Założenia tworzonego języka Rift

2023

Spis treści

1 Wstęp

Dokument ten ma na celu przedstawienie najważniejszych cech tworzonego języka. Nie zawiera on detali składni ani semantyki, a jedynie wysokopoziomowe spojrzenie na cechy, które staramy się uzyskać. Warto również zaznaczyć, że język jest na bardzo wczesnym etapie rozwoju i wiele funkcjonalności nie jest jeszcze precyzyjnie określona. Większość, jeśli nie wszystkie, opisane tu założenia mogą i będą podlegać dalszemu rozwojowi i modyfikacją. Obecnie obieramy pewne kierunki i istnieje już część implementacji, która zmierzają do interpretera, ale realizuje ona jedynie podzbiór języka i z natury jest wstępna.

2 Podstawowy cel

Naszym celem jest stworzenie języka, który połączy efektywność czasową i pamięciową z szybkim rozwojem oprogramowania oraz szerokimi możliwościami pisania generycznego kodu bibliotecznego.
Dodatkowo chcemy aby język był przystępny dla użytkownika. Choć wprowadzamy do języka funkcjonalności, które mogą częściowo temu przeczyć, to staramy się aby pewien podstawowy podzbiór języka, z którym programista będzie miał styczność przez większość czasu był możliwie prosty w użyciu.
Większość uwagi przypada na sam język, ale staramy się brać pod uwagę również inne ważne aspekty, takie jak na przykład, możliwość rozwoju ekosystemu języka, jakość narzędzi otaczających język, jakość komunikatów kompilatora czy możliwość wykonania w trybie REPL.

3 Kompilacja i wykonanie

Język z założenia jest kompilowany. Najprawdopodobniej użyjemy LLVM i generacja IR LLVM będzie ostatecznym krokiem kompilacji. Póki co skupiamy się jednak głównie na realizacji interpretera oraz dalszej kompilacji do bytecode-u maszyny wirtualnej. Zakładamy, że kompilacja do IR LLVM będzie następować bezpośrednio z bytecode-u maszyny, lub z ostatniej niskopoziomowej reprezentacji pośredniej używanej przez kompilator.

3.1 Interpreter

Obecnie staramy się stworzyć wstępny interpreter języka. Jest wysoce prawdopodobne, że ostatecznie nie będzie on używany, jednak mimo to pozwoli nam to na weryfikację tworzonego języka, oraz eksperymentowanie z nim w praktyce.

3.2 Maszyna wirtualna

Maszyna wirtualna tworzona jest przede wszystkim jako narzędzie dla programisty pozwalające analizować kod dynamicznie w zakresie znacznie przekraczającym możliwości analizy kodu natywnego.
Docelowo maszyna wirtualna będzie również prawdopodobnie używana jako interpreter języka i bardzo możliwe, że będzie bazą dla obliczeń czasu kompilacji.
Interpreter oraz możliwość wykonywania REPL pozwoli również na znacząco prostszy początek nauki języka.
Projektując maszynę wirtualną póki co skupiamy się na błędach pamięci, arytmetyki wskaźników, oraz precyzyjnego wglądu w pamięć programu.

4 Wysoka generyczność

4.1 Szablony

Szablony bazujemy głównie na szablonach z C++. Rozważamy wprowadzenie specjalizacji szablonów, jednak bardzo możliwe, że je porzucimy ze względu na inne mechanizmy zastępujące ich użycie.
Przykład kodu:

```
1    template{T: type, i: int}
2    fun foo(v: T) {
3      return v + i;
4    }
5
```

Szablonami mogą być klasy/struktury, zmienne/stałe, funkcje, inne szablony, oraz bloki kodu – w praktyce przestrzenie nazw. Przez blok kodu rozumiemy szablon, który działać na wielu symbolach, np:

```
template{T: type, i: int}
namespace QTemplate {
  struct Q {
    // ...
  }

  fun foo(v: Q) {
    return v + i;
  }
}

```

Realizacja szablonów póki co wciąż nie jest precyzyjne ustalona. W szczególności dwa istotne elementy, to:

- Kontekst wykonania lookup-u symbolów (miejsce deklaracji szablony / miejsce stworzenia instancji szablonu / miejsce użycia instancji szablonu)
- Reprezentacja szablonu gotowego do instancjonowania.

Pierwszy problem wydaje się mniej problematyczny, rozważany w nim również oddanie częściowej kontroli nad wyborem zachowania użytkownikowi.
Drugi problem jest trudniejszy i niesie istotne konsekwencje na detale zachowania szablonów. Najprostszym sposobem było by utrzymywanie drzewa składni szablonu i analizowanie go od nowa dla każdego wykonania. Takie podejście jednak nie pozwala na zgłaszanie błędów kompilacji szablonu już w trakcje jego deklaracji, oraz może mieć negatywny wpływ na czas kompilacji.

