Notes - 6/15: SENT

Wednesday, June 15, 2011 4:16 PM

Implementation

New class syntax implemented

Type inference

```
function foo() {
  var x = {};
  x.a = "hello";
  x.b = 5;
  return x;
}
```

Option: Open interfaces.

Decision: No change.

Statics and modules

{ invoke(s: string) : void }
(s: string) => void

Interface IPointFactory
{
(x: number, y: number) : Point
initFactrory();
origin: Point
}

Public test() {
Var f = { (x: number, y: number): number }
}

Var f : { (x: string): IElement; (n: number): IElement };

Decision:

- Interfaces can have one or more call signatures.
-

4 types of types:

- Classes

Interface IFoofy {
foo(): void;
}

Interface ICallback {
foo(x: number, y:number) :void;
foo(s: string) : void
}

1. Notes for 06/15/2011

Agenda

- Implementation Status
- Type inference
- Unifying interface and function types
- Core library typing

Attendees: SteveLuc, AndersH, MadsT, ChuckJ, LukeH, ToddPro

1. Implementation Status

Substantial progress on the implementation. The new class syntax is fully implemented in the self-hosting Strada implementation. The Strada implementation is also completely divorced from the initial C++ bootstrap compiler, and can bootstrap itself going forward. Several large JavaScript codebases are able to pass through the Strada compiler successfully, including Corsica, Sunspider and most Chakra tests.

2. Type Inference

We discussed the issue of type inference for {} again:

```
function foo() {
  var x = {};
  x.a = "hello";
  x.b = 5;
  return x;
}
```

Under current rules, x gets the empty interface type, which causes x.a = “hello” to report an error. We expect this to be a common enough case that we’re concerned about the type system not being able to

Fields are:

- Not overloadable

Question: What would you need to do to implement an overloaded method?

Question: Upper case infers newable signature?

```
module global
{
 public String:
  {
   new(s: string): String;
   (x: any): string;
   fromCharCode(i: number) : string
  }
 interface String
 {
  length: number
  charCodeAt(i: number): number
 }
}
```

```
class String(s: string)  {
// What about behaviour as String(...) - special case for built-ins?
// Also overloaded to allow "new String()" - no way to represent?

 public length : number

 //public constructor
 //This prototype property is on all classes,
 // but I guess cannot be declared explicit

 public toString() : string

 public valueOf() : string

 public charAt(pos: number) : string

 public charCodeAt(pos: number) : number

 public concat(...strings: string) : string
 // varargs?

 public indexOf(searchString: string, position: number) : number
 // arity-based overloading
 // public indexOf(searchString: string) : number

 public lastIndexOf(searchString: string, position: number) : number
 // arity-based overloading
 // public lastIndexOf(searchString: string) : number

 public localeCompare(that: string) : number

 public match(regexp: RegExp) : string[]

 public replace(searchValue: string, replaceValue: string) : string
```

handle it without user annotation.

We considered approaches that involve “open types”, which would allow type checking to change the type as it processes the function body seeing mutations on the open typed value. There are some big question marks around this though (when to close, cross-function mutation, tooling experience when type changes as they user types code, etc.)

Decision: No change for now, user needs to add type annotation to this sort of code, we’ll see how much of a problem this is in practice.

3. Unifying Interface and Function Types

One issue we’ve seen in typing many existing libraries is the occurrence of functions and constructors with properties directly on the function object, as in statics on constructors. Our function types have so far been a separate universe from interface types, but this isn’t a great fit for common JavaScript patterns.

We considered a model for function types that is much closer to JavaScript’s model for functions – treat them as any other interface, but with a “special” call definition:

Previous function type syntax:

```
(s: string) => void
```

New option for specifying a function type – an anonymous interface with a call signature:

```
{ (s: string) : void }
```

