Extraction Refactor Code Actions

Sam Zhou

What is it?

Goals for the resulting code:

- Type checks
- Preserves original semantics
- Nicely formatted

Goal of the code action:

Good user experience

Goal:
The refactored code should type check.

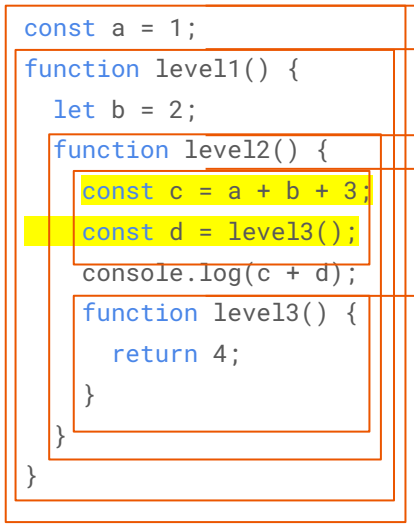
Sources of type errors after naive refactoring

```
const a = 1;
function level1() {
  let b = 2;
  function level2() {
    const c = a + b + 3;
    const d = level3();
    console.log(c + d);
    function level3() {
      return 4;
    }
  }
}
```

```
const a = 1;
function level1() {
  let b = 2;
  function level2() {
    newFunction();
    console.log(c + d);
    function level3() {
      return 4;
    }
  }
}
function newFunction() {
  const c = a + b + 3;
  const d = level3();
}
```

1. Using variables defined elsewhere that falls out of scope after refactoring
2. Escaping local definitions

Eliminating Undefined Variable Type Errors



```
const a = 1;
function level1() {
  let b = 2;
  function level2() {
    const c = a + b + 3;
    const d = level3();
    console.log(c + d);
    function level3() {
      return 4;
    }
  }
}
```

S1	a	a: S1
S2	b	b: S2
S3	c	c: S3, local
local	d	d: S3, local
S4	level3	level3: S3

A variable v will be undefined after refactored to scope S' if:

- v not defined local scope
- v is defined in scope S, and S' > S

Solution: pass these variables as parameters!

Eliminating Escaping Local Definition Type Errors

Escaping local definitions:

A definition that is defined within the extracted statement, but are used outside of it.

Solution:

Return these variables in a object and collect them.

```
function newFunction() {
  // …
  return {a, b, c};
}

const {a,b,c} = newFunction();
```

Adding type annotations

Flow has an ongoing local type inference project that will require type annotations on most function parameters in the future.

To make the code action future proof, we also need to generate type annotations for the parameters and return of function.

Solution:

- Find all the variables and their locations that we need the types
- Grab those types from Typed AST

Adding type annotations: problem with generics

Naively adding type annotations can run into problem with out-of-scope generic type parameters:

```
function logAnything<T>(x: T) {
  console.log(x);
}
```

```
function logAnything<T>(x: T) {
  newFunction(x);
}
function newFunction(x: T) {
  console.log(x);
}
```

Detecting out-of-scope generic type parameters

```
function foo<A, B, C: B>(c: C) {
  function bar<D, E = D, F: E>(f: F) {
    const obj = { c, f };
    console.log(obj);
  }
}
```

Generic type parameters in scope of function foo:
<C,B,A>

Type of obj:
<F:E,E=D,D,C:B,B,A> { c:C,f:F }

Used generic type parameters:
<F:E,E=D,D,C:B,B>

Final new function signature:

```
function newFunction<D,E=D,F:E>(f: F)
```

Misc Other Issues

- Async functions: detect them and make the function async, and await the function call
  - `async function foo() { ... }  await foo()`
- Special control flow statements: return, yield, break, continue, labels:
  - Ban them, since they introduce non-local behavior that can't be handled by function call
  - Support break and continue without label if they are already in a loop within selection
    - `for (...) { break; /* break here is safe */ }`

Goal:
The refactored code preserves original semantics.

How the original semantics can be broken

```
let a = 1;

a = 2;

console.log(a);
```

```
let a = 1;

newFunction(a);

console.log(a);

function newFunction(a) {

  a = 2;

}
```

Due to shadowing, the value of a is no longer correctly changed for log!

Solution:

```
let a = 1;

{a} = newFunction(a);

console.log(a);

function newFunction(a) {

 a = 2;

 return {a};
```

} Detection: a variable that is

- Used outside of selection
- Defined outside of selection
- Has a write inside selection

Goal:
The refactored code should be nicely formatted.

Existing pretty-printer for diff is not good enough

```
function level1() {
  console.log('foo');
  console.log('bar');
  function level2() {
    console.log('baz');
  }
}
```

```
function level1() {
  console.log('foo');
  console.log('bar');
  function level2() {
    (newFunction());
  }function newFunction() {
  console.log("baz");
}
}
```

Why the output code appears bad

How the new code is produced?

1. After diffing old AST and new AST, we get a list of node changes
    a. (Replace, 3:4-4:5, old_node, new_node)
2. For each node change, we emit an editor diff
    a. (3:4-4:5, pretty printed version of new node)

Issues:

- Lack of parent context, so we have to be conservative
    - Don’t know the parent, so we must always add parentheses in most cases
- Lack of indentation context, so we don’t know how many indentations we need
    - Always assuming indentation level is 0

Building a better diff printer

- Main solution: record parents of node changes
- Avoiding unnecessary parentheses: compare indentation level of parent and child
- Correct indentation level: infer indentation level from the parent

Other extraction features

Other supported extraction features

- Extract to methods
  - Similar to extract to functions, but we can only place the extracted method inside the closest class.
  - If the selection contains `this` or `super`, we cannot extract to functions
- Extract to constant/class fields
  - We still need to perform undefined variable analysis to decide where we can put in the constant/class property.
- Extract to type alias
  - Still need to find used but out-of-scope generic type parameters to add to type alias' generic type parameters

User experience

Limitations of Language Server Protocol

So far, all the IDE refactor features are provided by language server protocol.

However, current LSP does not allow user to specify the name of new functions/constant/etc.

LSP extension: snippets support in code actions

Modify the VSCode extension to support a simple form of snippet:

Server: emit snippets

```
const a = ${0:newFunction}();

function ${0:newFunction}() { ... }
```


Client: replace and multi-select

```
const a = newFunction();

function newFunction() { ... }
```

Questions?