HOCs + Context

Passing props the fun way

First: A crash course in some Typescript features

The plan

- Generic types
- Generic functions
- Generic type constraints
- Mapped types
- Conditional types
- ???
- Profit

Generic Type

- A variable to hold a type, rather than a value
- Example: Array<T>
- Can hold any type in place of T

```
type Person = {
  name: string;
  age: number;
};

const people: Array<Person> = [
  { name: 'Matt', age: 22 },
  { name: 'Chris', age: 27 },
  { name: 'Conaill', age: 42 }
];
```

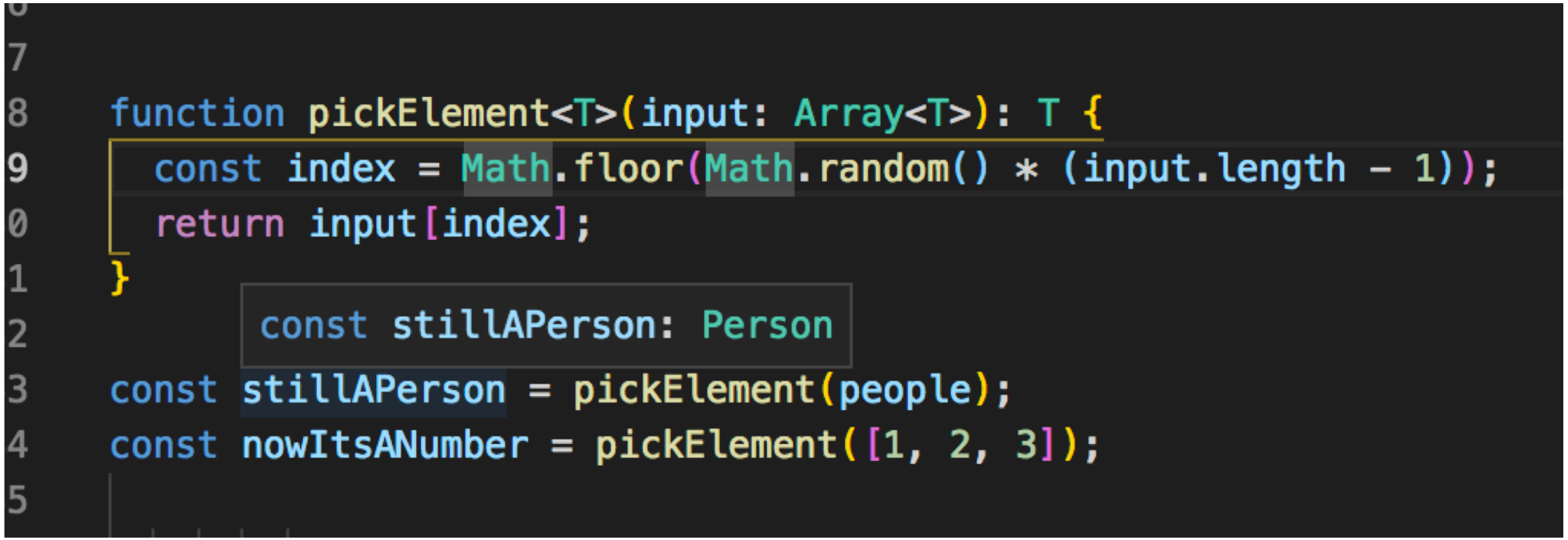
Nice

Generic Function

- Like a generic type, but can pass a type to a function instead
- Typescript can infer these types so we don't need to explicitly pass them

```
function pickPerson(persons: Array<Person>): Person {
  const index = Math.floor(Math.random() * (persons.length - 1));
  return persons[index];
}

function pickNumber(numbers: Array<number>): number {
  const index = Math.floor(Math.random() * (numbers.length - 1));
  return numbers[index];
}

const person = pickPerson(people);
const number = pickNumber([1, 2, 3]);
```

```
function pickElement<T>(input: Array<T>): T {
  const index = Math.floor(Math.random() * (input.length - 1));
  return input[index];
}

const stillAPerson = pickElement<Person>(people);
const nowItsANumber = pickElement<number>([1, 2, 3]);

        ////////////////////////
        //                    //
        //  Free real estate  //
        //   for more iTerm   //
        //      windows       //
        //                    //
        ////////////////////////
```



Typescript can infer the type arguments!