4.2 Argument Dependent Lookup (ADL) i overload symboli

Podobnie jak w języku C++ zakładamy, że może istnieć wiele symbolów (np funkcji) o tej samej nazwie.
W praktyce to zachowanie jest przydatne w przypadku funkcji oraz specjalizacji szablonów i nabiera znaczenia gdy jest połączone z ADL, czyli automatycznym wybraniem odpowiedniego symbolu na podstawie argumentów.

```
fun halve(i: i64) => i/2;
fun halve(i: BigInt) => i.fastHalve();
halve(12);

```

W podstawowym użyciu jest to jedynie zwiększenie wygody programisty, kosztem wprowadzenia nie jawnego zachowania.
Jednak w przypadku gdy taki „wielokrotny” symbol jest użyty na przykład wewnątrz szablonu, może on znacząco uprościć kod. Na przykład:

```
fun halve(i: i64) => i/2;
fun halve(i: BigInt) => i.fastHalve();

template{T: type}
fun withAdl() {
  // ...
  t: T;
  // ...
  halve(t);
  // ...
}

template{T: type}
fun withoutAdl() {
  // ...
  t: T;
  // ...
  compile_if (T == i64) {
    t / 2;
  }
```

```
    else compile_if(T == BigInt) {
      t.fastHalve();
    }
    // ...
  }

```

Funkcja `withoutAdl` mogła by również posiadać odpowiednie specjalizacje, jednak wtedy bylibyśmy zmuszeni duplikować kod kryjący sie pod `//...`.

4.3 Dynamiczne typowanie czasu kompilacji

W Rift "typ jest typem" i jego zachowanie nie różni się w żaden sposób od innych typów języka. Oczywiście język jest statycznie typowany, więc większość operacji na typach musi być wykonanych w czasie kompilacji. W praktyce znacząco upraszcza i uspójnia to język. Przykładowo szablon nie musi rozróżniać parametrów typowych i nie-typowych, tak jak robi to C++.
Otwiera to również wiele możliwości, możemy na przykład, tworzyć funkcje przyjmujące i zwracające typy:

```
  // trivial example
  const fun makeType(t: type) -> q:type {
    return std.list{t};
  }

  var interesting_variable: makeType(i32);

```

Upraszcza to wiele operacji na typach. Przykładowo porównanie czy dwa typu są sobie równe co w C++ realizuje `std::is_same<T, Q>` opierający się na szablonach, specjalizacji szablonów oraz ADL. W Rift możemy wykonać przez wbudowaną operację `T == Q`, co jest pełnoprawnym wyrażeniem języka.
Możemy więc napisać na przykład:

```
  template{T, Q: type}
  namespace some_generic_code {
    const v: bool = (T != Q) and T.size > 100;
    // ...
  }

```

4.4 Marka i proceduralna generacja kodu

Makra zachowują się istotnie inaczej niż preprocessor znane z C++. Są to makra syntaktyczne, które występują na przykład w językach Rust, Raku, Nim, Elixir.
Z technicznego punktu widzenia makra obecnie traktujemy jak funkcje czasu kompilacji, które są w stanie zwracać "fragmenty kodu". W praktyce będą one zwracały albo ciąg tokenów języka, albo fragmenty drzewa składniowego. Mogą one przyjmować dowolne wartości i fragmenty kodu. Mają również dostęp do "kontekstu" programu z którego zostały wywołane.
Zastosowania mark dzielimy na trzy kategorie:

- Użycie makra, do prostszego zapisania powtarzalnego fragmentu kodu (powszechne użycie `#define`)
- Proceduralną/automatyczną generacją kodu.
- Specyficzną modyfikację napisanego większego fragmentu kodu (na przykład dodanie metody do klasy).

Rozważamy stworzenie specjalnych rodzajów mark lepiej przystawnych do konkretnych zadań. Dodatkowo jest prawdopodobne, że będzie koniczne będzie wprowadzenie silniejszych ograniczeń na makra, niż na ogólne funkcje czasu kompilacji, z dwóch powodów. Po pierwsze makra mogą "gryźć" się z innymi cechami języka (całkiem dużym wyzwaniem, jest na przykład połączenie ich z kompilacją nie działającą od góry do dołu, oraz wyrażeniami `using, alias`). Po drugie czym bardziej ogólne makra tym bardziej problematyczne może okazać się podpowiadania programiście przez IDE.

```
  macro makeMethod() {
    return ${
      fn moo() => this.x;
```

```
    }
  }

  struct T {
    expand makeMethod();
  }

```

4.5 Atrybuty

Atrybuty są sposobem na wprowadzenie do programu dodatkowych informacji, modyfikujących jego zachowanie, dla których nie chcemy tworzyć nowej składni.
Każdy atrybut dotyczy konkretnego symbolu:

```
@example_attribute(attribute arguments)
fun someFunction() {...}

```

Zakładamy istnieje wielu atrybutów wbudowanych, i planujemy również wprowadzić atrybuty definiowane przez użytkownika.
Przykładowymi zastosowaniami atrybutów są:

- Automatyczna generacja kodu, na przykład `@memoize` zamieniająca czystą funkcje, w funkcję zapamiętującą wartości.
- Atrybuty typu `@compile_if(bool)` wpływające na sam fakt kompilacji symbolu.
- Ustalenie priorytetu wyboru symbolu przez ADL.
- Uruchamianie makra na symbolu, przed jego analizą.
- Zezwalanie na niestandardowe zachowania, np `@shadow` pozwalający wprowadzić zmienną ukrywającą inną zmienną.
- Atrybuty pozwalające budować inne, takie jak: dodawanie dziedziczenia do klasy,

