comparison of Haskell and Flow

1 Languages

1.1 Haskell

Haskell is mainly characterized by being purely functional and lazy. It has an expressive static type system with an above average focus on soundness. Its expressive type system benefits a lot from being at the forefront of language research as it is the choice of many language researches for trying out new ideas.

1.2 Flow

Flow is mainly characterized by being a statically typed language that is very close to and compiles to JavaScript. It is very similar to TypeScript but has made significantly better design choices with respect to soundness. It was released by Facebook shortly after TypeScript's appearance. It is used heavily in Facebook software.

2 Overview

comparison aspect	Haskell	Flow
support for algebraic data types, pattern matching	support for recursion. tagged sum types. pattern matching syntax.	support for recursion. untagged sum types. idiomatic manual tags. pattern matching via *flow-sensitive typing*.
support for different flavours of polymorphism	parametric polymorphism, deduced type varible instantiation, explicit type varible instantiation with `TypeApplications` language extension. ad hoc polymorphism via type classes.	parametric polymorphism, both deduced and explicit type varible instantiation. object oriented style ad hoc polymorphism.
Error and/or exception handling	pure code raises imprecise exceptions. `IO` code raises synchronous exceptions or throws asynchronous exceptions to other threads.	type `never` without inhabitant deduced for functions guaranteed to throw. exception semantics like Haskell's synchronous exceptions.
Memory management	*generational garbage collector*[Guic]. support for finalizers[Feu20].	JavaScript engine's *mark-and-sweep garbage collector*[Doc22b]
Support for strictness and laziness	lazy by default. strict with `Strict` language extension[Guia].	strict. laziness only manually via anonymous functions without arguments.

3 More extensive discussion of the comparison

3.1 Support for algebraic data types, pattern matching

3.1.1 Haskell

Here is an example of Haskell's support for algebraic data types and pattern matching.

```
1 import Prelude hiding (Bool (..))
2
3 data Boolean = False | True
4 data SomeBooleans = One Boolean | Many Boolean SomeBooleans
5
6 someBooleans :: SomeBooleans
7 someBooleans =
```

```
8     Many False $
9     Many False $
10    One True
11
12  oneBoolean :: Boolean
13  oneBoolean =
14    case someBooleans of
15      One b -> b
16      Many _ bs ->
17        case bs of
18          One b -> b
19          Many b _ -> b
20
21  foldSomeBooleans ::
22    (Boolean -> result) ->
23    (Boolean -> result -> result) ->
24    SomeBooleans ->
25    result
26  foldSomeBooleans one _many (One b) = one b
27  foldSomeBooleans one  many (Many b bs) =
28    many b (foldSomeBooleans one many bs)
```

Code Block 1: algebraic data types in Haskell

Notice the recursive use of `SomeBooleans` in line 4.

Haskell's sum types are *tagged* sum types. They represent a disjoint union. Even if you construct the sum of twice the same type, there is a *tag* distinguishing values as either belonging to the left or the right summand. For example values of a type `data Either = Left Boolean | Right Boolean` look like `Left False`, `Right False`,... `Left` and `Right` are the tags. `False`, `True`, `One`, `Many` are the tags in code block 1.

3.1.2 Flow

In Flow, all boolean, number, and string literals denote a predefined type besides being term level literals. For example there is a predefined type `false`. The only inhabitant of this type is the term level value `false`.

Flow supports algebraic data types and pattern matching via a code idiom that makes heavy use of string literal *singleton types*, which were explained in the previous paragraph, for *tags*.

Here is an example of this code idom.

```
1  type Boolean = "false" | "true";
2  type SomeBooleans =
3    {tag: "one",  value: Boolean} |
4    {tag: "many", head: Boolean, tail: SomeBooleans};
5
6  const someBooleans: SomeBooleans =
7    {
8      tag: "many",
9      head: "false",
10     tail: {
11       tag: "many",
12       head: "false",
13       tail: {
14         tag: "one",
15         value: "true",
16       }
17     }
18   };
```

Notice the recursive use of `SomeBooleans` in line 4.

