README.md`.

record with layout: A type expression `<type> layout <layout-expr>` of the form

```
{ f1: T1,...,fn: Tn} layout record { f1: L1 at O1,..., fn: Ln at On }
```

becomes

```
{ f1: TL1 at O1,...,fn: TLn at On}
```

where `TLi` is the combination of type `Ti` and layout `Li` as described in the following items.

primitive field with layout: For a field of primitive type `T` the current layout specification `L` must be a `<layout-size-expr>` of the form

```
S1 + ... + Sn
```

where all `Si` are a `<layout-size>` of the form `nat(b|B)`. The proposed combination `TL` is the type

```
T sized S1 + ... + Sn
```

unboxed record field with layout: For a field of unboxed record type `T = #{ f1:  T1,...,fn:  Tn}` the current layout specification `L` must have the form

```
record { f1: L1 at O1,..., fn: Ln at On }
```

The proposed combination `TL` is the type

```
#{ f1: TL1 at O1,...,fn: TLn at On}
```

where `TLi` is the combination of type `Ti` and layout `Li`.

boxed record field with layout: For a field of boxed record type `T = { f1:  T1,...,fn:  Tn}` the current layout specification `L` must have the form

```
pointer
```

The proposed combination `TL` is simply the type `T`. Since it is boxed the layout information that it is stored as a pointer is already implicitly present in the type.

variant field with layout: For a field of variant type `T = < C1 T1 |...| Cn Tn>` the current layout specification `L` must have the form

```
variant (LTag) { C1(v1): L1, ..., Cn(vn): Ln }
```

where `LTag` is the tag layout and `vi` are the tag values. The proposed combination `TL` is the type

```
< TLTag | v1 -> C1 TL1 |...| vn -> Cn TLn >
```

where TLi is the combination of type Ti and layout Li. TLTag is the combination of a numeric type sufficient to hold all tag values, and the layout LTag.

tuple field with layout: The intended layout for tuple types is not yet implemented, but the implementation accepts record layout specifications for them. For a field of tuple type T = (T1,...,Tn) the current layout specification L must have the form

```
record { p1: L1 at O1,..., pn: Ln at On }
```

The proposed combination TL is the type

```
(TL1 at O1, ..., TLn at On)
```

where TLi is the combination of type Ti and layout Li.

4 Layout Polymorphism

Since in the proposal every layout becomes a type, the normal polymorphism mechanism automatically applies to layouts as well. However, the current intention is to say that a layout variable can only be instantiated by layouts which match a given type. In the proposal this becomes the constraint, that the layout erasure of all instantiations must be equivalent to the given (layout-free) type.

I propose to support this by the following extension of the surface grammar:

$$\text{kind signature} \quad s \quad ::= \quad v\ (\texttt{:<}\ \kappa)^{?}\ (\texttt{:}\ll\ \tau)^{?}$$

Now in addition to the permissions, a type can be specified as constraint for every type variable. Its semantics is that for every instantiation of the type variable its layout erasure must be equivalent to the type specified as constraint.
Example:

```
type R = {a: U32, b: U16}
f: all r :<< R. r -> (r take b, U16)
f rec = let rec{b} = rec in (rec,b)
```

Now the polymorphic function f can be instantiated with all possible kinds of layouted forms of type R, such as

```
type R1 = {a: U32 sized 8B, b: U16 sized 4B at 8B}
... f[R1] ...
```

Note that :« corresponds to a very restricted form of subtyping. You can think of many possible extensions!

Note also that the take operator can be safely applied to type variable r since it is known that it must be instantiated by a layouted form of the record type R.

5 Layout Properties of a Type

The semantics of the proposal uses the following new type properties:

size The minimal size in bits required to represent all values of the type.

offset An offset in bits prepended to every value of the type.

usize The size of #T for a boxed type T.

uoffset The offset of #T for a boxed type T.

overlap A relation on types whether the value representations overlap.