Type Constraints

- We can specify a type parameter extends another type
- T is **safely assignable** to Person

```
34
35  function pickPersonLike<T extends Person>(input: Array<T>): T {
36    const index = Math.floor(Math.random() * (input.length - 1));
37    return input[index];
38  }
39
40  const superheros = [
41    { name: 'Matt', age: 22, superpower: 'charm' },
42    { name: 'Chris', age: 27, superpower: 'looks' },
43    { name: 'Conaill', age: 42, superpower: 'intellect' }
44  ];
45
46  const randomSuperpower = pickPersonLike(superheros).superpower;
47
48
```

Type Constraints II

- We can add many type parameters
- And even form new types from these parameters

```
function giveItemToPerson<T extends Person, U>(input: Array<T>, item: U): T & { item: U } {
  const index = Math.floor(Math.random() * (input.length - 1));
  return {
    ...input[index],
    item,
  };
}

const superheroWithLaptop = giveItemToPerson(superheros, 'laptop');
console.log(`${superheroWithLaptop.name} has a ${superheroWithLaptop.item}`); // -> 'matt has a laptop'
```

```
const superheroWithSuperhero = giveItemToPerson(superheros, pickElement(superheros));
console.log(`${superheroWithSuperhero.name} has a ${superheroWithSuperhero.item.name}`); // -> 'chris has a conaill'
```

Mapped Types

- Iteration for types!

```
type Useless<T> = {
  // for every key T has, it's value is T[key]
  [key in keyof T]: T[key];
};

// so Useless<T> is just T...
```

Mapped Types II

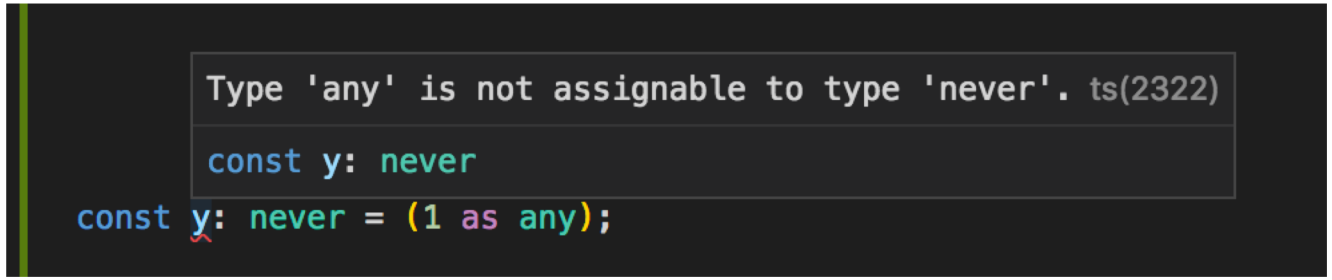
```
type Hero = Person & { superpower: string };

type PickProperties<TKeys extends keyof TObject, TObject> = {
  [key in TKeys]: TObject[key];
};

type NonIdentityRevealingProperties = 'superpower' | 'age';
type AnonymousSuperHero = PickProperties<NonIdentityRevealingProperties, Hero>;

// success
const spiderman: AnonymousSuperHero = { superpower: 'spider stuff', age: 30 };
```

Picking properties from is useful so Typescript includes as part of it’s built-in types: Pick<T, U>

Any and never

- Top types: **any** & **unknown**
- **Anything** can be assigned to them

- Bottom type: **never**
- **Nothing** can be assigned to never, not even any
- It cannot exist



Conditional Types

- We have iteration
- What about if statements for types?

