Type Systems

David Barrett-Lennard

21 Feb 2013

# Introduction

This is a work in progress in its early stages that aims to formalise many of the ideas contained in TTM/IM, and also to make some relatively minor changes such as dropping the notion of types *omega* and *alpha*.

This document doesn't attempt to properly explain the rationale for the various definitions.

# Background

These are standard definitions that are repeated here for convenience.

Definition: (P,≤) is a *poset* (partially ordered set) if

* ∀a∈P a≤a (reflexivity)
* ∀a∈P ∀b∈P ((a≤b) ∧ (b≤a) → (a=b)) (antisymmetry)
* ∀a∈P ∀b∈P ∀c∈P ((a≤b) ∧ (b≤c) → (a≤c)) (transitivity)

Definition: Let (P,≤) be a poset and S⊆P. Then

* x is an *upper bound* of S if (x∈P) ∧ (∀a∈S a ≤ x)
* x is the *greatest element* of S if (x∈S) ∧ (∀a∈S a ≤ x)
* x is the *supremum* of S if x is the least element of the set of upper bounds of S

 i.e. (x∈P) ∧ (∀a∈S a ≤ x) ∧ (∀y∈P ((∀a∈S a ≤ y) → (x ≤ y)))

* x is a *maximal element* of S if (x∈S) ∧ (∀a∈S (x ≤ a → x=a))
* x is a *lower bound* of S if (x∈P) ∧ (∀a∈S x ≤ a)
* x is the *least element* of S if (x∈S) ∧ (∀a∈S x ≤ a)
* x is the *infimum* of S if x is the greatest element of the set of lower bounds of S

 i.e. (x∈P) ∧ (∀a∈S x ≤ a) ∧ (∀y∈P ((∀a∈S y ≤ a) → (y ≤ x)))

* x is a *minimal element* of S if (x∈S) ∧ (∀a∈S (a ≤ x → x=a))

Note: each of the following is unique if it exists:

* greatest element
* supremum
* least element
* infimum

Definition: Let (P,≤) be a poset. (P,≤) is a *lattice* if every two members a and b of P have both a supremum (called the *join* denoted by a∨b) and an infimum (called the *meet* denoted by a∧b).

Definition: A lattice is a *bounded lattice* if it has both a greatest and least element. These are typically called *top* (⊤) and *bottom* (⊥) respectively.

Definition: A lattice is a *distributive lattice* if the join and meet operations distribute over each other. i.e.

 ∀a∈P ∀b∈P ∀c∈P a∧(b∨c) = (a∧b)∨(a∧c)

 ∀a∈P ∀b∈P ∀c∈P a∨(b∧c) = (a∨b)∧(a∨c)

Definition: Let (P,≤) be a lattice. (S,≤) is a *sublattice* of (P,≤) if

 (S≠∅) and (S⊆P) and (∀a,b ∈ S (a∨b)∈S) and (∀a,b ∈ S (a∧b)∈S)

# Some lattice properties

Definition: Let (P,≤) be a poset and S⊆P. Let

* upperbounds(S) = {x∈P | ∀a∈S a ≤ x}
* lowerbounds(S) = {x∈P | ∀a∈S x ≤ a}

Claim: Let (P,≤) be a poset. Then ∀x∈P ((x is a maximal element of P) ↔ upperbounds({x}) = {x})

Proof:

(→)

Let x be a maximal element of S

So (x∈P) ∧ (∀a∈P (x ≤ a → x=a))

Need to show upperbounds({x}) = {x}

 (upperbounds({x}) ⊆ {x})

 Let y∈upperbounds({x})

 So y∈P and ∀a∈{x} a ≤ y

 So x ≤ y

 Therefore y = x (because (x ≤ y) and (∀a∈P (x ≤ a → x=a)))

 (upperbounds({x}) ⊇ {x})

 This follows from the fact that x is an upper bound of {x}

(←)

Let x∈P satisfy upperbounds({x}) = {x})

i.e. the only upper bound of {x} is x.