Below, these properties are defined syntactically according to the surface grammar in 1. It should be straightforward to transfer the definitions to the core syntax.

```
size(T sized S) := S
size(T #sized S) := size(T)
size(T at S) := size(T) + S
```

`size(T #at S) := size(T)`
`size(T1 -> T2) := <function size>`
`size(T (take|put) ...)  := size(T)`
`size(C T1...Tn) := size(T)`   if `C T1...Tn` is a synonym for `T`
`size(C T1...Tn) := <pointer size>`   if `C T1...Tn` is abstract
`size(T!) := size(T)`
`size(#T) := usize(T)`
`size({ f1:  T1,...,fn:  Tn}) := <pointer size>`
`size(<T|v1->C1 T11 ... T1m`$_1$`|...|vn->Cn Tn1 ... Tnm`$_n$`>) :=` $\max($`size(T)`$, \max_{i=1}^{n} \max_{j=1}^{m_i}($`size(T`$ij$`)`$))$
`size((T1,...,Tn)) :=` $\max_{i=1}^{n}($`size(T`i`)`$)$
`size(()) := 0`
`size(t) := 0`
`size(T endian E) := size(T)`
`size(Unn) := nn`
`size(Bool) := 1`
`size(String) := <pointer size>`
`size(C) := size(T)`   if `C` is a synonym for `T`
`size(C) := <pointer size>`   if `C` is abstract
where `<function size>` is implementation dependent and may depend on the number of functions in the program, and `<pointer size>` depends on the target architecture.

`usize(T sized S) := usize(T)`
`usize(T #sized S) := S`
`usize(T at S) := usize(T)`
`usize(T #at S) := usize(T) + S`
`usize(T (take|put) ...)  := usize(T)`
`usize(C T1...Tn) := usize(T)`   if `C T1...Tn` is a synonym for `T`
`usize(T!) := usize(T)`
`usize({ f1:  T1,...,fn:  Tn}) :=` $\max_{i=1}^{n}($`size(T`i`)`$)$
`usize(T endian E) := usize(T)`
`usize(C) := usize(T)`   if `C` is a synonym for `T`
For all other types `T` the unbox-size is `usize(T) := 0`.

`offset(T at S) := S`
`offset(T (take|put) ...)  := offset(T)`
`offset(C T1...Tn) := offset(T)`   if `C T1...Tn` is a synonym for `T`
`offset(T!) := offset(T)`
`offset(#T) := uoffset(T)`
`offset(T endian E) := offset(T)`
`offset(C) := offset(T)`   if `C` is a synonym for `T`
For all other types `T` the offset is `offset(T) := 0`.

`uoffset(T #at S) := S`
`uoffset(T (take|put) ...)  := uoffset(T)`
`uoffset(C T1...Tn) := uoffset(T)`   if `C T1...Tn` is a synonym for `T`
`uoffset(T!) := uoffset(T)`
`uoffset(T endian E) := uoffset(T)`
`uoffset(C) := uoffset(T)`   if `C` is a synonym for `T`
For all other types `T` the unbox-offset is `offset(T) := 0`.

`overlap(T1,T2) := size(T1)` $>$ `offset(T2)` $\wedge$ `size(T2)` $>$ `offset(T1)`

For explanations see 8.

6 Default Layout

The default layout results from rules how to automatically add layout specifications in types. They correspond to a type operator `layout` : $\tau \rightarrow \tau$ which creates a "fully layouted" type from an arbitrary type. The default layout operator is parameterized by the default endianness.

The following rules are a simplified example and do not take into account the C alignment rules. Otherwise they try to reproduce the default layout of the current Cogent implementation.

- Types `U16`, `U32`, `U64`, if not followed by ω, are extended by `endian <default endianness>`.

