Несколько слов о reference и value объектах. Основное семантическое (смысловое, поведенческое) различие проявляется в том, как работает присваивание. Возможны 4 варианта

1. reference1 := reference2 // присваивается один ссылочный объект другому
2. reference := value // присваивается объект значение объекту ссылка
3. value1 := value2 // присваивается один объект значение другому
4. value := reference // присваивается ссылочный объект объекту значение

При этом все тип объекта в правой части присваивания конформен типу объекта в левой.

1. reference1 := reference2 // присваивается один ссылочный объект другому
   1. после выполнения такого присваивания на один и тот же объект указывают две ссылки
2. reference := value // присваивается объект значение объекту ссылка
   1. создаётся копия value объекта и ссылка на нее размещается в reference
3. value1 := value2 // присваивается один объект значение другому
   1. в каждое поле (member attribute) value1 копируется содержимое соответствующего поля из value2
4. value := reference // присваивается ссылочный объект объекту значение
   1. в каждое поле (member attribute) value копируется содержимое соответствующего поля из объекта, на который указывает ссылка reference

Аналогичное правило действует и для передачи параметров подпрограммам. Пример с комментариями ниже.

reference **is ref** A

value **is val** A

foo (reference) /\* просто передаётся ссылка на объект (reference) как параметр процедуры\*/

foo (value) /\* создаётся копия value и ссылка на неё передаётся процедуре\*/

foo (ref value) /\* создаётся ссылка на объект value и эта ссылка передаётся, как параметр процедуры\*/

goo (reference) /\* в каждое поле нового объекта значения типа А прописывается соответствующее значение поля из того объекта, на который ссылается reference и этот новый объект и будет параметром goo \*/

goo (value) /\* в каждое поле нового объекта значения типа А прописывается соответствующее значение поля из объекта value и этот новый объект и будет параметром goo \*/

foo (argument: **ref** A) **do** … **end**

goo (argument: **val** A) **do** … **end**

* Ограничение на использование **ref** <expression>. Такая конструкция валидна если тип выражения <expression> является типом значением (value)
* Небольшой комментарий про перегрузку имён. Нельзя перегружать имена подпрограмм, если их сигнатуры различаются лишь видом передаваемых параметров. Т.е. нельзя делать перегрузку вида

foo (argument: **ref** A) **do** … **end**

foo (argument: **val** A) **do** … **end**

Другими словами **ref** A и **val** A – это все равно один и тот же тип А.

А теперь ещё пример о параметрах подпрограмм. Пусть некоторая процедура описана следующим образом

foo (arg: **ref** Integer) **do**

arg := arg + 1

**end**

x1: **val** Integer **is** 5

x2: **ref** Integer **is** 5

// очевидно, что значения x1 и x2 совпадают и равны 5

foo (x1)

foo (x2)

// А чему тут равны x1 и x2?

На этот вопрос сразу нет ясного ответа. Поэтому самое простое решение — это запретить присваивание в параметры подпрограмм внутри тел подпрограмм. Т.е. можно рассматривать параметры любой подпрограммы как локальные сущности, описанные с префиксом const. Примерно так (это пример некоторого псевдо кода)

foo (**\_1**) **do** /\* здесь \_1 означает, что foo надо вызывать с не менее чем одним параметром\*/

**const** arg: **ref** Integer **is \_1** /\* а это обращение к первому параметру \*/

arg := arg + 1 // И тогда это ошибка времени компиляции

**end**

И есть ещё один вариант вызова

foo (**ref** x1)

но семантически он эквивалентен foo (х2)

Таким образом все параметры передаются по значению и семантика параметров задана в сигнатуре подпрограммы явно через префикс ref или val перед типом или при отсутствии префикса будет тот тип параметра как указано в теле юнита, который задаёт тип параметра.

А уж если было упомянуто ключевое слово const, то давайте рассмотрим, как константные объекты передаются как параметры.

**const** tConst **is** Т

foo (tConst)

Вопрос – а какая должна быть сигнатура у foo, чтобы такой обращение было валидным?

foo (**const** arg: T) **do**

arg.goo (…)

**end**

Константные объекты можно передавать в подпрограммы только в константные параметры. Это необходимо для того, чтобы компилятор мог на этапе статической проверки правильности программы убедиться в том, что все обращения к членам-подпрограммам валидны. Аналогично тому, как если бы пытались вызывать ту же goo в том же блоке, где описано tConst

tConst.goo (… )

такой вызов валиден, если только goo в контейнере Т описана как **safe** или **pure** подпрограмма. Только в таком случае гарантируется целостность константного объекта на этапе компиляции.

А если у нас есть описание вида

**deep const** dConst **is** T

то тогда необходимо проверить транзитивно весь граф вызовов начиная с самого первого, чтобы убедиться, что это граф содержит только обращения к safe и pure подпрограммам.

goo (**deep** **const** arg: T) **do**

arg.goo (…)

**end**

dConst.goo (…)

Таким образом **deep const** имеет семантику, что весь граф объектов достижимых от данного неизменен и это значит, что все операции над этим графом валидны, если их выполняют safe и pure подпрограммы. А просто const что подграф, непосредственно достижимых объектов от данного неизменен и, соответственно подпрограммы, которые могут работать с подграфом должны быть **safe** или **pure**.

……

Рассмотрим обращение вида foo () или x.foo () или a или a.b. Другими словами обращение к члену контейнера как квалифицированное, так и не квалифицированное валидно тогда и только тогда

- имя члена юнита однозначно разрешается исходя из области видимости и в случае подпрограмм исходя из типов фактических параметров

- при наличии параметров кортеж типов фактических параметров согласуется с кортежем формальных параметров

- если обращение квалифицированное, то тот член к которому производится обращение не объявлен, как hidden

Несколько определений:

Тип согласуется с другим типом, если

- тип конформен другому типу

- есть функция преобразования от первого типа ко второму

Подпрограммы преобразования – это подпрограммы с именем := такие, что

Если это процедура с одним параметром, то эта процедура вызывается, когда присваивание идёт для объекта данного контейнера, а если это функция без параметров, то когда объект данного типа передаётся как параметр в некоторую подпрограмму с типом параметра куда производится преобразование. Рассмотрим пример

**unit** T1

:= (that: T2) **do … end** // Преобразование из T2 в T1

:= (): T2 **do … end** // Преобразование из T1 в T2

**end**

**unit** T2

**end**

t1 **is** T1

t2 **is** T2

t1 := t2 // Обращение к t1.”:=” (t2)

t2 := t1 // Ошибка компиляции, так как в T2 нет подпрограмм преобразования

foo (a: t2) **do … end**

foo (t1) // Что эквивалентно foo (t1.”:=” ())