Need to show x is a maximal element of P

(∀a∈P (x ≤ a → x=a))

 Let a∈P and x ≤ a

 a is an upper bound of {x} (because x ≤ a)

 x=a (the only upper bound of {x} is x)

Claim: Let (P,≤) be a poset. Then ∀x∈P ((x is a minimal element of P) ↔ lowerbounds({x}) = {x})

Claim: Let (P,≤) be a lattice. Then ∀S⊆P (upperbounds(S),≤) is a sublattice of (P,≤)

Proof:

upperbounds(S) ⊆ P

(∀a,b ∈ upperbounds(S) (a∨b) ∈ upperbounds(S))

 Let a,b ∈ upperbounds(S)

 Need to show (a∨b) ∈ upperbounds(S)

 Need to show ∀c∈S c ≤ (a∨b)

 Let c∈S

 (c ≤ (a∨b))

 a∨b is upper bound of {a,b} (because a∨b is least upper bound of {a,b})

 a ≤ (a∨b) (definition of upper bound)

 a is upper bound of S (because a ∈ upperbounds(S))

 c ≤ a (because (c∈S) and (a is upper bound of S))

 c ≤ (a∨b) (because (c ≤ a) and (a ≤ (a∨b)))

(∀a,b ∈ upperbounds(S) (a∧b) ∈ upperbounds(S))

 Let a,b ∈ upperbounds(S)

 Need to show (a∧b) ∈ upperbounds(S)

 Need to show ∀c∈S c ≤ (a∧b)

 Let c∈S

 (c ≤ (a∧b))

 a is upper bound of S (because a ∈ upperbounds(S))

 c ≤ a (because (c∈S) and (a is upper bound of S))

 b is upper bound of S (because b ∈ upperbounds(S))

 c ≤ b (because (c∈S) and (b is upper bound of S))

 c is lower bound of {a,b} (because c ≤ a and c ≤ b)

 c ≤ (a∧b) (because a∧b is greatest lower bound of {a,b})

Similarly: Let (P,≤) be a lattice. Then ∀S⊆P (lowerbounds(S),≤) is a sublattice of (P,≤)

# Lemmas

Lemma 1: If X is a finite set of subsets of Y with Y∈X and X is closed under intersection then

 ∀a∈X ∀b∈X ∃u∈X ( (a⊆u) ∧ (b⊆u) ∧ (∀v∈X (((a⊆v) ∧ (b⊆v)) → u⊆v)))

(i.e. for every two sets in X there exists the minimum superset of the two sets).

Proof: Let X be a finite set of subsets of Y

 Let Y∈X

 Let X be closed under intersection

 Let a∈X

 Let b∈X

 We show ∃u∈X ( (a⊆u) ∧ (b⊆u) ∧ (∀v∈X (((a⊆v) ∧ (b⊆v)) → u⊆v)))

 Suppose not

 i.e. ∀u∈X (((a⊆u) ∧ (b⊆u)) → ∃v∈X ((a⊆v) ∧ (b⊆v) ∧ ¬(u⊆v))) ................(1)

To obtain a contradiction we show there exists an infinite sequence S1, S2, S3, ... of elements of X where each element Si+1 is a proper subset of the previous element Si. This contradicts the assumption that X is finite.

Define predicate P on subsets of Y such that ∀y⊆Y P(y) = ((y∈X) ∧ (a⊆y) ∧ (b⊆y))

We show by induction ∀i ≥ 1 P(Si).

Let S1 = Y

P(S1)

 (we show P(Y) i.e. (Y∈X) ∧ (a⊆Y) ∧ (b⊆Y)

 Y∈X (assumption)

 a∈X (assumption)

 b∈X (assumption)

 ∀y∈X (y⊆Y) (because X is a set of subsets of Y)

 a ⊆ Y (substitute y=a in ∀y∈X (y⊆Y))

 b ⊆ Y (substitute y=b in ∀y∈X (y⊆Y))

In general for i ≥ 1 to get Si+1 from Si we assume Si satisfies P(Si) and show as follows there exists some Si+1 which is a proper subset of Si and satisfies P(Si+1):

 (Si∈X) ∧ (a⊆Si) ∧ (b⊆Si) (assume P(Si))

 ∃v∈X (a⊆v) ∧ (b⊆v) ∧ ¬(Si ⊆ v) (substitute u = Si in (1))

 let Si+1 = Si∩v

 P(Si+1)

 (we show (Si+1∈X) ∧ (a⊆Si+1) ∧ (b⊆Si+1))

 a ⊆ Si∩v (because (a⊆Si) ∧ (a⊆v))

 a ⊆ Si+1 (substitute Si+1=Si∩v in a ⊆ Si∩v)

 b ⊆ Si∩v (because (b⊆Si) ∧ (b⊆v))

 b ⊆ Si+1 (substitute Si+1=Si∩v in b ⊆ Si∩v)

