Ballerina Type System

James Clark

Enterprise Service Bus

- Workhorse of enterprise integration
- XML-based DSL for working with network services
  - ESB products typically include graphical editor
- Uses components written in Java
- Real-world applications often require a combination of the DSL and Java
- Typically licensed on a per-server basis
- Overall not a good fit for the cloud era

Language concept

- Replace Java/XML DSL combo with a single programming language
- A proper programming language with everything expected of a modern programming language
- Batteries included: libraries, package manager, build system, test system, documentation system, IDE support
- Cloud native: designed from the outset for the cloud
- Should be easy for programmers familiar with Java/C#/JavaScript to get started
- Not a JVM language

Business model

- Developed by WSO2
  - founded 2005, current headcount ~850, engineering mostly in Sri Lanka
  - enterprise integration, open source
- Ballerina language free
  - including "batteries"
- Make money off cloud service built on top of Ballerina
- Long-term project

Type system goals

- Describe the types of variables and functions used within the program
- Allow mutation in a similar style to Java or JavaScript
- Describe the data that network services send and receive
  - Good fit with JSON is important
- Cognitive load that the type system features impose on the programmer must be justified by the assistance that they provide in writing correct, maintainable programs
  - cannot expect the programmer to invest a lot of time mastering a complex type system in order to write a 100-line program

Big idea: semantic subtyping

- Not novel: pioneered by XDuce and CDuce
- Types denotes sets of values
- Subtyping relationship on types defined by subset relationship on the sets they denotes
- Works well for describing tree-structured data such as JSON
- Presents implementation challenges
- Implies structural typing

Basic types and unions

```
boolean b = true;

int n = 42;

boolean|int v1 = false;

boolean|int v2 = 17;
```

- Universe of values is partitioned into *basic types*
- Semantics of operations on a value determined by basic type of value
- Most important kind of type descriptor is one that corresponds to a complete basic type
- `boolean` basic type contains two values: true and false
- `int` basic type represents 64-bit signed integers: contains 2^{64} values
- Union type descriptor describes a type as a union of two other types

Type definitions

```
type B boolean;

type I int;

type BI boolean|int;
```

- Type definitions give names to types
- Like typedef in C
- Name is not part of the type

Conditional type narrowing

```
type BI boolean|int;

function toInt(BI v) returns int {
  if v is int {
    return v;  // v is int
  }
  else if v {  // v is boolean
    return 1;
  }
  else {
    return 0;
  }
}
```

- Programs typically work with a union by using the `is` operator
- When a variable is used with `is` in a condition, the type of the variable is narrowed
- Kotlin and TypeScript have similar features

Singletons

```
type ONE 1;

const R = 0;
const G = 1;
const B = 2;
type RGB R|G|B;

RGB color = R;


const T = true;
const F = false;
// Completely equivalent to boolean
type B T|F;
```

- ONE is a singleton type: type that denotes a set with exactly one member
- `const` declaration gives a name to a value known at compile-time
- Name defined with const can be used in two ways
  - as a type in a type descriptor context
  - as a value in an expression context
- Union can be used with singleton type just as with any other type

Nil and optional

```
// T1 and T2 are equivalent
type T1 int?;
type T2 int|();

function f(int? v) returns int {
  if v == () {
    return 0;
  }
  else {
    return v; // v has type int
  }
}
```

- Nil type contains a single value `()`
- Optional type is represented as union with nil
- Functions that do not explicitly return a value return nil
- Conditional type narrowing happens with `==`, `!=` as well as `is`

string and enum

```
type Operator "+"|"-"|"*"|"/"|"%";

Operator op = "+";
string s = op;


enum Color { RED, GREEN, BLUE }

// equivalent to

const RED = "RED";
const GREEN = "GREEN";
const BLUE = "BLUE";
type Color RED|GREEN|BLUE;
```

- string type - immutable sequence of Unicode code points
- singleton strings are used for enumerations
- `enum` declaration provides a shorthand

Floating point: values vs shapes

```
float x = 1.0;

const ONE = 1.0d;

ONE d1 = 1.0d;
ONE d2 = 1.00d;

d1 == d2 // true
d1 === d2 // false
```

- `float` type is 64-bit binary floating point
- `decimal` type is 128-bit decimal floating point
- Decimal values include precision
- == operator for decimal ignores precision
- Shape is equivalence class defined by == equivalence relation
- Types are actually sets of shapes rather than sets of values
- === operator tests for identical value
- Similar issue with `float` zero: +0 and -0 are == but not ===