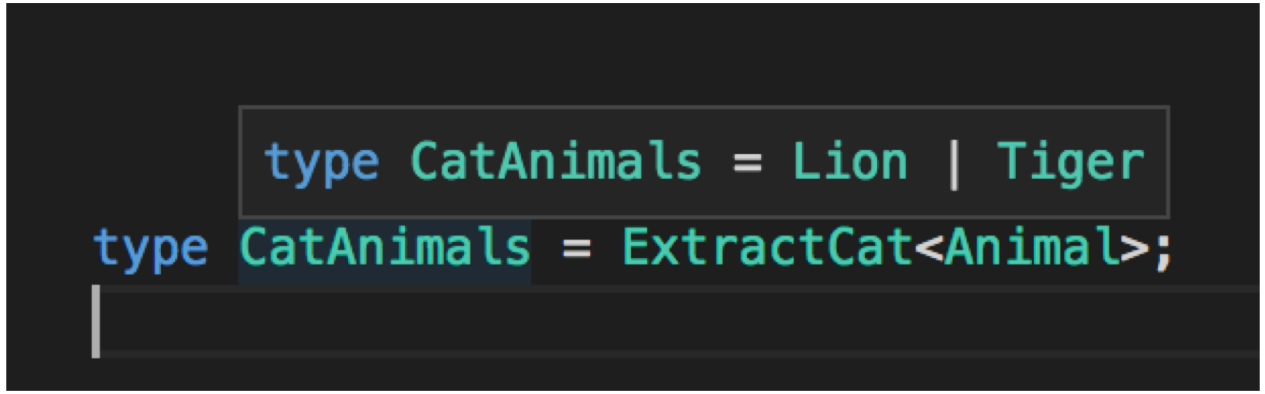
```
type Lion = { meow: () => void; eat: (food: string) => void; age: number; }
type Zebra = { bark: () => void; eat: (food: string) => void; age: number; }
type Tiger = { meow: () => void; eat: (food: string) => void; age: number; }
type Shark = { splash: () => void; eat: (food: string) => void; age: number; }

type Animal = Lion | Zebra | Tiger | Shark;
```

Here are some animals

Conditional Types II

```
type ExtractCat<T> = T extends { meow: () => void; } ? T : never;
```

- If T has a function meow: () => void, then give us T
- Otherwise it can never exist on ExtractCat<T>



?

- ?


Max. 2
Min. -1
Recycle me
WUT

Under the hood

1.

```
type CatAnimals1 =
  | ExtractCat<Lion>  // T extends { meow: () => void; } is true, so give us T (Lion)
  | ExtractCat<Zebra> // T extends { meow: () => void; } is false, so give us never
  | ExtractCat<Tiger> // T extends { meow: () => void; } is true, so give us T (Lion)
  | ExtractCat<Shark> // T extends { meow: () => void; } is false, so give us never
```

2.

```
type CatAnimals2 =
  | Lion
  | never
  | Tiger
  | never;
```

3. Discard never

```
type CatAnimals3 =
  | Lion
  | Tiger;
```

*This example was shamelessly stolen from http://artsy.github.io/blog/2018/11/21/conditional-types-in-typescript/

Conditional Types III

```
type ExcludeType<T, U> = T extends U ? never : T;
```

- Exclude: If we can assign T to U, ditch it

```
type NullablePrimitive = number | string | boolean | null | undefined;

// give us all the T's, unless they are assignable to null or undefined
type NotNullable<T> = ExcludeType<T, null | undefined>;

type NonNullableNumber = string | number | boolean
type NonNullableNumber = NotNullable<NullablePrimitive>;
```

Both Exclude<T, U> and NonNullable<T> are Typescript built-ins

All together now

- It would be cool if we could subtract two **objects**
- **Spoiler: we now can**
- We need the keys from T that are not assignable to U

```
type KeysOnlyInT<T, U> = Exclude<keyof T, keyof U>;
```

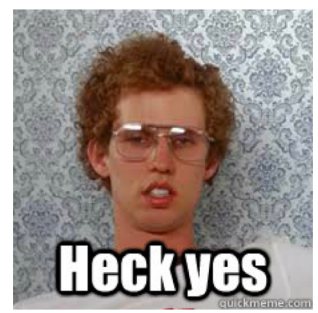
- Then Pick these property values back out of T

```
type Subtract<T, U> = {
  [key in KeysOnlyInT<T, U>]: T[key];
}
```

```ts
type Person = {
  name: string;
  age: number;
};

type Hero = {
  name: string;
  age: number;
  superpower: string;
};

// hero - person = { superpower: string }, right?
```

```ts
type Subtract<T, U> = Pick<T, Exclude<keyof T, keyof U>>;
```

```ts
type OnlyTheSuperPower = {
    superpower: string;
}
type OnlyTheSuperPower = Subtract<Hero, Person>;
```



