Proposal for type system

David Barrett-Lennard

15 Nov 2013

# Required set theoretic properties of a type system

For a separation of concerns we first describe the required set theoretic properties of the type system before commencing with a discussion of a *Type Definition Language* (TDL).

It is proposed that a (static) type system adheres to the following principles:

1. A *type system* is a set of sets which satisfies certain requirements below. The elements of a type system are called *types*. The elements of types are called *values*. Of course two types are equal if and only they represent the same set of values.
2. Let T1,T2 be types of a type system. T1 is called a subtype of T2 if and only if T1 is a subset of T2. For any type T let subtypes(T) denote the set of subtypes of T.
3. There is exactly one empty type ∅ in the type system. ∅ is a subtype of every type
4. Unlike TTM there is no concept of a type *alpha*.
5. A *root type* is a type with no proper supertype.
6. Let R1, R2 be root types. Then R1≠R2 ⇒ subtypes(R1)∩subtypes(R2)={∅}
7. Equality is defined between values if and only if they are of the same root type. Let the static types of expressions e1,e2 be subtypes of root types R1,R2 resp. then the expression e1=e2 produces a static type check failure if and only if R1≠R2.
8. Let R1, R2 be root types. Let T1∈subtypes(R1) and T2∈subtypes(R2) then T1∩T2 and T1∪T2 exist as types in the type system if and only if R1=R2.
9. It follows from the previous item that for each root type R, subtypes(R) is closed under intersection and union. i.e. ∀T1∈subtypes(R) ∀T2∈subtypes(R) T1∩T2∈subtypes(R) and T1∪T2∈subtypes(R). Therefore subtypes(R) is a *lattice* with top R and bottom ∅.

Within a lattice the closure under intersection and union is what we need to allow the static type of relation expressions to be easily defined.

The intersection of two types in the same lattice is the least specific common subtype and the union of two types is the most specific common supertype.

## Types and operators are abstract

Types are mathematical sets and therefore are pure abstractions divorced from particular programming languages and implementations on computer hardware (or even abstract Turing machines). We allow and indeed promote types such as the reals which are uncountably infinite. Clearly we do not require a type to support representation of all its values.

## Defining a set of literals which represent values

The set of literals which conform to a grammar provides a basis for formalising the representation of values on an abstract machine. The grammar specifies syntax. A denotation specifies semantics. The TDL promotes a *separation of concerns* is this regard by providing language constructs that defines the grammar independently of the semantics.

Let NATURAL be a type name. Consider the following two *operator signatures*:

NATURAL ← 0

NATURAL ← s(NATURAL)

(in the first case '0' is the name of a nullary operator).

These two operators can be regarded as *productions* of a grammar where NATURAL is interpreted as a *non-terminal* symbol. This defines a *language* - i.e. the set of finite strings that conform to the grammar:

{ 0, s(0), s(s(0)), s(s(s(0))), ... }

We call these strings *literals*. Each string is finite, but there are an infinite number of them. A *denotation* assigns a value to each literal. In this case the intended denotation is:

literal 0 denotes the integer 0

literal s(0) denotes the integer 1

literal s(s(0)) denotes the integer 2

literal s(s(s(0))) denotes the integer 3

etc

We can summarise this approach as follows:

1. Define types as mathematical sets of values
2. Define a set of operator signatures and interpret as a grammar defining a set of literals
3. Define a denotation which assigns a value to each literal (noting that often there may be multiple literals that denote a given value)

This provides a general way to formalise a notion of representing values on an abstract machine. Perhaps to illustrate its flexibility consider the following operator signatures:

BIT ← 0

BIT ← 1

BYTE ← byte(BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT)

WORD ← word(BYTE,BYTE)

The language for this grammar gives rise to exactly 65536 literals of type WORD.

Example: Define a set of operator signatures that describe the IEEE 754 double precision binary float format.

Syntax:

BIT ← 0

BIT ← 1

SIGN ← sign(BIT)

EXPONENT ← exponent(BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT)

FRACTION ← fraction(

BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,

BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,

BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,

BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,

BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT,

BIT,BIT)

IEEEFLOAT64 ← float64(SIGN,EXPONENT,SIGNIFICAND)

Semantics:

sign(x) = (-1)x

exponent(b0,b1,...,b10) = b0 + 2b1 + ... + 210b10

fraction(b0,b1,...,b51) = 1 + 2-1b51 + ... + 2-52b0

float64(s,e,f) = s × f × 2e-1024

Example: Define a set of operator signatures allowing for the representation of a list of signed 8 bit numbers.

