DEVELOP AND ANALYZE THE GRADUAL IMPLEMENTATION OF DATA TYPE VALIDATION AND INFERENCE USING PARAMETRIC POLYMORPHISM AND THE USE OF THE TYPESCRIPT PROGRAMMING LANGUAGE

Kniazev I.V.[1], Kopteva A.V.[2] (Russian Federation)

[1]Kniazev Ilia Vadimovich – Senior Software Developer,
JUNE HOMES, BELGOROD;
[2]Kopteva Anna Vitalievna – Senior Software Developer,
YANDEX, MOSCOW

Abstract: *the article analyzes and develops the gradual implementation of checking and inference of data types, applies parametric polymorphism as a development pattern, discusses the features of development in the TypeScript programming language and its application. Various bindings to the lexical environment and the context environment based on function parameters are also considered and developed. The result of the research is the developed functionality for checking the output for the presence of polymorphic types and auxiliary functions.*
Keywords: *polymorphism, functions, types, typescript, javascript, programming.*

РАЗРАБОТКА И АНАЛИЗ ПОСТЕПЕННОГО ВНЕДРЕНИЯ ПРОВЕРКИ И ВЫВОДА ТИПОВ ДАННЫХ С ПОМОЩЬЮ ПАРАМЕТРИЧЕСКОГО ПОЛИМОРФИЗМА И ИСПОЛЬЗОВАНИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ TYPESCRIPT

Князев И.В.[1], Коптева А.В.[2] (Российская Федерация)

[1]Князев Илья Вадимович – старший разработчик программного обеспечения,
June Homes, г. Белгород;
[2]Коптева Анна Витальевна – старший разработчик программного обеспечения,
Яндекс, г. Москва

Аннотация: *в статье анализируется и разрабатывается постепенное внедрение проверки и вывода типов данных, применяется параметрический полиморфизм как паттерн разработки, рассматриваются особенности разработки на языке программирования TypeScript и его применение. Также рассматриваются и разрабатываются различные привязки к лексическому окружению и контекстной среде на основе параметров функций. Результатом исследования является разработанный функционал для проверки вывода на наличие полиморфных типов и вспомогательные функции.*
Ключевые слова: *polymorphism, functions, types, typescript, javascript, programming.*

Введение

Есть некоторые функции, аргументы которых должны быть определенного типа. Например, функция «добавить» может складывать два значения типа Int. Нет смысла добавлять два значения Bool. Однако могут быть некоторые функции, которые не заботятся о фактическом типе своего аргумента. Их тело не может налагать никаких ограничений на тип их аргументов. Они известны как полиморфные функции. Простейшим примером полиморфной функции является функция идентификации, которая просто возвращает свой аргумент как есть:

(x) => x

Таким образом, эта функция должна работать для Int, Bool или любого другого типа. Его возвращаемый тип такой же, как и тип его аргумента. В нашей текущей системе такую функцию невозможно выразить, поскольку нам нужно присвоить функции конкретный тип.

Универсальная количественная оценка

Полиморфные типы можно выразить с помощью универсальной количественной оценки. Тип такой

(для всех A) A -> A

Это говорит о том, что для всех типов «A» тип этой функции - A -> A. Такие типы со всеми квантификаторами (также называемые универсальными квантификаторами, универсальными, поскольку они количественно определяют по всем типам) могут быть «инстанцированы» при использовании с конкретным видом. Создание экземпляра типа аналогично вызову функции. Точно так же, как когда вы вызываете функцию (x) => x на 1, мы заменяем экземпляры x в теле на 1 и удаляем параметр, создание экземпляра квантифицированного типа эквивалентно замене экземпляров «параметра типа» в его body с «аргументом типа» и удалением предложения forall. Например, если вы вызываете идентификатор с

аргументом Int, все экземпляры «A» должны быть заменены на «Int», давая тип Int -> Int. То же и для других типов. Количественные типы иногда называют схемами типов.

Существуют и другие виды количественной оценки, такие как ограниченная количественная оценка и экзистенциальная количественная оценка. Ограниченная количественная оценка полезна, когда нам нужно применить другие ограничения к переменным универсального типа. Это обычно наблюдается в типах интерфейсов.