- Type `Bool`, if not followed by `sized` β is extended to `Bool sized 4B`.
- Type `()`, if not followed by `sized` β is extended to `() sized 4B`.
- For a record type `{f1: T1,...,fn: Tn}` let $(\texttt{fp}_1, \texttt{Tp}_1), ..., (\texttt{fp}_m, \texttt{Tp}_m)$ be all fields with $\texttt{offset(Tp}_i\texttt{)} > 0$ (the "positioned fields"), and let $(\texttt{fu}_1, \texttt{Tu}_1), ..., (\texttt{fu}_k, \texttt{Tu}_k)$ be all fields with $\texttt{offset(Tp}_i\texttt{)} = 0$ in alphabetical order of the field names (the "unpositioned fields"). Let $o_{min} := \min_{i=1}^{m} \texttt{offset(Tp}_i\texttt{)}$ the minimal offset and $s_{max} := \max_{i=1}^{m} \texttt{size(Tp}_i\texttt{)}$ the maximal size of the positioned fields. Let $\forall 0 \leq i \leq k : s_i := \sum_{j=1}^{i} \texttt{size(Tu}_i\texttt{)}$ the size of the first i unpositioned fields together (with $s_0 := 0$). Let p be the maximal index with $0 \leq p \leq k \wedge s_p \leq o_{min}$, i.e., the maximal index so that the first p unpositioned fields together fit into the minimal offset of the positioned fields. Then for $1 \leq i \leq p$ extend $\texttt{Tu}_i$ to $\texttt{Tu}_i$ `at` s_{i-1} and for $p + 1 \leq i \leq k$ extend $\texttt{Tu}_i$ to $\texttt{Tu}_i$ `at` $s_{max} + s_{i-1} - s_p$.
- For a variant type with variants without tag value let v be the maximal tag value specified, or -1 if no variant has a tag value specified. Let `C1,...,Cn` be the constructor names used in the variants without tag value in alphabetical order. Let for $k \in \{1, ..., \texttt{n}\}$ be

  $$v_k := v + k$$

  Then v_k is the tag value to be used for the variant with constructor `C`k.
- For a variant type `<v1->C1 T1 |...| vn->Cn Tn>` without tag type specification use `U32` as tag type.
- For a variant type with tag type and all tag values extend the unpositioned types as described for record types, but group the unpositioned types in notation order according to the variant they belong to and order the groups alphabetically for the constructor names. If the tag type is unpositioned order it before all other unpositioned types.
- For a tuple type extend the unpositioned types as described for record types, but order the unpositioned types in notation order.

The sample default layout for a partially layouted record type puts the unpositioned fields contiguously in alphabetical order "around" the positioned fields: as much as possible before the positioned fields, the rest after them. If all fields are unpositioned this results in the same default layout as in the current Cogent implementation, if the rules would additionally respect the C alignment rules.

Note that a field which is explicitly positioned to `0b` is treated as unpositioned by the rules above, but when there is only one such field the rules will put it nevertheless at position 0, if there is enough space.

The default tag values for a partially tagged variant are counted from the largest explicit tag (or 0 if there is none) in alphabetical order of the constructor names.

If the tag type in a variant type is not positioned the rule above puts the tag field at position 0, if there is enough space.

Instead of a single default layout (or two, for big and little endian) you could define more named layout operators and apply them explicitly to types using a syntax such as `<type> layout <name>`. The difference to the current mechanism is that a layout operator is not type specific and can be applied to every possible type expression. You could even go one step further and introduce a syntax for specifying layout type operators.

7 Wellformedness

The following additional wellformedness rules apply to the extended type expressions.

- A type `T sized S` requires $\texttt{size(T)} \leq \texttt{S}$
- A type `T #sized S` requires $\texttt{usize(T)} \leq \texttt{S}$
- A record type `{f1: T1,...,fn: Tn}` requires $\forall 1 \leq i \neq j \leq n : \neg\texttt{overlap(T}i\texttt{,T}j\texttt{)}$
- A variant type `<T|v1->C1 T11 ... T1m`$_1$`|...|vn->Cn Tn1 ... Tnm`$_n$`>` requires

  $$\texttt{size(T)} \geq \lceil \log_2(1 + \max_{i=1}^{n} \texttt{v}i) \rceil \wedge$$

  $$\forall 1 \leq i \neq j \leq \texttt{n}\ \forall 1 \leq k \neq l \leq \texttt{m}_i : \texttt{v}i \neq \texttt{v}j \wedge \neg\texttt{overlap(T}ik\texttt{,T}il\texttt{)} \wedge \neg\texttt{overlap(T,T}ik\texttt{)}$$
- A tuple type `(T1,...,Tn)` requires $\forall 1 \leq i \neq j \leq n : \neg\texttt{overlap(T}i\texttt{,T}j\texttt{)}$

An arbitrary type `T` is wellformed, iff `layout(T)` is wellformed.