4.6 Kompilacja warunkowa

Przez kompilacje zależną, rozumiemy przede wszystkim możliwość ominięcia kompilacji jakiegoś symbolu (lub nie uwzględniania go w lookup-ie symbolów), wprowadzenie warunków użycia danego symbolu (na przykład dotyczących parametrów szablonu), wprowadzania fragmentów kodu jedynie przy spełnieniu odpowiednich warunków.
Rozważamy tutaj kilka mechanizmów:

- Blok `compile_if + else`, który może zostać użyty za równo w kodzie statycznym. Jest to analogia mechanizmu `static_if` z języka D. Podobne zadania spełnia również `#if` z C++.
- Analogia mechanizmu `requiers / std::enable_if` z C++. Mechanizm ten opierał by się na słowie kluczowym `requiers`, lub atrybucie `@compile_if/@enable_if`.
- Blok `compile_try`, który kompiluje dany fragment kodu, jedynie jeśli nie wystąpiły w nim błędy lookup-u symbolu lub zgodności typów. Pozwala to w bardzo przejrzysty sposób pisać w kodzie schematy takie jak customization point objects.
- Mechanizm analogiczny do konceptów znanych z C++. Widzimy tutaj dwie możliwości: koncepty działające jako duck-typing (C++) oraz trait-y działające nominalnie, czyli każdy trait musi być jawnie przypisany do typu (Rust). Oba podejścia mają swoje zalety, a mieszanie ich może okazać się problematyczne. Warto również zauważyć, ze koncepty są analogicznym mechanizmem do value-constraint dla typu **type**.

4.7 Refleksje

Przez refleksje zakładamy dodatkowe informacje o programie, a w szczególności o typach i symbolach dostępne zarówno w czasie kompilacji jak i w czasie wykonania.
Zastosowania i funkcjonalności refleksji częściowo pokrywają się z markami, dlatego bardzo możliwe, że te dwie funkcjonalności zostaną częściowo lub w pełni połączone w jedną całość.
Obecnie nie posiadamy konkretnych założeń, ani modelu refleksji, ale chcemy aby mogły one realizować następujące zadania poprzez funkcje wbudowane, lub implementacje użytkownika bazujące na prostszych funkcjonalnościach:

- Możliwość automatycznego stworzenia mapy z pewnego zbioru
- Możliwość automatycznego uzyskania nazwy symbolu w postaci napisu.
- Możliwość iteracji po "podsymbolach" symbolu, możliwość iteracji po polach typu, oraz jego innych elementach.
- Możliwość pozyskania informacji o typie dowolnej wartości (w tym jego napisową reprezentacje). W szczególności pozyskiwanie informacji o typie.
- Możliwość automatycznego mapowania wartości typów enumeracyjnych i na ich reprezentację napisową.

4.8 Obliczenia czasu kompilacji

Zakładamy istnieje możliwie szerokiego podzbioru języka będącego możliwego do wykonania w czasie kompilacji.
Poza oczywistą możliwością obliczania wartości stałych w czasie kompilacji, silne obliczenia czasu kompilacji otwierają również wiele dodatkowych możliwości:

- W wielu miejscach gdzie język oczekuje wartości **true**/**false** (na przykład block `compile_if`) w czasie kompilacji, nie musimy wprowadzać do języka dodatkowych specjalnych wyrażeń. Możemy oczekiwać zwyczajnego wyrażenia o typie `bool`, które wyliczamy w trakcje kompilacji.
- Zdolność do pisania arbitralnej logiki pozwala rozwiązać wiele problemów za pomocą prostej funkcji lub prostego wyrażenia czasu kompilacji. Przykładem tego typu problemów jest na przykład biblioteka "type_traits", z języka C++.

Elementy, które prawdopodobnie nie będą dostępne, to:

- Operacje na zmiennych globalnych. Choć jest to technicznie wykonalne, to wprowadzało by trudne do zarządzania przez programistę zachowania. Rozważamy wprowadzenie zmiennych globalnych jedynie czasu kompilacji.
- W pełni dowolne operacje na pamięci, surowe wskaźniki. Problematyka tych operacji leży w ich charakterze. Zachowują się one zależnie od detali reprezentacji zmiennych w pamięci. Powoduje to, że są one wrażliwe, na architekturę komputera oraz modyfikacje w kodzie wykonującym obliczenia czasu kompilacji. Kompilacja więc mogła by drastycznie zmieniać się między komputerami lub wersjami kompilatora co jest zachowaniem nieporządnym.

Pewną niepewnością jest wpływ obliczeń czasu kompilacji na czas działania kompilatora. Możliwe, że niezbędne będą dodatkowych ograniczeń w celu otworzenia możliwości przyspieszenia ich ewaluacji.

