Scraph You Boilerplate

A Practical Design Pattern for Generic Programming [3]

Duane Irvin
duane@student.chalmers.se

Room EG-5215A — 16 November 13.15
DAT-315 — The computer scientist in society
Chalmers University of Technology

Apps > Powerpoints

Let's look at how we'd increase all salaries!

Increase salary at company

```
export const increase =
  (k: number) =>
  ({ departments, ...rest }: Company): Company => ({
    ...rest,
    departments: departments.map(increaseDepartment(k)),
  })
```

Increase salary at each department

```
const increaseDepartment =
  (k: number) =>
  ({ manager, subunits, ...rest }: Department): Department => ({
    ...rest,
    manager: increaseEmployee(k)(manager),
    subunits: subunits.map(({ unit, ...rest }) => {
      switch (unit.type) {
        case 'employee':
          return {
            ...rest,
            unit: increaseEmployee(k)(unit),
          }
        case 'department':
          return {
            ...rest,
            unit: increaseDepartment(k)(unit),
          }
        default:
          return switchFallback(unit)
      }
    }),
  })
```

Increase salary for each employee

```
const increaseEmployee =
  (k: number) =>
  ({ salary, ...rest }: Employee): Employee => ({
    ...rest,
    salary: increaseSalary(k)(salary),
  })

const increaseSalary =
  (k: number) =>
  ({ value, ...rest }: Salary): Salary => ({
    ...rest,
    value: value * (1 + k),
  })
```

That's a lot of code! 😭

```ts
export const increase =
  (k: number) =>
  ({ departments, ...rest }: Company): Company => ({
    ...rest,
    departments: departments.map(increaseDepartment(k)),
  })

const increaseDepartment =
  (k: number) =>
  ({ manager, subunits, ...rest }: Department): Department => ({
    ...rest,
    manager: increaseEmployee(k)(manager),
    subunits: subunits.map(({ unit, ...rest }) => {
      switch (unit.type) {
        case 'employee':
          return {
            ...rest,
            unit: increaseEmployee(k)(unit),
          }
        case 'department':
          return {
            ...rest,
            unit: increaseDepartment(k)(unit),
          }
        default:
          return switchFallback(unit)
      }
    }),
  })

const increaseEmployee =
  (k: number) =>
  ({ salary, ...rest }: Employee): Employee => ({
    ...rest,
    salary: increaseSalary(k)(salary),
  })

const increaseSalary =
  (k: number) =>
  ({ value, ...rest }: Salary): Salary => ({
    ...rest,
    value: value * (1 + k),
  })

const switchFallback = (fallback: never) => fallback
```

What if we could?

```
everywhere({
  data: company,
  matcher: isSalary,
  transformer: (salary) => ({
    ...salary,
    value: salary.value * (k + 1),
  })
```

Let's look at a different traverse!

increase Before

```
export const increase =
  (k: number) =>
  ({ departments, ...rest }: Company): Company => ({
    ...rest,
    departments: departments.map(increaseDepartment(k)),
  })

const increaseDepartment =
  (k: number) =>
  ({ manager, subunits, ...rest }: Department): Department => ({
    ...rest,
    manager: increaseEmployee(k)(manager),
    subunits: subunits.map(({ unit, ...rest }) => {
      switch (unit.type) {
        case 'employee':
          return {
            ...rest,
            unit: increaseEmployee(k)(unit),
          }
        case 'department':
          return {
            ...rest,
            unit: increaseDepartment(k)(unit),
          }
        default:
          return switchFallback(unit)
      }
    }),
  })

const increaseEmployee =
  (k: number) =>
  ({ salary, ...rest }: Employee): Employee => ({
    ...rest,
    salary: increaseSalary(k)(salary),
  })

const increaseSalary =
  (k: number) =>
  ({ value, ...rest }: Salary): Salary => ({
    ...rest,
    value: value * (1 + k),
  })

const switchFallback = (fallback: never) => fallback
```

increase After

```
export const increase =
  (k: number) =>
  (company: Company): Company =>
    everywhere({
      data: company,
      matcher: isSalary,
      transformer: (salary) => ({
        ...salary,
        value: salary.value * (k + 1),
      }),
    })
```

Similarily we could reduce all salaries

```
everything({
  data,
  matcher: isSalary,
  query: ({ value }) => value,
  reducer: (a, b) => a + b,
  zeroValue: 0,
})
```

bill Before

```
export const bill = ({ departments }: Company): number =>
  departments.map(billDepartment).reduce((a, b) => a + b, 0)

const billDepartment = ({ manager, subunits }: Department): number =>
  billEmployee(manager) +
  subunits
    .map(({ unit }) => {
      switch (unit.type) {
        case 'employee':
          return billEmployee(unit)
        case 'department':
          return billDepartment(unit)
        default:
          switchFallback(unit)
          return 0
      }
    })
    .reduce((a, b) => a + b, 0)

const billEmployee = ({ salary }: Employee): number => billSalary(salary)

const billSalary = ({ value }: Salary): number => value

const switchFallback = (fallback: never) => fallback
```

bill After

```
everything({
  data,
  matcher: isSalary,
  query: ({ value }) => value,
  reducer: (a, b) => a + b,
  zeroValue: 0,
})
```

What is `everything` and `everywhere`?

Where did `isSalary` come from?

Haskell Before

```haskell
bill :: Company -> Float
bill (Company ds) = sum $ map billDepartment ds

billDepartment :: Department -> Float
billDepartment (Department _ manager subunits) =
  billEmployee manager + sum (map billSubUnit subunits)

billSubUnit :: SubUnit -> Float
billSubUnit (EmployeeUnit employee) = billEmployee employee
billSubUnit (DepartmentUnit department) = billDepartment department

billEmployee :: Employee -> Float
billEmployee (Employee _ salary) = billSalary salary

billSalary :: Salary -> Float
billSalary (Salary value) = value
```

Haskell After

```haskell
bill' :: Company -> Float
bill' = everything (+) (0 `mkQ` billSalary)

billSalary :: Salary -> Float
billSalary (Salary value) = value
```

Key idea

Rethink how we traverse data

$$\Downarrow$$

Pure boilerplate can separated and generated

Prior related work

A new approach to generic functional programming. Hinze, R., 2000 [1]

But poly-typic programming is too strict to be useful

Derivative work

Go beyond `show`, `map` and `reduce`:

"Scrap Your Boilerplate" Revolutions. Hinze, R., and Löh, 2006 [2]

References

[1] Hinze, R.
A new approach to generic functional programming.
In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (2000), pp. 119–132.

[2] Hinze, R., and Löh, A.
"scrap your boilerplate" revolutions.
In *International Conference on Mathematics of Program Construction* (2006), Springer, pp. 180–208.

[3] Lämmel, R., and Jones, S. P.
Scrap your boilerplate: a practical design pattern for generic programming.
ACM SIGPLAN Notices 38, 3 (2003), 26–37.

Source code: github.com/irvin93d/scrap-your-boilerplate