Syntax:

BIT ← 0

BIT ← 1

SBYTE ← signedbyte(BIT,BIT,BIT,BIT,BIT,BIT,BIT,BIT)

LIST ← empty

LIST ← headtail(SBYTE,LIST)

Semantics:

signedbyte(b7,b6,...,b0) = b0 + 2b1 + ... + 26b6 - 27b7

empty = []

headtail(x,L) = append([x],L)

## Canonical forms for type expressions

For each root type R, subtypes(R) must form a lattice closed under union and intersection. This can lead to an explosion of combinations. Consider for example that we have three overlapping subtypes of INTEGER named PRIME, EVEN and SN where SN are integers in [1,1000]. That leads to a lattice with 14 types!

INTEGER

SN =

(EVEN∩PRIME)∪SN

PRIME

EVEN

PRIME∪SN

EVEN∪SN

EVEN∪PRIME

EVEN∩PRIME =

EVEN∩PRIME∩SN

EVEN∩SN

PRIME∩SN

∅

EVEN∪PRIME∪SN

(EVEN∩SN)∪PRIME

(PRIME∩SN)∪EVEN

It's unreasonable to expect a programmer to invent a name for each type. Instead we assume the type system utilises *type expressions* allowing union and intersection operators to be used to express *compound types* to help represent all the types of a lattice.

For any type there are infinitely many type expressions that denote it. We shall define the unique *canonical type expression* for each type.

We need to determine appropriate *language constructs* that implicitly define a lattice of types in a straightforward manner.

The following principles are used

1. There is no need to declare that ∅ is a subtype of any type - it is implicit.
2. A type is a root type if and only if it is defined using language constructs which neither explicitly nor implicitly defines a proper super type.
3. A type which has a simple name is typically called an *atomic type* and denoted by a unique *atomic type expression*. Otherwise it is called a *compound type* and its canonical type expression involves a union of terms where each term is an intersection of atomic type expressions.
4. The symbol ∅ denotes the empty atomic type.

## Reducing the set of distinct types

By default, if types like EVEN, PRIME, SN each are declared as subtypes of INTEGER, then the compiler assumes they give rise to all combinations of compound types on them (i.e. we don't expect the compiler to be clever enough to determine that the lattice can be simplified). Therefore we allow the TDL to declare additional subtype relationships that exist between type expressions even though in theory these declarations are redundant and could be derived by the compiler. E.g.

type EVEN∩PRIME ⊆ SN;

Note that this one subtype declaration implies two equalities on the lattice:

EVEN∩PRIME = EVEN∩PRIME∩SN;

SN = (EVEN∩PRIME)∪SN;

(so two of the lattice nodes are removed by the subtype declaration).

## Read only operators

The *signature* of a read only operator named f is of the form

T ← f(T1,...,Tn)

where n ≥ 0. T1,...,Tn are the *operand types*. T is the *return type*.

f is *total* if it is defined on all of T1×T2×...×Tn.

f is *surjective* if

∀y∈T ∃x1∈T1...∃xn∈Tn y=f(x1,...,xn)

In this document we allow for the *definition* of a read only operator to be given with the following syntax

T ← f(T1 x1,...,Tn xn) :: *expr*;

where x1,...,xn may be free variables in *expr*.

## Value substitution principle

The *value substitution principle* is adhered to. Let read only operator f have signature

T←f(T1,...,Tn)

The expression f(e1,...,en) satisfies the static type check iff

1. each expression ei recursively satisfies the static type check; and
2. the static type of each ei is a subtype of Ti.

The static type of the expression f(e1,...,en) is defined to be T (regardless of whether the compiler can determine statically from the definition of f that it could in fact be assumed to be a proper subtype of T according to the set of possible values of the expression).

## Selector

By definition a *selector* of type T means a read only operator with signature of the form

T ← T(T1,...,Tn)

where n ≥ 0. In other words the name of the operator coincides with the name of the return type. It is not generally a requirement that the operator be total or surjective.