5 System typów

5.1 Semantyka wartości, kategorie wartości, co dokładnie jest częścią typu

todo...
Częścią typu jest:

- Pola typu, w tym również pola reprezentujące dziedziczenie.
- Operacje analogiczne do konstruktorów oraz destruktorów z C++, między innymi: operacja przeniesienia, kopii, domyślnego stworzenia, usunięcia.

- Część dodatkowych operacji takich jak: przyrównywanie, porównywanie, hash-owanie. Operacje te są częścią typu ze względu na powszechność ich występowania, oraz konieczność użycia ich w instancjonowaniu szablonów.
- Metody wirtualne.

Częścią typu nie są metody. Traktujemy je jako "syntatic sugar" na zwyczajne funkcje.

5.2 Abstrakcja wskaźników

Domyślnie wskaźniki nie są częścią języka. Są zastępowane przez typy:

- Referencyjne – "wskaźnik na obiekt"
- Tablicowe – "wskaźnik tablicę obiektów"
- Funkcyjne – "wskaźnik funkcję"

Pozwala to zwiększyć bezpieczeństwo kodu, oraz nie pozwalać na niespodziewane zachowania.

5.3 Wbudowane typy algebraiczne oraz pattern matching

Tworzony system typów posiada wbudowanie typy:

- `tuple`
- `variant` (tagged union)
- `optional`

Dodatkowo planujemy wprowadzić typy analogiczne do typów enumeracyjnych znanych z języka Rust. Podobnie jak istnieją struktury oraz typ `tuple`, które pełnią podobną funkcje, jednak w inny sposób, tak typy enumeracyjne z Rust-a są analogicznie powiązane z typem `variant`.
Wbudowanie tych typów do języka pozwala na wprowadzanie specjalnej składni oraz semantyki, upraszczającej korzystanie z nich.
Dodatkowo możliwa jest realizacja pattern matching-u znanego z języków funkcyjnych lub języka Rust.

5.4 Wbudowane typy funkcyjne oraz funkcje pure

Tworzony system typów posiada trzy rodzaje wbudowanych typów funkcyjnych:

- Obiekt funkcyjny – najogólniejszy typ, mogący przechowywać dowolną funkcje
- Wskaźnik na funkcję – typ reprezentujący wskaźnik na funkcję znany z C++. Istnieje tego typu ma charakter czysto wydajnościowy.
- Typ symbolu funkcyjnego – typ reprezentujący funkcję napisaną w kodzie jako **fun** `foo()`.

Z punktu widzenia użytkownika podział ten jest uproszczony, oraz zachodzą automatyczne konwersje wskaźnik funkcyjny -> obiekt funkcyjny, symbol funkcyjny -> wskaźnik funkcyjny.

```
1    fun foo(int) -> int {...}
2
3    var v: int->int = foo; // ok, function object
4    var v: FuncPtr{int->int} = foo; // ok, function pointer
5
```

Rozważamy również wprowadzenie automatycznej dedukcji, czy typ `T->Q`, może być "optymalizowany" do typu wskaźnikowego.
Dodatkowo każdy z tych typów może posiadać znacznik **pure**, będący częścią typu. Określa on, że funkcja nie może powodować efektów ubocznych.
Wprowadzenie typów funkcyjnych motywowane jest paroma powodami:

- Implementacje biblioteczne (na przykład `std::function`) są niewygodne w użyciu, oraz nieefektywne czasowo.
- Wbudowane typy funkcyjne pozwalają na dobrą interakcję z wyrażeniami lambda/closure.
- Wbudowane typy funkcyjne pozwalają wprowadzać dedykowaną semantykę typu. Przykładowo istnieje dzięki temu możliwość wprowadzenia efektywnej częściowej aplikacji.
- Wbudowane typy funkcyjne mogą zapewnić szybszą kompilację, oraz błędy kompilacji lepszej jakości.

5.5 Klasy

W klasach poza prostymi polami oraz metodami, skompilowaną częścią jest realizacja dziedziczenia. Obecnie dziedziczenie bazujemy w pełnie na języku C++. Wprowadzamy wielodziedziczenie, funkcje wirtualne, dziedziczenie wirtualne. Planujemy również wprowadzić statyczne pola wirtualne. W praktyce zachowują się ona tak samo, jak wskaźniki na funkcje wirtualne. Zamiast wskaźnika na funkcje trzymamy w tablicy wirtualnej dane zmiennej, lub wskaźnik na jej dane. Pierwsze podejścia generuje jedno mniej odwołanie do pamięci przy czytaniu zmiennej, jednak posiada również albo musimy trzymać co najmniej dwie tablice wirtualne, albo tablica wirtualna musi być pamięci, po której możemy pisać.
Rozwiązują one poniższy problem:

```
struct T {
  // version 1:
  // works well, but increase size of T.
  kind: Kind = Kind.T;

  // version 2:
  // works well, but overhead of calling virtual function tends to be high
  virutal getKind { return Kind.T; }
}

struct Q: public T {
  Q() {
    T();
    kind = Kind.Q;
  }

  overwrite getKind { return Kind.T; }
}

```

Zamiast tego możemy napisać:

```
struct T {
  // single for all instances of T
  static virtual kind: Kind = Kind.T;
}

struct Q: public T {
  // single for all instances of Q
  static override kind: Kind = Kind.Q;
}

var q: Q;
q.kind; // requires only single lookup in virtual table.