Back under the hood

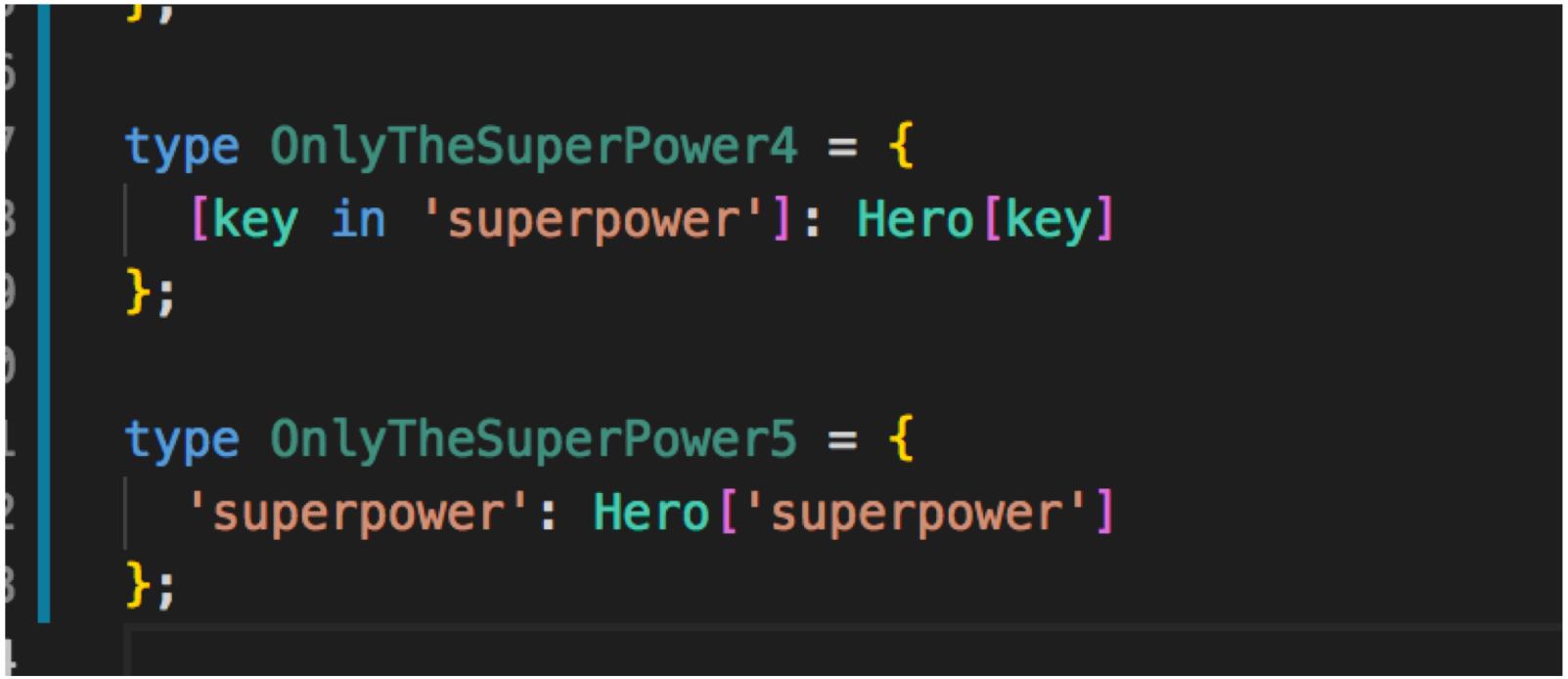
Let’s evaluate step by step…

```
type OnlyTheSuperPower = Subtract<Hero, Person>;

type OnlyTheSuperPower2 = {
  [key in Exclude<keyof Hero, keyof Person>]: Hero[key];
}
```

Back under the hood

```ts
type OnlyTheSuperPower3 = {
  [key in (
      ('superpower' extends 'age' | 'name' ? never : 'superpower') // -> 'superpower'
    | ('name' extends 'age' | 'name'  ?  never : 'name') // -> never
    | ('age' extends 'age' | 'name'  ?  never : 'name') // -> never
  )]: Hero[key]
};
```

Back under the hood



HOCs

A function which takes a component and gives you back a component

HOCs

- A way of reusing code in components without using inheritance
- Example: redux connect() or react-router's withRouter()
- A main type of HOC is an **injector –** dependency injection in react

Context

Pass props down the component tree without passing through every layer

Context 101

- Create a context with React.createContext(*defaultValue*)
- Add a content provider tag high up the component tree
- Add a content consumer to receive the value from anywhere

Context example

content.json

```
{
  "title": "This is my page",
  "content": [
    "Here is a paragraph",
    "And another paragraph",
    "etc etc etc"
  ],
  "footer": "footer footer"
}
```

```
import React from 'react';
import content from './content.json';

const MyContext: React.Context<{
    "title": string;
    "content": string[];
    "footer": string;
}>
const MyContext: React.Context<typeof content> = React.createContext(content);
```