 (Si∩v) ∈ X (X closed under intersection ∧ (Si∈X) ∧ (v∈X))

 Si+1 ∈ X (substitute Si+1=Si∩v in (Si∩v) ∈ X)

 Si+1 is a proper subset of Si

 (we show (Si+1 ⊆ Si) ∧ (Si+1 ≠ Si))

 Si+1 ⊆ Si (because Si+1 = Si∩v)

 Si+1 ≠ Si

 otherwise Si+1 = Si

 Si = Si ∩ v (substitute Si+1 = Si in lhs of Si+1 = Si ∩ v)

 Si ⊆ v (because Si = Si ∩ v)

 contradiction to ¬(Si ⊆ v)

Note: Without the assumption that X is finite the lemma doesn't hold. For example let Y = [0,2] which is a subset of the reals, and let X = { {}, {0}, {1}, [0,2] } ∪ { {0,1} ∪[x,2] | 1 < x ≤ 2 }. Then X is an infinite set of subsets of Y with Y∈X and X is closed under intersection. Let a={0} and b={1}. Let U be the set of upper bounds of {a,b}. i.e. U = {u∈X|a⊆u and b⊆u} = { [0,2] } ∪ { {0,1} ∪[x,2] | 1 < x ≤ 2 }. U has no least element therefore sup {a,b} doesn't exist.

Lemma 2: Let X be a finite set of subsets of Y.  If Y is an element of X and X is closed under intersection then the poset (X,⊆) is a bounded lattice.

Proof:

(every pair of elements in X have an infimum in X)

 Let a,b∈X.

 We will show the infimum of {a,b} is a∩b.

 So need to show a∩b is the greatest element of the set of lower bounds of {a,b}

 The set of lower bounds of {a,b} is given by L = {x∈X | x⊆a ∧ x⊆b }

 a∩b ∈ L

 (a∩b) ∈ X (because X is closed under intersection)

 (a∩b) ⊆a (property of intersection)

 (a∩b) ⊆b (property of intersection)

 (a∩b) is upper bound of L

 Need to show: ∀x∈L x ⊆ (a∩b)

 Let x∈L

 (x∈X) ∧ (x⊆a) ∧ (x⊆b) (by definition of L)

 x ⊆ (a∩b) (because (x⊆a) ∧ (x⊆b))

(every pair of elements in X have a supremum in X)

 Let a,b∈X.

 Need to show the set of upper bounds of {a,b} has a least element u

 i.e. ∃u∈X ( (a⊆u) ∧ (b⊆u) ∧ (∀v∈X (((a⊆v) ∧ (b⊆v)) → u⊆v)))

 This was proven by lemma 1

(X has a least element)

 Let ⊥ = ∩X.

 ⊥∈X because X is finite and X is closed under intersection.

 ⊥ is the least element of X.

(X has a greatest element)

 Let ⊤=Y.

 ⊤∈X because Y∈X.

 ⊤ is the greatest element of X because X is a set of subsets of Y.

# Type system inheritance graph

This section is purely concerned with the required properties of the inheritance graph of a type system. It is not concerned with how to define types in a DDL.

Definition:  A *type* is a non-empty set of elements called *values*.

Note therefore that two types are identical if and only if they are equal as sets of values.

Definition:  Let S be a set of types and T1∈S, T2∈S. In the context of S we say T1 is a *subtype* of T2 (or equivalently T2 is a *supertype* of T1) if T1⊆T2.

Note: The subtype relation gives a partial order on a given set of types S (because the subtype relation is antisymmetric, transitive and reflexive). i.e. (S,⊆) is a poset.

A note on terminology: it is convenient to use a special term subtype instead of just subset because a subset of a type may not be a type of the type system. A subtype means both a subset and a type of the type system.

Definition: Let S be a set of types. In the context of S

* For every type T, let supertypes(T) = { T'∈S | T⊆T' } denote the set of supertypes of T.
* For every type T, let subtypes(T)= { T'∈S | T'⊆T } denote the set of subtypes of T.
* Type T∈S is called a *maximal type* if supertypes(T) = {T}. i.e. T has no proper supertype in S.
* Type T∈S is called a *minimal type* if subtypes(T) = {T}. i.e. T has no proper subtype in S
* Let maximaltypes(S) = { T∈S | supertypes(T) = {T} }
* Let minimaltypes(S) = { T∈S | subtypes(T) = {T} }

Definition: A *typeset* S is a set of types satisfying

* ∅ ∉ S
* ∀R1∈maximaltypes(S) ∀R2∈maximaltypes(S) (R1 ≠ R2 → (subtypes(R1) ∩ subtypes(R2) = ∅))
* ∀R∈maximaltypes(S) ∀t1∈subtypes(R) ∀t2∈subtypes(R) ((t1∩t2) ∈ subtypes(R))

i.e.