```

Rozważamy jednak odejścia od tego podejścia, na rzecz podejścia bliższego temu, który realizuje Rust. Dodatkowo rozważamy wprowadzenie klas różnych typów, które ograniczają konieczność ręcznego definiowania zachować. Znanym przykładem są tak zwane `data class` występujące w językach takich jak Kotlin, Dart, Python.

5.6 Ograniczenie rozróżnienia typów wbudowanych oraz definiowanych przez użytkownika

Wiele języków wprowadza silne rozróżnienie typów definiowanych przez użytkownika oraz wbudowanych. Chcemy od tego podejścia odejść, a w szczególności przybliżyć typy wbudowane do tych definiowanych. Oznacza, to dla nas:

- Z punktu widzenia użytkownika typy wbudowane posiadają identyczną semantykę wartości, co każdy inny typ.

- Typy wbudowane mogą posiadać metody, metody statyczne, inne pola wewnątrz dostępne jako **type**`.field`.
- Po typach wbudowanych możemy dziedziczyć.
- Możemy definiować nowe metody typów wbudowanych.

6 Model pamięci oraz współbieżność

Współbieżność jest elementem, który póki co nie jest jeszcze dokładnie przemyślany. Wiemy, że chcielibyśmy połączyć możliwość wielozadaniowość bez wywłaszczania (cooperative multitasking), opartą na przykład na "fibers" z współbieżnością pozwalająca na wykonanie równolegle, opartą o wątki systemowe.
Pierwsza funkcjonalność pozwala pisać nie blokujące programy zawierające operacje, na które program musi poczekać. Znacząco ułatwiają przy tym problemy wynikające z programowania wielowątkowego. Druga funkcjonalność jest kluczowa dla prędkości działania wielu programów/algorytmów.

7 Odejście od kompilacji od góry do dołu

Podobnie jak język Rust, chcemy aby programista nie musiał pisać "forward deklaracji" aby użyć funkcji przed jej definicją. Każdy symbol statyczny powinien byś dostępny w "całym" programie o ile zezwalają na to reguły widoczności. Nie chcemy również, aby konieczna była jakakolwiek analogia plików nagłówkowych znanych z C/C++. Zakładamy, że każdy symbol jest napisany dokładnie raz i w jednym miejscu.

Spełnienie tych założeń jest dodatkowo utrudnione ponieważ chcemy posiadać system modułów, możliwość kompilacji inkrementalnej, przestrzenie nazw, overload-y symboli, obliczenia czasu kompilacji, marka syntaktyczne oraz wyrażenia **using**/`alias` (inspirowane C++ oraz Scalą).

Przykładowo wynik lookup-u symbolu może więc być zależy, od wartości obliczonej już w czasie kompilacji:

```
1    @compile_if(calculateSomePredicate1())
2    fun foo() {...}
3
4    @compile_if(calculateSomePredicate2())
5    fun foo() {...}
6
7    foo();
8
```

Dodatkowo na wynik może również wpłynąć kod generowany przez makro. Musimy również posiadać gwarancję, że znalezione będą wszystko symbole w przypadku overload-u.

Oryginalna koncepcja rozwiązująca te problemy opierała się na kilku przejściach po drzewie składniowym, z których każde realizowało by po kolei odpowiednio wpisanie symboli to tablicy symboli, obliczenie **using**/`alias`, obliczenie makr, wykonanie lookup-u użytych w programie nazw. Takie podejście okazało się jednak silnie ograniczające język.

Znacząco elastyczniejszym podejściem okazało się wykonywanie tych wszystkich kroków w pewnym sensie jednocześnie – nie każdy symbol jest w tej samej fazie analizy w trakcje działania programu. Algorytm realizujący ten proces jest obecnie implementowany, i pozwala na bardzo szeroki zakres możliwości z zaskakująco spójny sposób.

7.1 using oraz alias

Wyrażenia te sprowadzają się do dwóch przypadków:

- `alias name = some.longer.name.with.dots;`
- **using** `some.symbols.somewhare.*;`

Alias pozwala nam na używanie `name` zamiast `some.longer.name.with.dots` i sam w sobie jest traktowany jak symbol. Using pozwala na używanie wszystkich symboli zawartych wewnątrz `some.symbols.somewhare.*` i jest traktowany jako specjalny symbol bez nazwy (z pustą nazwą). Oznacza to również, że jeśli przestrzeń nazw `N` zawiera `alias x = y` to możemy na przykład wykonać `N.x`.

Dodatkowo wyrażenia te mogą również dotyczyć zmiennych:

```
struct T {
  var a: i32;
}

var t: T;
alias x = t.a;
x += 2; // ok

struct Q {
  var b: T;
  alias a = b.a;
}

var q: Q;
q.a; // ok

```

8 Językowe wsparcie testowania

8.1 Brak niejednoznacznej pamięci

Niejednoznaczna interpretowana pamięć występuje w języku C/C++ w postaci typów `union`. Pomijając fakt braku gwarancji poprawnego odwołania do pamięci, taka funkcjonalność dodatkowo znacząco utrudnia dynamiczną analizę kodu.