## Copy selectors

Every type T has a total and surjective *copy selector* with signature

T ← T(T)

## Disjoint union declaration

There is a language construct which defines a *disjoint union* of given types T1,...,Tn. This implicitly declares that the Ti are mutually disjoint, and gives rise to compound type expressions for all possible unions of the Ti.

Example

disjoint union SHAPE { CIRCLE, TRIANGLE, RECTANGLE };

This

1. Implies the existence of compound types CIRCLE∪TRIANGLE∪RECTANGLE, CIRCLE∪TRIANGLE, TRIANGLE∪RECTANGLE and CIRCLE∪RECTANGLE
2. Implies CIRCLE∩TRIANGLE = TRIANGLE∩RECTANGLE = CIRCLE∩RECTANGLE = ∅
3. Declares that SHAPE is an *alias* for CIRCLE∪TRIANGLE∪RECTANGLE.

RECTANGLE

TRIANGLE∪RECTANGLE

CIRCLE∪RECTANGLE

CIRCLE∪TRIANGLE

∅

SHAPE =

CIRCLE∪TRIANGLE∪RECTANGLE

TRIANGLE

CIRCLE

Specifying the alias name 'SHAPE' for the union is optional! i.e. the following syntax is permitted:

disjoint union { CIRCLE, TRIANGLE, RECTANGLE };

Assuming an implementation of CIRCLE, TRIANGLE and RECTANGLE is defined, it is straightforward for a compiler to generate an implementation of the disjoint union type, using a tagged union. The disjointness property implies

bool ← =(CIRCLE∪TRIANGLE∪RECTANGLE,CIRCLE∪TRIANGLE∪RECTANGLE)

can be implemented in terms of

bool ← =(CIRCLE,CIRCLE)

bool ← =(TRIANGLE,TRIANGLE)

bool ← =(RECTANGLE,RECTANGLE)

by appropriate testing of the tags used in the implementation of the tagged unions.

It is possible that without this declaration CIRCLE, TRIANGLE and RECTANGLE were root types of different type lattices. But the declaration implies that these are subtypes of CIRCLE∪TRIANGLE∪RECTANGLE. So the effect is pull together otherwise separate lattices.

Even though CIRCLE,TRIANGLE,RECTANGLE are mutually exclusive, equality is defined between them because they are subtypes of a common type, and to do otherwise would be at odds with the value substitution principle.

## Variable declarations

In general a variable is declared with a type expression followed by the variable name. For example

INTEGER x;

declares a variable name x of type INTEGER.

## Constraints

The following declares variable x with an additional constraint:

INTEGER x with x > 0;

Constraint declarations can appear on function signatures. For example

REAL y ← sqrt(REAL x) with x ≥ 0 ∧ y ≥ 0 ∧ x = y2;

Such constraints are ignored by the static type checker, but nevertheless they allow a compiler to automatically insert additional run time checks, provide useful warnings or generate more optimal code.

## Specialisation by constraint

One of the ways of defining a new type is to define it as a subtype of an existing type using a constraint that can be regarded as expressing a restricted comprehension over the given supertype. For example EVEN = { x∈INTEGER | x mod 2 = 0 } is expressed in the TDL as:

type EVEN : INTEGER x with x mod 2 = 0;

Consider that types EVEN and ODD have been defined as subtypes of INTEGER by specifying constraints, and this means the compiler can generate implementations of all three types. In theory the compiler can deduce that EVEN∩ODD=∅ and INTEGER = EVEN∪ODD. Nevertheless we allow a disjoint union declaration to be used to tell the compiler that this is the case:

disjoint union INTEGER { EVEN, ODD };

## Mounting a lattice into another lattice

Consider that T1 has so far been defined as a root type and T2 is a subtype of a root type not equal to T1. That implies that so far nothing depends on a definition of equality between these types. Then it may be possible to instead impose the condition that T1 is a subtype of T2. To do this we define a total selector for T2 which takes a single operand of type T1.

T2 ← T2(T1)

The selector must be total meaning there are no constraints imposed on the operand of type T1 - i.e. every value of type T1 must be mapped to some value of type T2. Normally this selector would not be surjective (otherwise it would follow that T1 = T2).

This can be regarded as a mechanism for "mounting" one lattice into another lattice at a defined position to create a bigger lattice.

The TDL uses the following syntax to mount an existing root type as a subtype of another type:

subtype T1 *x* as *expr*;

where *expr* is an expression of static type T2 in formal parameter *x*.