* Every type is non-empty
* If R1, R2 are distinct maximal types then subtypes(R1) and subtypes(R2) are disjoint
* For every maximal type R, subtypes(R) is closed under intersection

Claim: ∀R∈maximaltypes(S) ((subtypes(R) is finite) → ((subtypes(R),⊆) is a bounded lattice))

Proof: This follows from lemma2

The essential idea is that the types in a typeset can be partitioned into a set of mutually exclusive bounded lattices.   The partition provides a basis for the concept of type check failures on equality comparisons in a programming language. It is assumed equality between the maximal types is undefined as far as users of the type system are concerned. Note that we are talking about mutual exclusion between sets of types, we're not talking about mutual exclusion between sets of values. This can be contrasted with the following TTM/IM axiom which states that distinct root (i.e. maximal) types are disjoint:

 ∀R1∈maximaltypes(S) ∀R2∈maximaltypes(S) (R1≠R2 → (R1∩R2 = ∅))

This axiom doesn't appear to be consistent with the idea that equality between root types is undefined and therefore should produce static type check failures. If that were the case, why bother with an axiom that states that values always compare not equal? If this axiom matters to the user then the user must care about equality comparisons between distinct root types so why would such comparisons be outlawed?

Definition:  Let S be a typeset

* ∀T∈S let top(T) denote the unique maximal type of S which is a supertype of T
* ∀T∈S let bottom(T) = ∩subtypes(top(T)) which is the unique minimal type of S which is a subtype of T
* ∀T1,T2 ∈S with top(T1)=top(T2), let (T1∧T2)∈S denote the type which is the intersection of T1,T2 (and happens to also be the unique least specific common subtype)
* ∀T1,T2 ∈S with top(T1)=top(T2), let (T1∨T2)∈S denotes the type which is the unique most specific common supertype.

Claim: Let S be a typeset. There are numerous equivalent conditions for when T1,T2 ∈S belong to the same bounded lattice:

∀T1∈S ∀T2∈S

 (top(T1) = top(T2)) ↔

 (bottom(T1) = bottom(T2)) ↔

 ((supertypes(T1)∩supertypes(T2)) ≠ ∅) ↔

 ((subtypes(T1)∩subtypes(T2)) ≠ ∅) ↔

 (T1∈subtypes(top(T2))) ↔

 (T1∈supertypes(bottom(T2)))

Note:

* ∀T∈S (subtypes(top(T)) = supertypes(bottom(T)))
* ∀T∈S (bottom(T) ⊆ T ⊆ top(T))
* ∀T∈S (supertypes(T),⊆) is a bounded lattice with minimal type T and maximal type top(T)
* ∀T∈S (subtypes(T),⊆) is a bounded lattice with minimal type bottom(T) and maximal type T
* ∀T∈S minimaltypes(supertypes(T)) = maximaltypes(subtypes(T)) = {T}
* ∀T∈S maximaltypes(supertypes(T)) = {top(T)}
* ∀T∈S minimaltypes(subtypes(T)) = {bottom(T)}
* ∀T1∈S ∀T2∈S ((T1⊆T2) → (supertypes(T2),⊆) is a sublattice of (supertypes(T1),⊆)))
* ∀T1∈S ∀T2∈S ((T1⊆T2) → (subtypes(T1),⊆) is a sublattice of (subtypes(T2),⊆)))

## Type system without inheritance

Comment: dropping both *omega* and *alpha* means a type system which supports inheritance generalises one without inheritance.  The latter is just a special case of the former where each root type has no proper subtypes.

## MST of a value

Definition: We say v is a *value* of type T if v∈T. Note that a value can be of multiple types.

Definition: In the context of typeset S, for any value v let types(v) = {T∈S | v∈T}. We assume types(v)≠ ∅

Theorem: For any finite typeset S and value v in S, types(v) = supertypes(∩ types(v)).