Lists

```
int[] x = [1, 2, 3];

type Location [string, int];

Location loc = ["foo.bal", 17];

// Relies on 0 having singleton type
string file = loc[0];
int line = loc[1];
int k = 1;
string|int v = loc[k];
```

- List basic type represents an ordered list of values
- Array type descriptor describes a list using a single type for all members
- Tuple type descriptor can specify separate type for each member
- Tuples can end with repeatable member e.g. `[T,R...]`
- `T[]` equivalent to `[T...]`
- Array and tuple types are two ways of describing the same values: fits with JSON, which has single syntax for ordered list of values

List subtyping

```
type Coord [float, float];
Coord c = [1.0, 1.0];

type OptCoord [float?, float?];
// Coord is a subtype of OptCoord
OptCoord oc = c;

type Location [string,int];
// Location is a subtype of SI
type SI (string|int)[];

// T1 and T2 are equivalent
type T1 [int|string,int|string];
type T2 [int,int]|[string,string]
        |[int,string]|[string,int];
```

- Array and tuple types are covariant in their member types
- Obvious when you think in terms of sets of values
- A tuple is a subtype of an array of the union of its member types

Mappings

```
map<int> countryCode = {
  US: 1,
  UK: 44,
  TH: 66
};

type Person record {|
  record {|
     string first;
     string last;
  |} name;
  int id;
|};

Person p = {
   name: {
      first: "James",
      last: "Clark"
   },
   id: 123
};
```

- Mapping type represents mapping from strings to values
- `map` type descriptor describes mapping using single type for all members
- `record` type descriptor describes mapping using separate type for each member
- Map and record types are two ways of describing the same values: fits with JSON which has single syntax for maps and records

Optional fields

```
type Person record {|
   string name;
   int yearOfBirth?;
   string countryOfResidence?;
|};

Person p = {
   name: "James Clark",
   countryOfResidence: "Thailand"
};
```

- Fields can be optional

Open records

```
type Person record {|
  string name;
  int id;
  string...;
|};

Person p = {
   name: "James Clark",
   id: 123,
   "preferredBeverage": "coffee"
};

// T1 and T2 are equivalent
type T1 map<int>;
type T2 record {| int...; |};
```

- Records can be open, allowing fields other than those named
- The quotes are required in the mapping constructor for extra fields to avoid typos with optional fields

Recursive types

```
type LL record {|
   int value;
   LL? next;
|};

type json ()
          | boolean
          | int
          | float
          | decimal
          | string
          | json[]
          | map<json>;


type Bad int|Bad; // invalid
```

- Types can be recursive
- Recursive reference must traverse a type constructor
- Denotes infinite set of values
- Built-in recursive `json` type corresponds to values that can be represented in JSON syntax

anydata type

```
type anydata ()
              | boolean
              | int
              | float
              | decimal
              | string
              | xml
              | anydata[]
              | map<anydata>
              | table<map<anydata>>;

// R1 and R2 are equivalent
type R1 record { string name; };
type R2 record {|
   string name;
   anydata...;
|};
```

- `anydata` represents "plain old data" - data independent of any program
- `==` operator is defined for anydata
- anydata = json + tables + xml
- `xml` is a sequence type (like string) - similar model to XQuery
- `table` is similar to an array of records that allows records to be looked up by key that is part of the record
- `record { }` is shorthand for a record open to `anydata`

Types not in `anydata`

- error - error handling in Ballerina is based on returning error values
- function - module level functions and closures
- object - combines fields and methods
- typedesc - runtime type
- future - used for concurrency
- handle - used for FFI
- any - any type other than error

Object types

```
type Incrementable object {
   function increment();
};

class Integer {
   int n = 0;
   function increment() {
     self.n += 1;
   }
}

Incrementable inc = new Integer();
Incrementable inc = object {
   int n = 0;
   function increment() {
     self.n += 1;
   }
};
```

- Object types work uniformly with other basic types
- Service objects and client objects are using for providing and consuming network services
- Parameter/return types of methods on service/client objects describe format of network messages

Distinct types

```
type X distinct object {};
type Y distinct object {};


type IoError distinct error;
type IllegalArgError distinct error;
```

