Notes - 11/9

Friday, November 9, 2012 10:05 AM

[In 41/2731 at 10:00 today.]

Agenda:

- Generics update
    - Inference
        - Lambda arguments and fixing
        - Matching

foo<T,U>(x: T, y: U>): Pair<T,U>
foo(1, "hello") // T must be number, U must be string

sum<T>(items: List<T>): T
var myList = new List([1,2,3]);
sum(myList)

Thought: Inference is "match" then if that succeeds "infer"
Thought: Can "match" be "assignment compatible to T => any in target including all signature type parameters"

var f: <T>(x: T, y: U): void;
Var g: <T,U>(x: T, y: U): void;

f = g; // okay
g = f;

Thought: starting point, signature type parameters are any in assignmnent compatibility

```
List<T> {
 add: {
  (item: T): void;
  (items: T[]): void;
 }
}
```

```
FooList {
 add: {
  (item: Foo): void;
  (items: Foo[]): void;
 }
}
```

```
/////////  PROMISE ////////////
// As discussed in design meeting
//  (1) we don't yet know of any cases where this fails
//  (2) seems we need the four overloads of then to deal with success or failure being T
or Promise<T>
//  (3) progress should be added
interface Promise<T> {
  then<U>(success?: (x: T) => U, failure?: (x: any) => U): Promise<U>;
  then<U>(success?: (x: T) => U, failure?: (x: any) => Promise<U>): Promise<U>;
  then<U>(success?: (x: T) => Promise<U>, failure?: (x: any) => U): Promise<U>;
  then<U>(success?: (x: T) => Promise<U>, failure?: (x: any) => Promise<U>): Promise<U>;
  done(success?: (x: T) => any, failure? (x: any) => any): void;
}


var p = {
  then(success?: (x: string) => any): any;
  //done
}


/////////  WINRT /////////////
// Basic:
//  (1) just a handful of generic types
//  (2) no generic methods
interface Windows.Foundation.Collections.IIterable<T> {
    first(): Windows.Foundation.Collections.IIterator<T>
}
interface Windows.Foundation.Collections.IIterator<T> {
    current: T;
    hasCurrent: bool;
    moveNext(): bool;
    getMany(): T[];
}
interface Windows.Foundation.Collections.IVector<T> extends T[] {
    size: number;
    getAt(index: number): T;
    indexOf(value: T): { index: number; returnValue: bool; };
    getMany(startIndex: number): T[];
    setAt(index: number, value: T): void;
    insertAt(index: number, value: T): void;
    removeAt(index: number): void;
    removeAtEnd(): void;
    clear(): void;
    replaceAll(items: T[]): void;
}
```

- Examples
    - See aside
- Overloading on constants
- Design backlog

- … for arguments
- Static initialization (#74)
- Thinking on how to approach async via generators (#38)
- String interpolation
- Type-only mixins
- Decorations
- Update on Fundules, clodules
- SkyDrive feedback – implications for future thinking?