Proof:

v is a value so types(v) ≠ ∅

(∩types(v))∈S (because types(v) ≠ ∅ and types(v)⊆S and S is finite and S is closed under intersection)

(types(v) ⊆ supertypes(∩types(v)))

 Let T∈types(v)

 v∈T (because T∈types(v) and types(v) = {T∈S | v∈T})

 (∩ types(v)) ⊆T (because T∈types(v) and property of intersection)

 T ∈ supertypes(∩ types(v)) (because supertypes(∩ types(v)) = { T∈S | (∩types(v))⊆T })

(supertypes(∩types(v)) ⊆ types(v))

 Let T ∈ supertypes(∩types(v))

 (∩ types(v))⊆T (because supertypes(∩ types(v)) = { T∈S | (∩types(v))⊆T })

 ∀T'∈ types(v) (v∈T') (by definition of types(v))

 v∈ (∩ types(v)) (by definition of intersection)

 v∈T (because v∈(∩ types(v)) and (∩ types(v))⊆T)

 T∈types(v) (because v∈T and types(v) = {T∈S | v∈T})

Corollary: types(v) is a bounded lattice.

Definition: (∩types(v))∈S is called the *Most Specific Type* of v and denoted by mst(v).

# Type System as set of types plus set of read only operators

## Ordered tuple

Definition: Let L be an ordered list of zero or more elements. Then |L| denotes the size of the list, and L[i] denotes the ith element of the list for 0 < i < |L|. A list can be written explicitly in square brackets. The empty list is denoted by [].

Definition: Let S be a typeset and L be a finite ordered list of types with |L| ≥ 0 and ∀i L[i]∈S . Then ×L denotes the conventional ordered Cartesian product L[1]×L[2]×...×L[|L|] representing a set of *ordered* tuples. Let () denote the empty tuple.

Note:

 ×[] = { () }

 ∀L1 ∀L2 ((|L1|≠|L2|) → ((×L1 ∩ ×L2) = ∅))

 ∀L1 ∀L2 (((|L1|=|L2|) ∧ (∀i L1[i] ⊆ L2[i])) → (×L1 ⊆ ×L2))

A read-only operator has an ordered list of zero or more types for the input parameters. It is convenient to assume an invocation is made with an ordered tuple of values. Therefore the Cartesian product of the list of input types gives the set of ordered tuples on which the operator is *nominally* defined. We say nominally because we allow operators to be partial because it isn't generally practical for the type system to have all the types required for every operator to be complete.

We do not require these Cartesian products to be types of the type system.

## Read-only operator

A *read only operator* is a pure mathematical function, possibly partial on a given finite list of zero or more input parameters of given declared types, and there is a single declared return type. This is formalised as follows:

Definition: A *read only operator* O = (N,X,Y,G) is a 4-tuple where N is the *operator name*, X is a finite list of zero or more *declared input parameter types*, Y is the *declared output type* and G is the *graph* (or "body") of the operator. G is a subset of (×X) × Y which must satisfy

 ∀x ∀y1 ∀y2 (((x,y1)∈G ∧ (x,y2) ∈ G) → (y1 = y2))

### Restrictions

Given operators O1 =(N,X1,Y1,G1) and O2 = (N,X2,Y2,G2) with the same name, if (×X1)⊆(×X2) then we say O1 is a *restriction* of O2 and there is an axiom (below) to ensure that O1 is the mathematical function O2 restricted to a smaller domain, but possibly also allowing a subtype of the return type (i.e. a smaller codomain meaning Y1⊆Y2).

For example, let type INTEGER be a subtype of type RATIONAL. Consider the following operators:

 (ADDITION, [INTEGER,INTEGER], INTEGER, { ((x,y),x+y) | x∈INTEGER, y∈INTEGER})

 (ADDITION, [RATIONAL,RATIONAL], RATIONAL, { ((x,y),x+y) | x∈RATIONAL, y∈RATIONAL})

The motivation is for the type system to allow a wider set of valid programs to compile without unnecessary type check failures.