The default layout operator must have the property that for every layout-free type `T` the type `layout(T)` is wellformed. The example rules given in 6 have this property.

8 Detailed Semantics

A type `T sized S` represents the values of `T` at the beginning of a bitfield of size `S` (where the beginning in a memory word is the least significant bit). In other words, the values of `T sized S` are the values of `T` with a padding added *after* them.

A type `T at S` prepends a bitfield of size `S` to all values of `T`. In other words, the values of `T at S` are the values of `T` with a padding added *before* the values.

Examples: `U8 sized 4B` represents the numbers `0..255` in the least significant bits of a 32 bit word. `Bool sized 1b` represents boolean values in a single bit. `U8 sized 2B at 2B` represents the numbers `0..255` in the third byte of a 32 bit word (2 bytes padding before, 1 byte padding after). `U8 at 2B sized 4B` represents the same layout, the layout specifications do not commute!

Note that `size(U8 at 2B sized 4B) = 4B` whereas `size(U8 sized 2B at 2B) = 2B` which affects the `overlap` relation. A specification `T sized S` "burns in" any offset of `T` in the values of `T sized S`, the offset of `T sized S` is always 0.

For a boxed type `T` the specifications `T sized S` and `T at S` affect the pointer: `{a:U32,b:U16} sized 8B` represents the pointer to the structure in the lower half of a 64 bit word, if the pointer size is 32 bit on the target architecture. A similar size padding for the structure itself can be specified by `{a:U32,b:U16} #sized 8B` which inserts two bytes padding at the end of the structure.

Since `size(()) = 0` the unit type can be sized to zero: `() sized 0b`. Then it occupies no space in memory. When accessed its value can always be constructed. Since that is a better approach than in the current Cogent implementation it should become the default size for the unit type.

Abstract types can be sized. In

```
type A
type B = (A #sized 20B) sized 4B
```

type `B` represents a four byte pointer to a memory region of 20 byte size.

Type variables can be sized when they are used. In

```
type A a = { f1: U32, f2: a sized 12B, f3: a sized 8B }
```

the type variable `a` is used with two different sizes. Note that the rules in 7 imply, that the smallest size `S` used for a type variable imposes the implicit constraint on it that it may only be instantiated with types `T` for which `size(T)` $\leq$ `S`.

The fields in a record type are all placed at the same position. Explicit offsets (which are not "burned in") in the field types must be used to prevent field overlap. The default layout rules in 6 add corresponding offsets.

The components in a variant type are also all placed at the same position, together with the tag field. Again, explicit offsets in their types must be used to prevent overlap. The wellformedness rules in 7 allow components of different variants to overlap, although the default layout rules in 6 will not produce overlapping variants. Overlapping variants can be implemented with an int[] as common implementation in C, as it is used in the current Dargent implementation.

The endianness is only relevant for the types `U16`, `U32`, and `U64`. It specifies how the binary representation of their values is mapped to their memory region of 16, 32, or 64 bytes, respectively. For all other types it is ignored.

Note that you can explicitly specify an endianness for a single field which is different from the default endianness, for example, in a network package buffer.

9 Truncating Numeric Types

The wellformedness rules in 7 together with the `size` definition in 5 cause a type such as `U32 sized 3B` to be not wellformed. Such types result when translating a current layout specification such as `{a:U32,b:U16} layout record {a: 3B at 0B, b: 2B at 3B}` so that such layout specifications are not supported by the proposed approach.

However, such a field (as currently implemented) truncates values when they are stored and retrieved again. This is a bad idea for the Cogent semantics and the refinement proofs.

I propose the following solution: Extend Cogent to support as primitive types all types of the form `U<nat>`, such as `U24` or `U128`. Every such type `U`n is defined to be the set $\{0, .., 2^n - 1\}$ with arithmetic modulo 2^n.