- Others?

```
interface Windows.Foundation.Collections.IMapView<K,V> {
    size: number;
    lookup(key: K): V;
    hasKey(key: K): bool;
}
interface Windows.Foundation.Collections.IMap<K,V> {
    size: number;
    lookup(key: K): V;
    hasKey(key: K): bool;
    getView(): Windows.Foundation.Collections.IMapView<K,V>;
    insert(key: K, value: V): bool;
    remove(key: K): void;
    clear(): void;
}

/////////  ES6 Iterators ////////////
interface Iterator<T> {
  next(): T;
}
interface Iterable<T> {
  iterator(): Iterator<T>;
}
// This guy is hard
interface Generator<T> extends Iterator<any> {
    send(x: any): any;
}
var f: () => Generator<string> = function *() {
    var data = yield $.ajax(url);
    $('#result').html(data);
    var status = $('#status').html('Download complete.');
    yield status.fadeIn().promise();
    yield sleep(2000);
    status.fadeOut();
    return "hello";
}
var p: Promise<string> = spawn(f);

/////////  KNOCKOUT ////////////
// (1) Generic "Observable" interface the central object of KnockOut
// (2) Nice interaction of generics and interface call signatures
// (3) Interesting need for 'generic this' for chaining of sets
declare module ko {
    export interface Observable<T> {
        (): T;
        (value: T): void;
        <O>(this: O, value: T): O; // Is this legit?
        subscribe((newValue: T) => void ): {
        dispose(): void
    };
    }
    export function observable<T>(value: T): Observable<T>;
    export function applyBindings(viewModel: any): void;
    export function computed<T>(() => T, owner: any): Observable<T>;
    export function computed<T>({ read: () => T; write?: (value: T) => void; owner?: any}): Obs
    export function isComputed(o: any): bool;
    export function isObservable(o: any): bool;
    export function observable(o: any): bool;

}

var o = {
   x: ko.observable("hello"),
   y: ko.observable(5),
   z: ko.computed(function() {
      return this.x + this.y;
   })
}

o.x("goodbye").y(6);


/////////  UNDERSCORE ////////////
// Looks to be fairly standard LINQ-like library from the generics standpoint
// (1) Iteration over both arrays and objects (ordered string keyd dictionaries)
// (2) Not yet modelled below - API allows chained mode _([1,2,3]).map(f).filter(g)

_.each({a: 1, b: 4}, function(n) { return n + 1; });
// Note:

interface Underscore {

    each<T>(
          list: T[],
          iterator: (element: T, index: number, list: T[]) => any,
          context?: any): void;
     each<T>(
          obj: {[x: string]: T},
          iterator: (value: T, key: string, object: {[x: string]: T}) => any,
          context?: any): void;
     map<T, U>(
          list: T[],
          iterator: (element: T, index?: number, list?: T[]) => U,
          context?: any): U[];
     map<T, U>(
          obj: {[x: string]: T},
          iterator: (value: T, key?: string, object?: {[x: string]: T}) => U,
          context?: any): U[];
    //?
    invoke(list: any[], methodName: string, ...arguments: any[]): void;
    pluck<T,U>(list: T[], propertyName: string): U[];
```

```
        max(list: number[]): number;
        max<T>(
              list: T[],
              iterator: (element: T, index: number, list: T[]) => number,
              context?: any): number;
        groupBy<T>(
              list: T[],
              iterator: (element: T, index: number, list: T[]) => string,
              context?: any): { [key: string]: T[]; };
        }

    _.groupBy([1,2,3, 19], function(x) { return x.toString()[0]; })

    {"1": [1,19], "2": [2], "3": [3] }


    (<number[]>_.pluck(["a","b","c"], "length")) => [1,1,1]
    _.pluck<string, number>(["a","b","c"], "length") => [1,1,1]
```

```
// Questions:
// (1) Is the 'string' overload required, or optional?
// (2) What about case-insensitivity?
// (3) Are RegExps a reasonable extension here?


interface Document {
   //createElement(name: string): Element;
   createElement(name: /$img/i): HTMLImageElement;
   //createElement(name: 'div'): HTMLDivElement;
   // AS IF
   //createElement(name: enum{"img"}): HTMLImageElement;
}
document.createElement('img').src;



enum Colors { "red", "green", "blue" }

switch (s.toLowerCase()) {
   case "red": return 1;
}


interface EventSource {
   addEventListener(name: string, listener: (e: Event) => void): number;
   addEventListener(name: 'mousemove', listener: (e: MouseEvent) => void): number;
   addEventListener(name: 'mousedown', listener: (e: MouseEvent) => void): number;
   addEventListener(name: 'mouseup', listener: (e: MouseEvent) => void): number;
   addEventListener(name: enum{'mousemove', 'mouseup', 'mousedown'}, listener: (e: MouseEvent) =>
void): number;
}
document.body.addEventListener('mousemove', function (e) {
   console.log(e.clientX);
})
```

```
declare module FB {
    export interface Response {
      error?: {message: string; type: string; code: number; }
    }
    export interface User extends Response {
    }
    export interface Friends {
        data: {name: string, id: number;}
    }
    export function api(call: string, callback: (response: any) => void );
    export function api(call: /$\/me\/friends/i, callback: (response: Maybe<User>) => void );
    export function api(call: /$\/me/i, callback: (response: Maybe<User>) => void );
}
FB.api('/platform', function(response) {
  alert(response.company_overview);
});


FB.api('/me/picture', function(response: Picture) {
  if(response.error) {
  } else {
  }
});
```