## Alias declaration

Example:

type SHAPE : CIRCLE∪TRIANGLE∪RECTANGLE;

The compiler generates an error if the given type expression involves types that come from different lattices.

## Defining equality between two types

Defining equality between two types has the effect of declaring the existence of union (and intersection) of the two types!

Example

bool ← =(RHOMBUS a, RECTANGLE b) ::

side(a) = width(b) ∧

width(b) = height(b) ∧

dotprod(vec1(a),vec2(a))=0;

By symmetry of equality this implicitly defines =(RECTANGLE, RHOMBUS) as well.

This construct implies the existence of types RHOMBUS∪RECTANGLE and RHOMBUS∩RECTANGLE.

If an implementation is defined on types RHOMBUS and RECTANGLE then the compiler is able to generate an implementation for type RHOMBUS∪RECTANGLE using a tagged union and defining

bool ← =(RHOMBUS∪RECTANGLE, RHOMBUS∪RECTANGLE)

in terms of

bool ← =(RHOMBUS,RHOMBUS)

bool ← =(RECTANGLE,RECTANGLE)

bool ← =(RHOMBUS, RECTANGLE)

bool ← =(RECTANGLE, RHOMBUS)

This is achieved using conditional testing of the tags in the tagged unions of the two operands and calling one of the four specialisations of equality.

## Functions on union types

Consider that the following *overloads* are defined:

T4 ← f(T1)

T5 ← f(T2)

T6 ← f(T3)

If T4∪T5∪T6 exists and the disjoint union T1∪T2∪T3 exists, we assume these overloads can in fact be interpreted as *implicitly* defining the function with signature

T4∪T5∪T6 ← f(T1∪T2∪T3)

## Defining an isomorphic type

We can define CARTESIAN to be an alias for TUP{ REAL x, REAL y }

type CARTESIAN : TUP{ REAL x, REAL y };

That means that values of type CARTESIAN are values of type TUP{ REAL x, REAL y }.

Alternatively we can define CARTESIAN to be a new and distinct type which is *isomorphic* to TUP{ REAL x, REAL y }

type CARTESIAN isomorphic TUP{ REAL x, REAL y };

Given this definition it would (by default) be assumed CARTESIAN is a root type of a separate lattice, and equality between CARTESIAN and all other types is undefined.

This declaration causes selectors to be defined with signatures

CARTESIAN ← CARTESIAN(REAL x, REAL y)

CARTESIAN ← CARTESIAN(TUP{ REAL x, REAL y })

These both exist because an isomorphic type "inherits" the selectors from the type it is isomorphic to. The latter is the inherited copy selection operator.

Note that isomorphic is a special case of defining a type as a quotient set of an already existing type.

We can also define a constraint. E.g.

type POLAR isomorphic TUP{ REAL r, REAL θ} with r≥0 ∧ θ∈[0,2π);

## Mounting existing root types as subtypes of isomorphic types

Let REAL be an existing root type. Consider that COMPLEX is defined to be isomorphic to TUP{ real REAL, imag REAL } as follows:

type COMPLEX isomorphic TUP{REAL real, REAL imag};

We can now make REAL a subtype of COMPLEX using

subtype REAL r as COMPLEX(r,0);

This declares a total selector with signature COMPLEX ← COMPLEX(REAL r).

## No POSSREPS per se

TTM has an example of type POINT with "possreps" named CARTESIAN and POLAR. We reject the idea that CARTESIAN and POLAR are "things" that are not types. Instead all three are types and we have a concept of declaring isomorphisms between types in order that one type can be regarded as a *representation* of another. We also generalise this by allowing isomorphism to quotient sets to be declared. Allowing multiple isomorphisms allows for alternative representations to be defined.

Consider that we define disjoint types CARTESIAN and POLAR as follows:

type CARTESIAN isomorphic TUP{ REAL x , REAL y };

type POLAR isomorphic TUP{ REAL r, REAL θ };

disjoint union { POLAR, CARTESIAN };

We intend these to be regarded as possible representations of type POINT. Note that we have chosen not to impose a constraint on POLAR. This means we will need to define POINT to be isomorphic to a quotient set on POLAR. There are some advantages to imposing a canonical form. E.g. it avoids a constraint on the selector of POLAR.