The opposite is possible as well. It is useful to allow operators of the form (N,X,∅,∅) to indicate the operator named N is never callable on X and therefore should produce a type check failure. For example let type ZERO be a subtype of type RATIONAL. Consider the following operators:

 (DIVISION, [RATIONAL,RATIONAL], RATIONAL, { ((x,y),x/y) | x∈RATIONAL, y∈RATIONAL\{0} })

 (DIVISION, [RATIONAL,ZERO], ∅,∅ )

## Type system

A *type system* (S,F) is a typeset S and a set of read-only operators F satisfying the following:

1. ∀N ∀X ∀Y ∀G

(N,X,Y,G)∈F →

 (∀i X[i] ∈ S) ∧ (Y∈(S∪{∅}))

1. ∀N ∀X ∀Y1 ∀Y2 ∀G1 ∀G2

(N,X,Y1,G1)∈F ∧ (N,X,Y2,G2)∈F →

 (Y1=Y2) ∧ (G1=G2)

1. ∀N ∀X1 ∀X2 ∀Y1 ∀Y2 ∀G1 ∀G2

((N,X1,Y1,G1)∈F ∧ (N,X2,Y2,G2)∈F ∧ (×X1 ∩ ×X2)≠∅) →

 ∃X ∃Y ∃G(×X1 ⊆ ×X) ∧ (×X2 ⊆ ×X) ∧ (N,X,Y,G)∈F

1. ∀N ∀X1 ∀X2 ∀Y1 ∀Y2 ∀G1 ∀G2

((N,X1,Y1,G1)∈F ∧ (N,X2,Y2,G2)∈F ∧ (×X1 ⊆ ×X2)) →

 ((Y1≠∅ ∧ Y2≠∅) → (Y1~Y2)) ∧ (Y1⊆Y2) ∧ (G1 = (G2 ∩ (×X1 × Y2))))

1. bool ∈ S, where bool = {false,true}. The following must be satisfied:
	1. (FALSE,[],bool,{((),false)})∈F
	2. (TRUE,[],bool,{((),true)})∈F
	3. (AND,[bool,bool], bool,GAND)∈F

where GAND = {((false,false),false),((false,true),false),((true,false),false),((true,true),true)}

* 1. (OR,[bool,bool], bool,GOR)∈F

where GOR = {((false,false),false),((false,true),true),((true,false),true),((true,true),true)}

* 1. (NOT,[bool], bool,GNOT)∈F

where GNOT = {((false),true),((true),false)}

1. subtypes(bool) = supertypes(bool) = {bool}
2. ∀T1∈S ∀T2∈S

(top(T1)= top(T2)) →

 (=, [T1,T2],bool,{ ((x1,x2),x1=x2) | (x1∈T1)∧ (x2∈T2) } ) ∈F

1. ∀T1∈S ∀T2∈S ∀G

(top(T1)≠ top(T2)) →

 (=, [T1,T2],bool,G) ∉F

Explanation of each rule:

1. Ensures the types in the operator signature of each operator in F are drawn from the set of types S. Note however that the return type of a function can be given as the empty set.
2. Ensures there is at most one operator of a given name and list of input parameter types
3. Ensures that operators with the same name cannot be defined on partially overlapping input types unless they arise as restrictions of an underlying operator defined on common supertypes of the declared types of the input parameters.
4. Ensures that an operator with the same name and which has subtypes of the input parameters is in fact a restriction. We allow Y1⊆Y2 (we don't mandate Y1 = Y2) in order to support subtypes of return types on restrictions.
5. The type system must include the bool type with at least the logical FALSE, TRUE, AND, OR, NOT operators. It is convenient in this treatment to give them specific names, but the intention is not actually to mandate the choice of names in a programming language.
6. There are no proper supertypes or subtypes of bool.
7. Ensures the type system defines equality comparison operator =(T1,T2) whenever top(T1)= top(T2). In particular for every type T the type system has a complete read-only dyadic operator with signature =(T,T) which is *equality comparison on type T*. By the Principle of (Read-Only) Operator Inheritance, the *restrictions* =(T1,T2) of =(T,T) for all subtypes T1,T2 of T are available.
8. If top(T1)≠ top(T2) then equality comparison is undefined between T1 and T2.

## Declared type of an expression

The *declared type* of an expression of nested operator invocations under type system (S,F) is defined recursively as follows:

In an expression which represents an operator invocation consider that the compiler can determine statically the name N of the operator and the list X of the declared types of the argument expressions. The declared type of the operator invocation is given by

 T = ∩ { Y | ∃ (N,X',Y,G) ∈F ∧ (×X ⊆ ×X') }

Note that T∈(S∪{∅}). If T=∅ then it is expected the compiler will generate a static error (either reporting there is no operator named N or else reporting a type check failure on one more of the argument expressions).

If T∈S then any of the operators (N,X',Y,G) ∈F satisfying (×X ⊆ ×X') can be used (the axioms imply they must all agree on the result of the invocation if it exists). This embodies the *principle of read only operator inheritance*.