Flow's sum types (`|` operator) are *untagged* sum types. Therefore, manual tags are commonly added as record fields. The `tag` record field assumes this role in the code block above. Without them, there would be no systematic way to do pattern matching.

Being untagged allows Flow's sum types to be supertypes of their summands too. `boolean | string` is a supertype of `boolean` for example. This is not the case for Haskell's tagged sum types.

Here is an example of how Flow supports pattern matching.

```
1  const oneBoolean: Boolean =
2    someBooleans.tag === "one" ? someBooleans.value :
3    someBooleans.tail.tag === "one" ? someBooleans.tail.value :
4    someBooleans.tail.head;
5
```

```
const foldSomeBooleans =
  <T,>(
    one: (_: Boolean) => T,
    many: (_b: Boolean, _r: T) => T,
    sB: SomeBooleans,
  ): T =>
  sB.tag === "one"
    ? one(sB.value)
    : many(sB.head, foldSomeBooleans(one, many, sB.tail))
```

The ? and : operators are the normal conditional ternary operator of JavaScript[Doc22a]. We could have nested `if` statements just as well.

JavaScript does not have any pattern matching syntax. Flow, unwilling to add term level syntax to JavaScript, support pattern matching via *flow-sensitive typing*. Checks like `someBooleans.tag === "one"` determine the accessability and type of record fields like `value`, `head`, and `tail`. Without that check as for example in `const oneBoolean: Boolean = someBooleans.value;`, Flow would produce an error as `Cannot get ‘someBooleans.value‘ because property ‘value‘ is missing in object type.` This way, pattern matching in Flow reaches the same level of type safety as in a language with syntax support like Haskell.

3.2 Support for different flavours of polymorphism

3.2.1 Parametric polymorphism in Haskell

Haskell supports parametric polymorphism as follows.

```
data Pair a b = Pair {component0 :: a, component1 :: b}

swap :: Pair a b -> Pair b a
swap (Pair {component0, component1}) =
  Pair {component0 = component1, component1 = component0}
```

The parametrically polymorphic function `swap` could then be used with explicit type variable instantiation as follows.

```
{-# language TypeApplications #-}

pair :: Pair Bool String
```

```haskell
pair = Pair {component0 = False, component1 = ""}

test :: Pair String Bool
test = swap @Bool @String pair
-- == Pair {component0 = "", component1 = False}
```

This requires the `TypeApplications` language extension[Guib].

Haskell can usually deduce the type variable instantiation too.

```haskell
pair :: Pair Bool String
pair = Pair {component0 = False, component1 = ""}

test :: Pair String Bool
test = swap pair
-- == Pair {component0 = "", component1 = False}
```

Haskell compilers implement parametric polymorphism by boxing "every value, so all polymorphic operations can be expressed in terms of operations on pointers" ([KS23]).

3.2.2 Parametric polymorphism in Flow

Flow supports parametric polymorphism as follows.

```flow
type Pair<T, U> = {component0: T, component1: U};

const swap =
  <T, U>({component0, component1}: Pair<T, U>): Pair<U, T> =>
  ({component0: component1, component1: component0});
```

The parametrically polymorphic function `swap` could then be used with explicit type variable instantiation as follows.

```flow
const pair: Pair<boolean, string> =
  {component0: false, component1: ""};

const test: Pair<string, boolean> = swap<boolean, string>(pair);
// === {component0: "", component1: false}
```

Flow can usually deduce the type variable instantiation too. But less often than Haskell so.

```
const pair: Pair<boolean, string> =
  {component0: false, component1: ""};

const test: Pair<string, boolean> = swap(pair);
// === {component0: "", component1: false}
```