Since v2.9, typescript can load the type of JSON file

Context example

```
const App = () => {
  return (
    <MyContext.Provider value={content}>
      <MyPage />
    </MyContext.Provider>
  )
};
```

Create context.provider

Child consumes with context.consumer

```
const MyPage = () => {
  return (
    <MyContext.Consumer>
      { content => (
        <>
          <h1>{content.title}</h1>
          {content.content.map(text => <p key={text}>{text}</p>)}
          <h5>{content.footer}</h5>
        </>
      )}
    </MyContext.Consumer>
  )
}
```

HOCs with Context

- Should our component be concerned with what context is?
- We thought it probably shouldn’t care
- We can lift the context into a HOC

Designing the HOC

- It needs to be a function which takes a component, and returns a component with the values from context as it's props
- The props of the resulting component will be the **component's props, without the props from context**
- Sounds like Subtract<ComponentProps, ContextProps>….
- ComponentProps will include ContextProps, so ComponentProps extends ContextProps will be a type constraint

Result

```tsx
type ContextValues = typeof content;
type SubtractComponent<T, U> = React.ComponentType<Subtract<T, U>>;

function withContent<ComponentProps extends ContextValues>(Target: React.ComponentType<ComponentProps>): SubtractComponent<ComponentProps, ContextValues> {
  return (props: Subtract<ComponentProps, ContextValues>) => (
    <MyContext.Consumer>
      {contextValues => <Target {...{...props, ...contextValues } as ComponentProps} />}
    </MyContext.Consumer>
  );
}
```

On casting as ComponentProps – this is an open TS issue with spreading generics:
https://github.com/Microsoft/TypeScript/issues/28748#issuecomment-450497274

```tsx
type WithContentProps = ContextValues & { extraProp: string };

const WithContentPage = (content: WithContentProps) => <>
  <h1>{content.title}</h1>
  {content.content.map(text => <p key={text}>{text}</p>)}
  <h5>{content.footer}</h5>
</>;

const WrappedWithContentPage = withContent(WithContentPage);
```

```tsx
const App = () => {
  return (
    <MyContext.Provider value={content}>
      <RegularContextPage />
      <WrappedWithContentPage extraProp="hi" />
    </MyContext.Provider>
  )
};
```

Can we do better?

Issues

- In CX we're creating many contexts for different values (e.g content, shell values, analytics info)
- We don't want to rewrite the HOC every time we want a different context
- Add another argument, and pass context as a variable

1.
```
type MyContextInjector<ComponentProps extends ContextValues> =
    (Target: React.ComponentType<ComponentProps>) => SubtractComponent<ComponentProps, ContextValues>;
```

2.
```
type ContextInjector<ComponentProps extends ContextType, ContextType> =
  (Context: React.Context<ContextType>) =>
    (Target: React.ComponentType<ComponentProps>) => SubtractComponent<ComponentProps, ContextType>;
```