8.2 Value constraint, invariants

Value constraint jest możliwością tworzenia typów, które posiadają niezmiennik sprawdzany w czasie wykonania.

```
// Q is type
const T: type = Q requiers (somePurePredicate(self));

var t: T;
t = value; // somePurePredicate is checked

```

Rozważamy również możliwość stworzenia statycznej analizy kodu, która mogła by dowodzić zachodzenie, lub niepoprawność niektórych warunków (na przykład nierówności liczb całkowitych).

8.3 Warunki In/Out

Warunki In/Out są w praktyce lukrem syntaktycznym na asercje na początku oraz na końcu funkcji.

```
fun trivialExample(x: i32) -> r: i32
in (x > 10)
out (r < 0) {
  return 10 - x;
}

```

Pozwala to zwiększyć czytelność, i ograniczeniach możliwości popełnienia błędu przez programistę – dużo "łatwiej" jest nieświadomie zmodyfikować niezbędną asercję, niż warunek `in/out`.
Otwiera to również możliwości statycznej analizie kodu. W niektórych przypadkach względnie łatwo jest wykryć, że warunek nie musi być przez daną funkcję spełniony, lub że argumenty nie spełniają warunku `in`.

8.4 Słowa kluczowe test, debug

Jest to pewnego rodzaju detal, który jednak uważamy, za istotny.
Funkcjonalność ta jest szczególnym przypadkiem kompilacji warunkowej, i można traktować ją jako lukier syntaktyczny.

Przykład działania (analogicznie `debug`):

```
    test {
      // runs only if target = test
    }

    test (...) {
      // runs only if target = test, and additional condition is satisfied
    }

    // runs only if target = test
    test assert(...);

    // constraint only if target = test
    const T: type = i32 test requiers (self > 100);

    const T: type = i32 test(...) requiers (self > 100);

    fun trivialExample(x: i32) -> r: i32
    test in (x > 10) { // run-time checked only if target = test
      return 10 - x;
    }

```

Jest to bardzo powszechny przypadek kompilacji zależnej. Pozwala on pozostawiać w kodzie dodatkowe weryfikacje poprawności, których nie chcemy posiadać w ostatecznym programie (na przykład ze względu na wydajność). Planujemy wprowadzić "poziomy" testowania/debug-u programu, oraz możliwość dokładniejszej kontroli, które fragmenty są uruchamianie.
Dodatkowo możliwe jest wprowadzanie innych analogicznych mechanizmów, przykładowo dotyczących odporności programu na ataki.

9 System kompilacji, system modułów

System modułów jest częścią języka i definiuje dokładnie zachowanie wyrażeń `import`, podziału plików na paczki, widoczność paczek. Określa też jakie dokładnie symbole i w jaki sposób są widoczne między plikami. W aktualnej koncepcji system ten będzie bazowany na systemie języka Python, jednak wydaje się, że zmiany w systemie nie powinny powodować problemów w trakcje rozwoju języka. System modułów ostatecznie na podstawie wyrażeń `import` oraz struktury plików przekazuje kompilatorowi informacje o powiązaniach pojedynczych plików. O ile sam interface tych powiązań nie ulegnie znaczącej zmianie, to zmiany w systemie modułów będą przezroczyste dla reszty kompilatora.
System kompilacji jest natomiast oddzielnym systemem, który automatyzuje proces kompilacji i zarządza zależnościami. Uważamy, że system kompilacji powinien być dla języka dedykowany i zintegrowany z kompilatorem. W praktyce widzimy wysoką specjalizację niektórych systemów pod konkretne języki (na przykład Cmake - C/C++, Maven - Java). Takie podejście może znacząco ograniczyć konieczność konfiguracji i ułatwić pracę z nim.
Przykładowo domyślna konfiguracja projektu może być generowana automatycznie, nie jest konieczne wypisywanie plików źródłowych oraz podziału na moduły. Dodatkowe flagi kompilacji mogą być przekazywane przy poleceniu kompilacji.
Taki system współpracując z kompilatorem, może również automatycznie generować analogię plików nagłówkowych i/lub pre-kompilowanych plików, które mogą znacząco przyspieszyć kompilacje inkrementalną. Pre-kompilowany pliki nie muszą być jedynie plikami obiektowymi, ale mogą również być plikami, na których wykonana została już pewna liczba faz kompilacji.
Takie podejście pozwala również stworzyć język konfiguracji, zawierający cechy wspólne z językiem Rift, co upraszcza zrozumienie go, i ułatwia naukę. Hipotetycznie konfiguracja kompilacji mogła by być nawet plikiem źródłowym Rift, analizowanym w specjalnym kontekście.
Dodatkowo uspójnia to ekosystem języka. Projekty mają podobne struktury i korzystają z jednego narzędzia, które nie wymaga dodatkowej instalacji.