- object types and error types can both be `distinct`
- Each occurrence of `distinct` in a source module has a unique id, which includes the id of the module
- Values belonging to a distinct type are tagged with the distinct type's unique id
- Provides functionality of nominal typing within a structural typing framework
- Similar concept to *branded* in Modula 3

Mutation

Mutable structures: shape vs value

```
string[] x = ["hello"];
// y == x && y === x
string[] y = x;
// x == z && x !== z
string[] z = ["hello"];
// y[0] is changed, but z[0] is not
x[0] = "goodbye";


type LL record {|
  LL? next;
|};

LL ll = { next: () };
ll.next = ll; // cycle
```

- Mutation means values have an identity
- === and == are different
- === means stored in same location
- Two structures are == if they have the same keys and values for every key is ==
- Type is set of shapes, where shape is equivalence class under ==
- For structures, type is effectively a set of trees
- A graph with a cycle has s shape that is an infinite tree

Aliasing + mutation + covariance = problem

```
string[] v1 = ["s"];   // 1: create a list

string?[] v2 = v1;     // 2: OK because of covariance

// v2 now refers to the same structure as v1

v2[0] = ();            // 3: OK because string? allows nil

string s = v1[0];      // 4: Type of v1[0] should be string but it's not!
```

Solution: inherent types

- A mutable structural value includes an *inherent type*
- The inherent type constrains how the value can be mutated
- Constraint enforced at runtime
  - Conscious trade-off to reduce complexity in the type system
  - Type system still provides compile-time guarantees e.g. v having type int[] guarantees that when you get a member from v it will have type int
  - Compile-time guarantees do not extend to stores e.g. v having type int[] does not guarantee that you can store a value of type int in v
- Similar to how Java arrays work

Inherent type violations are runtime errors

```
string[] v1 = ["s"]; // Compile-time context for list constructor requires string[]
                     // Causes constructed value to have inherent type string[]

// This would be compile-time error
// v1[0] = ()

string?[] v2 = v1;   // v2 has static type string?[] but refers to value
                     // with inherent type string[]

v2[0] = ();          // Mutating member 0 to have value () is incompatible with
                     // inherent type of v2, so results in runtime error
```

Inherent type complicates things

- Matching on value and matching on type are different operations
- Conversion from json to user-defined type cannot be done as a downcast
- Type narrowing works unintuitively with mutable values
- Programmer model is more complex than types are sets of values

Value match vs type match

Two relationships between a mutable structural value and a type

- A value v *looks like* a type T iff the current shape of v is a member of the set of shapes denoted by T - value match
- A value v *belongs to* a type T iff v will always looks like T no matter how v is mutated - (inherent) type match

match statement

```
json employee = {
   type: "tech",
   id: 1234
};

match employee {
  { type: "tech" } => {
     techCount += 1;
  }
}
```

- `is` operator does type match
- `match` statement does value match
- More powerful version of a `switch` statement

Converting from `json`

```
// Built-in function
function fromJsonWithType(
     json v,
     typedesc<anydata> t = <>)
   returns t|error;

// Use like this
type Point record {|
  float x;
  float y;
|};

json j = { x: 1, y: 2 };
Point|error p = j.fromJsonWithType();
```

- `fromJsonWithType` constructs a new value which is (roughly) equal to v but has the inherent type t
- Also does numeric conversions
- `typedesc<T>` is value representing a type that is a subtype of T
- The type of the return value depends on the *value* of the t argument (dependent typing)
- Value of t argument defaulted from contextually expected type
- Usually this is done automatically based on the declared parameter types of methods of service objects

Readonly type goals

- Foundation for concurrency safety
- Reduce negative impact of inherent types
- Enable table datatype
- Don’t complicate the language for beginners

Immutable values

- Structural values (lists and mappings) can be constructed as immutable
- Immutable values cannot be mutated after construction
- Immutability is deep: members of immutable lists and mappings are required to be immutable
- Some basic types are always immutable: nil, boolean, int, float, decimal, string, error, function
- Immutable structures do not need an inherent types: *belongs to* is the same as *looks like*

`readonly` type

```
readonly x = 1;

// v is constructed as immutable
readonly & int[] v = [1, 2, 3];

type Point readonly & record {
  int x;
  int y;
};

// p is constructed as immutable
Point p = { x: 1, y: 2 };
```