3.2.3 Ad hoc polymorphism in Haskell

Haskell supports ad hoc polymorphism via its language feature of *type classes* as follows.

```
import Data.List (findIndex)
import Prelude hiding (Eq)

data TwoBools = TwoBools {bool0 :: Bool, bool1 :: Bool}

class Eq a where
  equals :: a -> a -> Bool

instance Eq TwoBools where
  equals a b = bool0 a == bool0 b && bool1 a == bool1 b
```

The ad hoc polymorphic function `equals` could then be used in a parametrically polymorphic function `elemIndex` as follows.

```
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
elemIndex a as = findIndex (\b -> equals a b) as

twoBools :: TwoBools
twoBools = TwoBools {bool0 = False, bool1 = True}

test :: Maybe Int
test = elemIndex twoBools [twoBools] -- == Just 0
```

3.2.4 Ad hoc polymorphism in Flow

Flow supports ad hoc polymorphism using a more object oriented technique, which involves bundling functions with data, as follows.

```
class TwoBools {
  bool0: boolean;
  bool1: boolean;
  equals(a: TwoBools, b: TwoBools): boolean {
    return a.bool0 === b.bool0 && a.bool1 === b.bool1;
  }
  constructor(bool0: boolean, bool1: boolean): void
    {this.bool0 = bool0; this.bool1 = bool1;}
}
```

Code Block 2: object oriented style ad hoc polymorphism in Flow

The function `equals` could then be used in a parametrically polymorphic function `elemIndex` as follows.

```
interface HasEquals<T> {equals(T, T): boolean};

const elemIndex =
  <T,>(a: T & HasEquals<T>, as: Array<T>): number =>
  as.findIndex((b) => a.equals(a, b));

const twoBools: TwoBools = new TwoBools(false, true);

const test: number = elemIndex(twoBools, [twoBools]); // === 0
```

Ad hoc polymorphism can furthermore be achieved in Flow via sum types in some situations because Flow's untagged sum types are supertypes of their summands as explained in section 3.1.2.

```
type TwoBools = {bool0: boolean, bool1: boolean};

type ShowInput = boolean | string | TwoBools;
const show =
  (a: ShowInput): string =>
    typeof a === "boolean" ? (a ? "true" : "false") :
    typeof a === "string"  ? "\"" + a + "\"" :
    "(" + show(a.bool0) + ", " + show(a.bool1) + ")";
```

Code Block 3: ad hoc polymorphism via sum types in Flow

The ad hoc polymorphic function `show` could then be used in a parametrically polymorphic function `showAndCombine` as follows.

```
const showAndCombine =
  <T: ShowInput,>(a: T): [T, string] =>
  [a, show(a)];

const test: [boolean, string] = showAndCombine(false); // [false, "false"]
```

However, this technique stops working quickly when there are multiple arguments of the type `T` that we want the implementation to depend on. The reason is explained in section 4.1.

And even without multiple arguments of the type that we want the behavior to depend on, there is no systematic way of distinguishing the summands of the untagged sum `ShowInput` unless manual tags introduced as exemplified in section 3.1.2. JavaScript's `typeof` operator can only return primitive JavaScript types. `show` could only cover the case for `TwoBools` because the other possible cases of `ShowInput` are primitive JavaScript types.

4 Appendix

4.1 Ad hoc polymorphism via sum types does not work in Flow

Can we apply the technique of ad hoc polymorphism via sum types in Flow, which we used for `show` in code block 3, to `equals` in code block 2? `equals`'s type would become `(EqualsInput, EqualsInput) => boolean` where `EqualsInput`, analogous to `ShowInput` in code block 3, is a large untagged sum type of all the types we want to use `equals` with, like `boolean`, `string`, and `TwoBools`. But then the call `equals(false, "")` would type check although we only want to define `equals` for all pairs of the same type in `EqualsInput`.