10 Inne istotne cechy języka

10.1 Implicit context

Implicit context jest ideą w pełni zapożyczoną z języka Jai. Jai również wprowadził takie pojęcie. Jest to globalna (inna dla każdego wątku) struktura danych, zachowująca się jak stos, który trzyma wartości typu `Context`.
Jai wprowadza go przede wszystkim, aby kontrolować alokatory używane w programie. `Context` zawiera wskaźnik na alakator, a w trakcje działania program może wrzucać (i zdejmować) nowe wartości z innym alakatorem. Domyślna alokacja pamięci, korzysta z alokatora znajdującego się na szczycie stosu.
Samo podejście zarządzania alokatorami wciąż nie jest pewne, jednak Implicit context wprowadza ciekawe podejście zarządzania globalnym stanem programu, który może być wykorzystywany również do innych celów. Przykładowo możemy utrzymywać w nim struktury odpowiedzialne za wykonywanie I/O i log-owanie komunikatów.

10.2 Rozszerzony control-flow

Planujemy wprowadzić szerszy control-flow niż znany z języka C++. W szczególności chcemy wprowadzić:

- Nazwane bloki kodu, pozwalające na przykład wykonać wyrażenie **break** z pętli zewnętrznej.
- Wprowadzenie dodatkowych wyrażeń analogicznych do **break**/**continue**, w szczególności **restart**, **redo**, `last`, które pozwalają odpowiednio: uruchomić blok kodu od początku, wrócić na początek aktualnego obrotu pętli, oznaczyć obrót pętli jako ostatni.
- Wyrażenie **defer**, inspirowane językiem GO, jednak działające inaczej. Wyrażenie **defer** pozwala dodać wyrażenie, które będzie wykonany na końcu danego bloku/obrotu pętli. W szczególności wyrażenie **defer** jest w pełni statyczne i nie utrzymuje wyrażeń do wykonania dynamicznie.

Przykład:

```
for Outer(...) {
  for Inner(...) {
    break Outer;
  }
}

if (...) {
  var file := open("data.txt");

  // it will run at the end of `if` block
  defer file.close();

  // ...
}

```

10.3 Gramatyka wyrażeń

Poza pewnymi drobnymi detalami, chcemy wprowadzić możliwość definiowania dowolnych operatorów analogicznie jak Haskell i Ocaml.
Problemem jest określenia gramatyki operatorów dla każdego wyrażenia w języku (określanie priorytetu, arności, associativity, ...).
Język Ocaml rozwiązuje ten problem, przed definiowanie tych wartości jedynie na podstawie napisu reprezentującego operator. Przykładowo `+++` zawsze będzie infixowy, wiążący od prawej do lewej o priorytecie tym samym co `+`.
Alternatywnym rozwiązaniem, stosowanym przez język Haskell, jest definiowanie tych własności przez użytkownika. Rozwiązanie stosowane przez Haskell jest przez nas preferowane, jednak widzimy pewne potencjalne problemy:

- Patrząc na wyrażenie, programista musi dodatkowo zwracać uwagę w jakiej gramatyce jest ono zapisane. Może utrudniać to programowanie i rozumienie kodu szczególnie gdy nie jesteśmy z nim zaznajomieni (czytając na przykład kod w internecie).

- Możliwe jest udostępnianie na zewnątrz tego samego operatora o innej gramatyce przez różne moduły. W praktyce możemy wtedy importować tylko jeden z nich. Problem ten można rozwiązywać przykładowo przez wprowadzanie składni wołania operatorów niezależnej od gramatyki, jednak traci to wtedy zalety definiowania jej.

10.4 Wsparcie paradygmatu imperatywnego

Tworzony język wprowadza elementy z innych paradygmatów, ale niewątpliwie ma imperatywny charakter. Ze względu na to, chcemy, aby imperatywna część języka również była "zadbana".

Dwa szczególnie istotnie problematycznymi aspektami programowania imperatywnego są zmienne globalne oraz referencje pozwalające na modyfikację wartości.

Wiele języków usuwa zmienne globalne. Uważamy, że nie jest to konieczne, a zmienne globalne posiadają swoje zastosowania. Aktualnie rozważamy trzy mechanizmy poprawiające zmienne globalne:

- Wymuszanie jawnego zapisania w każdej funkcji z jakich zmiennych globalnych bezpośrednio korzysta
- Możliwość ukrywania widoczności zmiennych globalnych. Przykładowo zmienna globalna wewnątrz przestrzeni nazw (lub modułu/pliku) może być zadeklarowana jako prywatna, co zabrania używania jej poza przestrzenią nazw.
- Detekcja "Static Initialization Order Fiasco". System modułów oraz brak niezależnej kompilacji każdego pliku pozwala wykrywać cykliczne zależności inicjalizacji oraz sortować je topologicznie. Wciąż jednak pozostaje problem ograniczeń używanych narzędzi. W przypadku pre-kompilacji do plików obiektowych generowanych przez LLVM, zapewnienie odpowiedniej kolejności w procesie konsolidacji wymagało by napisanie dedykowanego programu konsolidującego. Można go jednak prosto rozwiązać poprzez napisanie prostego algorytmu odpowiadającego za inicjalizacje symboli – w praktyce jest to po prostu detekcja cykli w grafie skierowanym.