Approach 1: Consider that we define an equivalence relation ~ on POLAR∪CARTESIAN as follows:

~(CARTESIAN c1, CARTESIAN c2) { c1=c2 }

~(POLAR p1, POLAR p2) { r(p1)=r(p2) ∧ (r(p1)=0 ∨ (θ(p1) mod 2π)=(θ(p2) mod 2π)) }

~(POLAR p, CARTESIAN c) { x(c)=r(p)cos θ(p) ∧ y(c)=r(p)sin θ(p) }

~(CARTESIAN c, POLAR p) { p~c }

An equivalence relation on a union provides the basis for multiple representations. This allows us to define:

type POINT isomorphic POLAR∪CARTESIAN / ~;

This inherently declares 3-way isomorphisms POINT ↔ POLAR/~ ↔ CARTESIAN/~

Approach 2: We expect type POINT to be isomorphic to CARTESIAN, so we declare:

type POINT isomorphic CARTESIAN;

Now POINT is also isomorphic to a quotient set on POLAR. We therefore declare:

type POINT isomorphic POLAR /

p1~p2  ↔  r(p1)=r(p2) ∧ (r(p1)=0 ∨ (θ(p1) mod 2π)=(θ(p2) mod 2π));

There are two type definitions of POINT. This is acceptable, but we also need to define the equivalence relation between the two representations POLAR and CARTESIAN:

type POINT isomorphic (POLAR union CARTESIAN) /

p~c ↔ x(c)=r(p)cos θ(p) ∧ y(c) = r(p)sin θ(p);

## Morphisms

Definition: Let f : X → Z be a function on X. Let ~ be an equivalence relation on X. We say f *respects* ~ if u~v ⇒ f(u)=f(v)

Motivation: If f respects ~ then we can define f : X/~ → Z. For each q∈X/~, we let f(q) equal f(x) for some x∈q (it doesn't matter which)

Example: Consider that type POLAR has no constraint on component θ, so for each POINT (at the origin or not) there are infinite POLAR values that represent it. Consider that we define the following function on POLAR

POINT ← RotateAboutOrigin( POLAR p, REAL angle ) ::

POINT( POLAR( r(p), θ(p) + angle ) );

Claim: RotateAboutOrigin respects the equivalence relation on POLAR used to define the POINT quotient set. Therefore we can tell the system to automatically generate the implementation of an overload of RotateAboutOrigin defined on POINT to give a function with signature

POINT ← RotateAboutOrigin( POINT p, REAL angle )

Similarly a function that calculates a canonical form on POLAR will also respect the equivalence relation

POLAR ← GetCanonicalForm( POLAR p ) ::

< implementation > ;

and therefore an overload can be made available on POINT:

POLAR ← GetCanonicalForm( POINT p )

Evidently there's a very elegant way of defining operators on quotient sets, with the potential to have a straightforward syntax. Note that the technique can be extended to functions with multiple arguments. It only seems necessary to have a means of flagging which operators are morphisms. Actually it seems appropriate to flag which input parameters the operator is a morphism on.

The D&D approach to write the function RotateAbountOrigin would be comparatively messy (having to deal with the canonical representation, and therefore having to deal with the origin specially, and also comply with a constraint on the valid range of θ).

POINT ← RotateAboutOrigin( POINT p, REAL angle )

{

( r(p)=0) ? POINT(0,0) : POINT( r(p), (θ(p) + angle) mod 2π );

}

## Defining a type as equivalence classes

However, I find it less mysterious to think of the possrep name as a functor and the values of the possrep components as the parameters of a function invocation. So for example there might be an instance of a possrep in computer memory associated with the following selector invocation:

rational(1,2)

This is interpreted as a function invocation that maps a pair of integers (1,2) to a single rational 1/2.

Functions are not necessarily bijective. The lack of an inverse is what allows for "hiding information from above" as it were. Putting it another way, the function can be regarded as a canonical projection map that gives you equivalence classes over representations. In mathematics the idea of quotient sets is one of the most important ways to create new interesting sets from existing sets. For that reason I'm very suspicious of TTM requirement that selectors be bijective. For example, how can one represent the equivalence classes according to graph isomorphism, without using abstract identifiers for the nodes? Abstract identifiers are "private" identifiers used in the representation of a value that can be changed without changing the overall value that is represented.