Can we maybe use parametric polymorphism to enforce that both arguments are of the same type in `EqualsInput`? Here is a simplified example to demonstrate the problem we run into next.

```
type EqualsInput = boolean | string;
const f =
  <T: EqualsInput,>(a: T, b: T): boolean =>
    typeof a === "boolean" ? b : false; // type error
```

The following type error confirms that `a: boolean` does not imply `b: boolean`.

```
Cannot return `(typeof a) === "boolean" ? b : false` because
string [1] is incompatible with boolean [2]. [incompatible-return]

References:

3:   <T: EqualsInput,>(a: T, b: T): boolean =>
                                   ^ [1]

3:   <T: EqualsInput,>(a: T, b: T): boolean =>
                                       ^ [2]
```

`a` is of type boolean. That is, `a` is either `false` or `true`. So `false` or `true` inhabit `T`. Does that not mean that `T` is `boolean`? It does not because there are many more types inhabited by `false` and `true` as explained in section 3.1.2. All *supertypes* of `boolean` are inhabited by `false` and `true`. `boolean | string` for example.

In other words, `f(false, "");` can actually type check given `f: <T: EqualsInput,>(a: T, b: T) => boolean`. For example by instantiating `T` to `boolean | string`. This problem of `f` might be a surprising consequence of the presence of subtyping for some.

Flow's documentation seems to offer the solution of "intersections of functions" ([Doc]) for this very problem.

```
type TwoBools = {bool0: boolean, bool1: boolean};

const equals:
  ((a: boolean,  b: boolean ) => boolean) &
  ((a: string,   b: string  ) => boolean) &
  ((a: TwoBools, b: TwoBools) => boolean)
  = (a, b) =>
    typeof a === "boolean" ? a === b :
    typeof a === "string" ? a === b :
    equals(a.bool0, b.bool0) && equals(a.bool1, b.bool1);
```

But this code does not type check properly in Flow version 0.197.0. This situation might or might not improve in future versions. But even then would using `equals` in a parametrically polymorphic function incur an unacceptable amount of boilerplate. The entire function intersection type and the pattern match would need to be reperformed for every parametrically polymorphic caller as follows.

```
const elemIndex:
  ((a: boolean, as: Array<boolean>) => number) &
  ((a: string, as: Array<string>) => number) &
  ((a: TwoBools, as: Array<TwoBools>) => number)
  = (a, as) =>
    typeof a === "boolean" ? as.findIndex((b) => equals(a, b)) :
    typeof a === "string" ? as.findIndex((b) => equals(a, b)) :
    as.findIndex((b) => equals(a, b));

const twoBools: TwoBools = {bool0: false, bool1: true};

const test: number = elemIndex(twoBools, [twoBools]); // === 0
```

References

[Doc] Flow Documentation. *Intersection of function types*. URL: https://flow.org/en/docs/types/intersections/#toc-intersection-of-function-types (visited on 01/21/2023).

[Doc22a] MDN Web Docs. *Conditional (ternary) operator*. Dec. 13, 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator (visited on 01/21/2023).

[Doc22b] MDN Web Docs. *Memory management, Memory management*. Dec. 13, 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management#garbage_collection (visited on 01/21/2023).

[Feu20] David Feuer. *‘readFile‘ leaks file descriptors in the presence of asynchronous exceptions*. Dec. 23, 2020. URL: https://gitlab.haskell.org/ghc/ghc/-/issues/19114#note_319906 (visited on 01/21/2023).

[Guia] GHC User Guide. *Language extensions, Strict-by-default pattern bindings*. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/strict.html#extension-Strict (visited on 01/21/2023).

[Guib] GHC User Guide. *Language extensions, Visible type application*. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/type_applications.html (visited on 01/21/2023).

[Guic] GHC User Guide. *Runtime system (RTS) options, RTS options to control the garbage collector, `--copying-gc`*. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/runtime_control.html#rts-flag---copying-gc (visited on 01/21/2023).

[KS23] Gabriele Keller and Tom Smeding. *Concepts of Programming Language Design, Parametric Polymorphism*. Jan. 10, 2023. URL: http://www.cs.uu.nl/docs/vakken/mcpd/2021/website/slides/Polymorphism.pdf (visited on 01/21/2023).