Problem dotyczący referencji jest bardziej złożony. Poza samymi błędami pamięci mogą one wprowadzać również ciężkie do zarządzania zachowania do kodu. Poza aspektami składniowymi takimi jak jawne tworzenie referencji na wartość (inaczej niż C++), rozważamy również rozszerzenie typów referencyjnych na kilka rodzajów udostępniających odpowiednie analogiczne konwersje, które automatyzowały by część zarządzania pamięcią (przykładowo podobne do `std::uniqe_ptr`, `std::shared_ptr` z C++).
Możliwym rozwiązaniem części problemów jest również wymuszenie dodatkowych założeń o referencjach oraz semantyce przenoszenia wartości i napisanie tak zwanego borrow-checker-a, analogicznego jak w języku Rust.

10.5 Blok unsafe, czyste wskaźniki, cast-y pamięci

"Dowolne" operacje na pamięci pozostawiamy jako możliwe wewnątrz bloku `unsafe`. Możliwe w nim są operacje na surowych wskaźnikach, oraz zmiana typu wskaźnika. Jest jednak wciąż do pewnego stopnia ograniczone. Przykładowo możemy wciąż surowy wskaźnik na zmienną, ale nie ma funkcję.
Prawdopodobnie wprowadzimy założeniem bloku unsafe, które gwarantuje brak wycieku surowych wskaźników poza dany blok. Tj. nie jest możliwe istnieje referencji o niepoprawnej wartości poza blokiem unsafe. Możliwe, jest również wymuszenie silniejszego założenia, jakim jest brak możliwości nadpisania wartości o typie referencyjnym wewnątrz bloku unsafe. Jeśli konieczne jest wymiana wskaźników między blokami unsafe może odbywać się to przez typ trzymający parę `reference, offset`.
Rozważamy również wprowadzanie dodatkowych elementów legalnych wewnątrz bloków `unsafe` (lub innych analogicznych). Przykładem takich zachować są: naruszenie założeń typów referencyjnych (o ile takie założenia będą wprowadzone) lub dostęp do symboli nie widocznych w danym miejscu programu.

10.6 Widoczność symboli

Widoczność jest własnością każdego symbolu i określa gdzie i w jaki sposób symbol jest widoczny poza zakresem kodu, do którego należy. Jest to statyczna własność symbolu, przykładowo:

```
1    struct T {
2      public var a: i32;
3    }
4    T.a; // a is visible
5
```

symbol `a` jest widoczny wewnątrz symbolu `T`. Błąd użycia `T.a` nie jest błędem widoczności.
Trzy standardowe widoczności, które chcemy wprowadzić, to:

- **public** – symbol jest widoczny z zewnątrz.
- **private** – symbol nie jest widoczny z zewnątrz.
- `public_readonly` – symbol jest widoczny z zewnątrz, ale tylko w trybie odczytu.

Domyślne, oraz możliwe widoczności symbolu zależą od kontekstu. Przykładowo zmienna statyczna wewnątrz funkcji domyślnie jest prywatna, podczas gdy funkcja wewnątrz przestrzeni nazw domyślnie jest publiczna.
Dodatkowo istnieje specjalny rodzaje widoczności dotyczące klas i dziedziczenia – `protected`. Możliwe, że dodatkowe rodzaje będą też wprowadzone przez system modułów, do określania widoczności symboli między plikami/modułami.

10.7 Uproszczenie refactor-u oraz nauki języka

Projektując składnie poza aspektami takimi jak niska ilość kodu "boilerplate", czytelność, zwracamy również dużą uwagę na "spójność". Rozumiemy przez, że schematy składniowe o analogicznych zadaniach występujące w różnych miejscach/kontekstach, powinny mieć podobną konstrukcję, w której te samy elementy mają te same znaczenie. Powiązaną własnością do spójności jest również łatwość refactoru kodu. Przykładowo prosta powinna być zmiana lambdy w funkcję. Spójna powinny też takie konwencje jak `len(container)` / `container.len()`.
Te dwie własności poza uproszczeniem modyfikacji kodu, mamy nadzieję, że spowodują również, że język będzie intuicyjny. Przykładowo programista widząc nową funkcje języka, może wydedukować jej znacznie dzięki znajomości innych podobnych do niej. Mamy nadzieje, że wpłynie to pozytywnie na naukę języka oraz szybkość rozumienia nowego kodu.

10.8 Składnia "przyjemna"

Drugą własnością składni języka, którą chcemy uzyskać, może bardzo nie formalnie określić jako "przyjemność" z programowania w niej. Ciężko określić dokładnie jakie własności są tutaj najważniejsze, i jest to silnie subiektywne odczucie, ale elementy, na których się skupiamy:

- Niska ilość kodu "boilerplate" – nie wprowadzającego żadnego znaczenia do programu.
- Istnieje elementów składniowych realizujące powszechnie stosowane schematy w zrozumiały sposób. Przykładowo `while (true) { -> loop {`, `for (_: 0..n) { -> repeat (n) {`.
- Balans lukrów sytaktycznych. Wprowadzanie tego typu elementów, pozytywnie wpływa na składnie, ale zbyt duża ich liczba może składnie "przekomplikować".