1 Gramatika

Policy jsou definovány jednoduchou gramatikou definující strukturu dat (rozšiřitelnou o strategie procházení, zarovnání, atd.).

⟨*policy*⟩ ::= '`policy<`' ⟨*shape*⟩ '`>`'

⟨*shape*⟩ ::= ⟨*tag*⟩
| ⟨*tag*⟩ '`, plus, `' ⟨*shape*⟩
| ⟨*tag*⟩ '`, cross, `' ⟨*shape*⟩
| ⟨*tag*⟩ '`, `' ⟨*shape*⟩ (same as the previous alternative: i.e. '`cross`' assumed)

⟨*tag*⟩ ::= ⟨*simple_tag*⟩
| ⟨*type_tag*⟩

⟨*simple_tag*⟩ ::= ⟨*array*⟩
| ⟨*tuple*⟩
| ⟨*vector*⟩

⟨*array*⟩ ::= '`array<`' ⟨*N*⟩ '`>`'

⟨*tuple*⟩ ::= '`tuple<`' ⟨*shape*⟩ '`>`'

⟨*vector*⟩ ::= '`vector`'

⟨*type_tag*⟩ ::= ⟨*type*⟩

⟨**simple_tag**⟩ definuje jednoduchou strukturu s běžnou sémantikou.

⟨**type_tag**⟩ definuje typ dat ve struktuře, funguje na základě definování typového "módu", kdy všechna následující data sdílejí daný typ, podobnou funkci má i ⟨**tuple**⟩, je-li použit pouze s ⟨**type_tag**⟩.

cross a **plus** jsou operátory kombinující jednotlivé tagy. **cross** má přednost a jeho význam je, že struktura definovaná levým operandem je složena z podstruktur definovaných pravým operandem. Význam **plus** je jednoduché řetězení operandů za sebou, uvnitř ⟨**type_tag**⟩ má **plus** operátor funkci čárky.

2 Iterace

Jednoduchá iterace je definovaná pro jednoduché policy a pro všechny policy, jež využívají pouze **cross** operátor. Iterace skrze tuple je prováděna paralelně, zatímco skrze array sekvenčně s intuitivní definicí procházení kombinace tohoto dle definicí výše (jako u matic vždy iterátor řádků, sloupců; iterátory vícerozměrným systémem + s fixovanou danou souřadnicí/souřadnicemi).

Procházení je definováno pro operátor **plus** pouze, jsou-li po sobě jdoucí policy kompatibilního typu (vysvětleno příkladem níže), pak je prováděno sekvenčně.

2.1 Příklad

Uvažujme následující příklad pro ukázání iterace na SoA a AoS:

```
using policy1 = tuple<float, int, char> cross array<N>; //SoA
using policy2 = array<N> cross tuple<float, int, char>; //AoS

auto i1 = policy1::begin(data);
auto i2 = policy2::begin(data);
```

potom platí (vedle očekávaných vlastností), že `i1->get<n>()` a `i2->get<n>()` jsou stejného typu (kde $n \in [0..2]$) a je definováno lineární uspořádání na sjednocení množin obou iterátorů (? i operace na nich) tak, že:

$$\texttt{policy1::begin(data)} == \texttt{policy2::begin(data)} \wedge \texttt{i1} == \texttt{i2} \rightarrow \texttt{++i1} == \texttt{++i2}$$

3 Funkce `at()`

Funkce `at()` je definována rekurzivním voláním pomocí template například následujícím způsobem (subject to change; zde jen velice zjednodušeně pro array), měla by korespondovat s implementací iterátorů:

```
template<unsigned length> struct array;
template<typename ...Ts> struct shape;
struct cross;

template<typename T, typename... Ts, unsigned n>
struct shape<T, array<n>, cross, Ts...> {
    using sub_shape = shape<T, Ts...>;
    static constexpr unsigned size = sub_shape::size * n;

    template<typename index, typename ...indices>
    static T &at(T *memory, index first, indices... rest) {
        return sub_shape::at(
            memory + (sub_shape::size * first), rest...);
    }
};

template<typename T, unsigned n>
struct shape<T, array<n>> {
    static constexpr unsigned size = n;

    template<typename index>
    static T &at(T *memory, index first) {
        return memory[first];
    }
};
```