This can also extend quite cleanly to overloaded function signatures, which were note

```
  // more complex overloading
  //public replace(searchValue: RegExp, replaceValue: string) : string
  //public replace(searchValue: string, replaceValue: Function) : string
  //public replace(searchValue: RegExp, replaceValue: Function) : string

  public search(regexp: RegExp) : number
  //effectively overloaded
  //public search(regexp: string) : number

  public split(seperator:

  public substring(start: number, end: number)
  //overloaded on arity
  //public substring(start: number)

  public toLowerCase() : string

  public toLocaleLowerCase() : string

  public toUpperCase() : string

  public toLocaleUpperCase() : string

  public trim() : string

  //indexer?

}
module String {

  public fromCharCode(...chars:string) : string

  //public prototype : any
  // This is on all classes,
  //  but I guess cannot be authored explicitly
  // What is it's type?

  //public length : number
  // Thisis on all classes,
  //  but I guess cannot be authored explicitly

}

// Other questions:
//  - What is the rule that allows "foo".indexOf('o') to work?
//  - Is an IString needed, to allow
//  - Can String be inherited from?
//    - Not really possible at runtime (String.call(this) doesn't initialize string
state into the instance)
//    - If so, it can't inherit the indexer

class Object() {
// What about overload "new Object(34)"?
// What about behaviour as a fucntion, Object(34)

  //public constructor
  //This prototype property is on all classes,
  // but I guess cannot be declared explicit

  public toString() : string
```

previously denotable in the type system:

```
  interface ICallback {
    (x: number, y:number)
:void;
    (s: string) : void
  }
```

Moreover, the notion of ‘constructable’ can also be captured more directly in the type system through the same mechanism:

```
 var Image :  {
new(width: number,
height: number): Image }
```

Overall, this feels like a powerful unifying feature in the type system, removing the need for separate ‘kinds’ of types for functions and constructors. It also offers significant flexibility for typing existing code.

Decision: Interface types can have zero or more call and construct definitions, which describe the signatures when a instance if used in call and construct positions respectively, with the syntax as above.

We considered whether this should fully replace the previous function type syntax, or if that should remain as a shorthand. The shorthand is only one character shorter, but it is more evocative of the intent. **No decision on this**, though leaning toward removing the function type syntax shorthand.

Inference for function types would continue as expected:

```
  function foo() { return
3; } infers that foo has type {
(): number }.
```

However, what about this:

```
  function Foo() { }
```

The user almost certainly intends Foo to be a constructor function, as indicated by casing. Should we make the inferred type

```
  //...

}
module Object {

  public getOwnPropertyDescriptor(o: Object, p: string) : IPropertyDescriptor

  //...
}

interface IPropertyDescriptor {
  public enumerable: bool
  public configurable: bool
}
interface IAccessorPropertyDescriptor : IPropertyDescriptor {
  public get() : any
  public set(v:any) : void
}
interface IDataPropertyDescriptor : IPropertyDescriptor {
  public value : any
  public writable : bool
}
```

From: Luke Hoban
Sent: Wednesday, June 15, 2011 11:47 AM
To: Strada Design Team
Subject: Strada Design Meeting Agenda - 6/15/2011

Agenda:

- Update on implementation from Steve
- Statics and modules
- Typing of core libraries – attached initial notes on core and DOM for discussion
- Backlog:
  - Promises typing
  - Top level scope?
  - Uninitialized variables/inference/defaults
  - O[x] with typed objects
  - Enums
  - Global module
  - Type compatibility rules
  - Foreach
  - Class-parameters as privates and inheritance
  - Function type compatibility (omit parameters, more parameters, varargs, etc.)
    Nested modules
  - Any -> number (what gets emitted?)
- Anything else?

have a construct as well as call member based on the casing? No decision yet, though we like the idea.

4. Core Library Typing

We began to revisit the core library typing. Several of the issues we had identified here were addressed by the interface/function type unification decision (behaviour of String(..) as a function, overloading of constructors).

We wrote down a subset of the library typing using the new type syntax:

```
  module global
  {
    public String:
      {
        new(s: string):
String // note the
capital-S String here
        (x: any):
string       // and the
lowercase-S string here
        fromCharCode(i:
number) : string
        substring(strart:
number, end: number):
string

substring(start:number):
string

        // ...

      }
    interface String
    {
      length: number
      charCodeAt(i:
number): number

      // ...

    }

    // ...

  }
```

This offers a nice high-fidelity encoding of the library typing. Several other issues were raised and bumped up on the backlog:

- Varargs
- Integer-valued indexing