Then the case above would be translated to `{a:U24 sized 3B, b:U16 at 3B}`. The main point now is that the difference is already visible for the layout erasure: the types `U32` and `U24` denote different value sets with different arithmetics. A type like `U32 sized 3B` is actually not a layout specification, it is a type conversion from `U32` to `U24` which must be done using a conversion function `U32 -> U24`.

Due to the wellformedness rules the proposed layout specification mechanism has the property that it never changes the type on the abstract level, every layout is an encoding of the unmodified value set specified by the layout erasure of the type.

Note that the proposal also allows types such as `U4 sized 2B`, which represents a hex digit stored in a half word.

The endianness is now relevant for all types Un with $n > 8$. Its general semantics is a permutation on the bytes needed for representing n bits (or even a permutation on the bits). It would be possible to extend the notation `(big|little)` to a syntax for denoting arbitrary permutations of $n \div 8$ or n elements, but perhaps that's overkill.

An even better approach for the syntax is to extend it for atomic types by the case

$$\text{atomtype} \quad \tau_a \quad ::= \quad \texttt{U}\ nat\ \omega^?$$

and remove the endianness from the atomic type constructor. Then the endianness can syntactically only be specified where it is relevant. To make the syntax backwards compatible the standard library could be extended by the definitions

```
type U8 = U 8
type U16 = U 16
type U32 = U 32
type U64 = U 64
```

10 Empty Layout

The type `U 0` has size 0 on the conrete level and corresponds to every single-valued type on the abstract level, in particular, to the unit type `()`. For a type with a single value the value need not be stored, it can always be constructed upon access, since it is unique. Thus, the unit type `()` can now be defined as a synonym for `U 0`.

The default layout rules specified in `default` will actually omit fields of type `U 0` (generally: of `size` 0) from layouted records, variants, and tuples.

The property of having size 0 can now be exploited for functions. Due to the default layout rules a function with a tuple type as parameter and/or result type will suppress all parameters and results of unlayouted unit type in the implementation. Moreover, if the parameter or result type is itself the unit type, nothing needs to be passed to or from the function in the C implementation. This makes it possible to implement a function

```
f: () -> T
```

by a parameterless C function

```
T f(void)
```

and a function

```
f: T -> ()
```

by a C function returning void

```
void f(T)
```

Note that in 5 the size of unboxed abstract types is defined to be 0: `size(#A)` := `usize(A)` := 0. This causes unlayouted unboxed abstract types to be omitted from the implementation. If you do not want this you have to size every unboxed abstract type as needed by using `#A #sized S`.

Empty layout can also be used for parameters which represent global state. With the proposed modifications it is possible to specify in Cogent that they should be omitted in the implementation. Example:

```
type Heap = U 0
type Rec = {a: U32, b: U16}
freeRec : (Rec,Heap) -> Heap
```

will result in the C function

```
void freeRec(Rec *p)
```

11 Enumeration Types

If the rule in 6 for inferring the default tag type for variant types is modified from always using `U32` to the type `U` n where $n := \lceil \log_2(1 + t_{max}) \rceil$ for the maximal tag value t_{max}, the default layout rules in 6 automatically produce the usual representation for enumeration types as (packed) numerical values, if the enumeration type is represented as a variant type with no layout specification.

For example, the type

```
< Red | Blue | Green >
```

will be extended by the default layout rule for adding tag values to

```
< 2 -> Red | 0 -> Blue | 1 -> Green >
```

and by the rule above to

```
< U 2 | 2 -> Red | 0 -> Blue | 1 -> Green >
```

which has `size` 2 and represents the tag values as two-bit entities. (It is not equivalent to `U 2` on the abstract level because value 3 is not in the value set!)

If we do not like the packed numerical value we can use

```
< U 2 | 2 -> Red | 0 -> Blue | 1 -> Green > sized U32
```

which stores the 2-bit values in 32-bit words. This also works on the abstract level:

```
< Red | Blue | Green > sized U32
```

has the same default layout.

In particular, the type `< True | False >` has default layout

```
< U 1 | 1 -> True | 0 -> False >
```

and has `size` 1 which corresponds to the representation of `Bool` in a single bit. This could even be used to define the type `Bool` based on the proposed `U` n types.