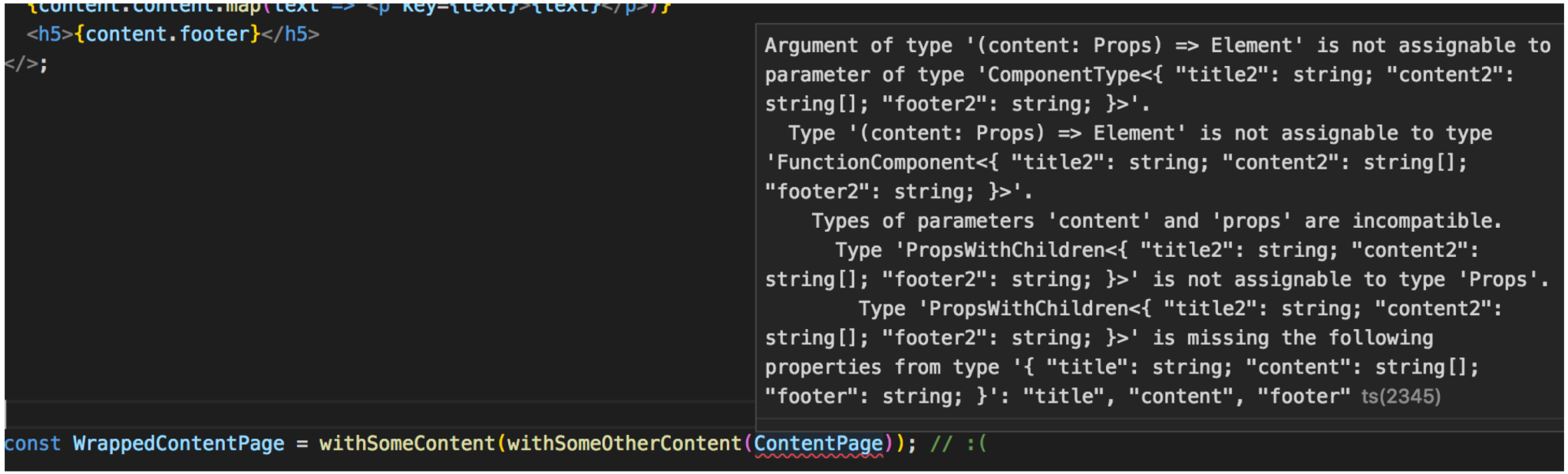
This should work

```
6
7   const injectContext = <ComponentProps extends ContextProps, ContextProps>(Context: React.Context<ContextProps>) =>
8     (Component: React.ComponentType<ComponentProps>): React.ComponentType<Pick<ComponentProps, keyof Subtract<ComponentProps, ContextProps>>> =>
9       props => (
0         <Context.Consumer>
1           {contextProps => <Component {...{ ...props, ...contextProps } as ComponentProps} />}
2         </Context.Consumer>
3       );
4
```

So spinning up an injector = 1 line

```
      );

  const withSomeContent = injectContext(MyContext);
  const withOtherContent = injectContext(MyContext2);
```

No Luck!



This used to be possible in typescript < 3.2. It's looking to be fixed. https://github.com/Microsoft/TypeScript/issues/28748#issuecomment-450497274

A compromise with any

```tsx
function consumeContexts<ExternalProps>([Context, ...remainingContexts]: React.Context<any>[]) {
  return (Child: React.ComponentType<any>): React.ComponentType<ExternalProps> => {
    const Wrapped = remainingContexts.length > 0
      ? consumeContexts<ExternalProps>(remainingContexts)(Child)
      : Child;

    return (props: ExternalProps) => (
      <Context.Consumer>
        {context => <Wrapped {...{...context, ...props}} />}
      </Context.Consumer>
    );
  }
}
```

And another HOC to inject context providers

```tsx
// HOC to inject context providers
type ProvideContext<T> = [React.Context<T>, T?];

function provideContexts<BaseProps>([[Context, value], ...remainingProviders]: ProvideContext<any>[]) {
  return (Child: React.ComponentType<BaseProps>): React.ComponentType<BaseProps> => {
    const Wrapped = remainingProviders
      ? provideContexts<BaseProps>(remainingProviders)(Child)
      : Child;

    // Use default value if not set
    const contextProps = value && { value };
    return (props: BaseProps) => (
      <Context.Provider {...contextProps}>
        <Wrapped {...props} />
      </Context.Provider>
    );
  }
}
```

Usage: Component tree stays simple

```tsx
type AppContainerProps = { header: string };

const AppContainer = (props: AppContainerProps) => {
  return (
    <div>
      <h1>{props.header}</h1>
      <WrappedContentPage extraProp="hello" />
    </div>
  )
};

const WrappedAppContainer = provideContexts<AppContainerProps>([
  [MyContext, content],
  [MyContext2, content2],
])(AppContainer);
```

Provide contexts

```tsx
type ExternalProps = { extraProp: string };
type InjectedProps = ContentValues & ContentValues2;

type Props = InjectedProps & ExternalProps;

const ContentPage = (content: Props) => <>
  <h1>{content.title}</h1>
  {content.content.map(text => <p key={text}>{text}</p>)}
  <h5>{content.footer}</h5>
</>;

const WrappedContentPage = consumeContexts<ExternalProps>([
  MyContext,
  MyContext2
])(ContentPage);
```

Consume contexts

Demo!

More Info

- All examples from this presentation: https://github.com/m-b-davis/typescript-hocs-context
- Typescript docs (Covers everything but a bit dry) - https://www.typescriptlang.org/docs/handbook/advanced-types.html
- Conditional types - https://artsy.github.io/blog/2018/11/21/conditional-types-in-typescript/