- Value belongs to `readonly` type only if it is immutable
- Works in conjunction with intersection operator &
- Mapping and list constructors construct immutable values when contextually expected type is readonly

What `readonly` does and doesn't do

- `readonly&T` is a subtype of `T`
- A type such as `any[]` says nothing about mutability
- An attempted store to a member of immutable structure may be detected at runtime not compile-time
- In C, the `const` in a parameter `const T*` is a constraint on the *callee* not to mutate via that pointer
- In Ballerina, the `readonly` in a parameter `readonly&T` is a constraint on the *caller* to provide a value that cannot be changed

Semantics of `readonly` are designed for concurrency

- Value being readonly guarantees that it can never be mutated
- Further guaranteed that no value reachable through a readonly value can be mutated
- When a variable has type readonly, we know at compile-time that the value can be safely passed to a function running on a separate thread

`isolated` functions and objects

- Goal with concurrency safety is to determine when it is safe to run a function on a separate thread
- A function defined as `isolated` can access mutable data only through its parameters: similar to pure function but specialized for concurrency
- `isolated` for functions complements `readonly` for data
- An object defined as `isolated` encapsulates its mutable state and provides compile-time guarantee that all concurrent access to that state is locked
- Within a module, `isolated` can be inferred: this allows the compiler to identify when services can safely be run in parallel and guide the user in adding the locks needed to enable this

Narrowing problem

```
type N [int];  // 1-tuple
type S [string];
type NS [int|string];

function f(N|S x) returns string {
   if x is N {
     return x[0].toString();
   }
   else {
     return x[0]; // COMPILE ERROR
   }
}

function g() {
   NS x = [42];
   f(x);
}
```

- `NS` is equivalent to `N|S`
- In the `else` branch, all we know is that the inherent type of x is not a subset of N
- Does this imply the inherent type of x is a subset of S?
- No: $(X \subseteq N \cup S) \wedge (X \not\subseteq N) \not\Rightarrow X \subseteq S$
- This is bad: users will be surprised to get a compile error here
- Unavoidable consequence of combination of semantic subtyping and mutability

Narrowing with `readonly`

```
type N readonly & [int];  // 1-tuple
type S readonly & [string];
type NS readonly & [int|string];

function f(N|S x) returns string {
   if x is N {
     return x[0].toString();
   }
   else {
     return x[0]; // Ok
   }
}

function g() {
   NS x = [42];
   f(x);
}
```

- Everything works properly with readonly
- `x is N` will be true: it is testing whether the shape of x is a member of N

`table` type

```
type Sym record {
   readonly string name;
   Value value;
};

type SymTab table<Sym> key(name);
```

- Works uniformly with lists and mappings: record can be a member of multiple tables
- Maintains the invariant that each member of the table is uniquely identified within the table by its key
- Key can come from one or more fields
- Key type can be any subtype of anydata
- Key fields must be readonly: both the field and the value stored in the field are immutable
- Powerful enough to make it unnecessary to have application-specific collection types in most cases

Subtyping algorithm overview

- To test whether S is a subtype of T, test if set difference of S and T is empty
- To test whether a type T is empty:
  - split up into disjoint sets, each a subtype of a single uniform type
  - basic types that are sometimes readonly are split into readonly/read-write uniform types
  - test whether each of these sets is empty
- Subtypes have representation specific to uniform type
  - must be closed under union, intersection difference
- Easy for simple types: e.g. integers are represented as a list of ranges
- Structures are represented as binary decision diagram representing logical combinations (and, or, not) of atomic structures in disjunctive normal function
- Testing emptiness for structures searches for a way it can be non-empty

Implementation status

- Current generation: jBallerina
  - Written in Java
  - Compiles to JVM bytecode
  - Evolved from when type system was very different: implements syntactic approximation to semantic subtyping
- Next generation: nBallerina
  - Written in Ballerina (bootstrap with jBallerina)
  - Compiles to LLVM; eventually to JVM also
  - Initial implementation focus is semantic subtyping
  - Will take several years before it can fully replace jBallerina
- Plan to backport semantic typing implementation

Future type features

- Generics
- Refinement types
  - Regular expressions
- Negation !T

Further information

https://ballerina.io/ Ballerina web site

G. Castagna, Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers), 2020 https://arxiv.org/abs/1809.01427

https://github.com/ballerina-platform/nballerina Implementation of semantic subtyping for Ballerina in Ballerina