Разработчик языка может выбрать, где разрешить квантификаторы в системе типов. Это может варьироваться от очень ограниченного, позволяющего квантификаторам быть только на верхнем уровне, до полностью неограниченного, позволяющего квантификаторам появляться везде, где может быть тип. Глубина, на которой может появиться квантификатор, обычно называется «рангом» типа. Например, в ранге 0 типы - это просто не полиморфные типы. Типы ранга 1 означают, что квантификатор может появляться только на верхнем уровне. Итак, при ограничении ранга 1 допустим следующий тип:

(для всех A) A -> A

Принимая во внимание, что это неверно:

(для всех A) A -> (для всех B. B -> B)

В этом конкретном случае мы можем просто переместить внутренний квантификатор на верхний уровень без изменения типа, поэтому он фактически считается классом Rank-1 как его эквивалент A -> B.

Классическая система типов Хиндли-Милнера допускает типы ранга 1. Это связано с тем, что в произвольной системе рангов вывод типа неразрешим, то есть невозможно вывести типы некоторых хорошо типизированных выражений без каких-либо аннотаций типов. Кроме того, сложнее реализовать вывод произвольного типа ранга.

На данный момент мы реализуем типы Rank-1, в которых квантификатор forall может появляться только в определенных местах.

Выражения let

Чтобы упростить реализацию, мы необходимо не присваивать аргументам функции полиморфные типы. Для этого решением будет добавление еще одного типа привязки, который позволит назначать переменным полиморфные типы. Это let привязки. Этот тип полиморфизма, ограниченный связыванием let, также известен как «let-полиморфизм».

Простое выражение let может быть описано так:

let x = 3 in plus (x) (1)

Здесь переменная x привязана к значению 3 в теле (plus (x) (1)). При вычислении это выражение вернет 4. В языках, ориентированных на инструкции, let утверждения не имеют тела - они просто привязывают переменную к значению выражения из следующего оператора к остальной части блока.

Итак, выражения let будут использоваться для присвоения полиморфных типов переменным. Например, функцию идентификации можно определить и использовать как:

let id = ((x) => x) in id (1)

Проверка типов привязок let

Давайте рассмотрим процесс проверки типов в этом фрагменте, чтобы неформально получить представление о процессе.

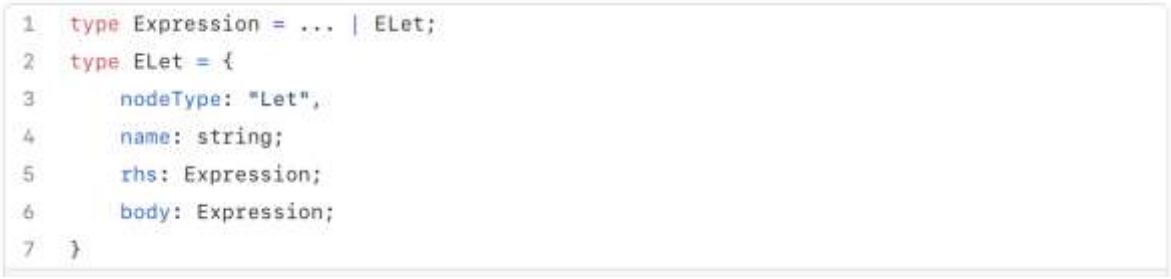
Начнем с объявления let, взяв правую часть присваивания ((x) => x) и вызвав по ней infer. Предполагаемый тип RHS будет (T0 -> T0), где T0 - это вновь созданная переменная типа.

Теперь видно, что (T0 -> T0) должно быть полиморфным. Таким образом, можно обобщить его до полиморфного типа (для всех T0. T0 -> T0) и проверить среду на наличие свободных вхождений переменных типа в тип, который обобщается. Если они не встречаются свободно в окружающей среде, то можно их оценить количественно. В этом случае среда была пустой, поэтому можно обобщить по T0.

Теперь, когда нашелся наиболее общий тип правой части let, можно добавить привязку к окружению (id: forall T0. T0 -> T0).

Теперь можно определить тип тела выражения let (id (1)). Начать стоит с попытки определить тип переменной «id» - ищем это в окружающей среде. Находим (для всех T0. T0 -> T0). Теперь функция infer должна возвращать обычный тип. У нас полиморфный тип. Итак, необходимо «создать экземпляр» полиморфного типа, удалив квантор. Сделать это можно взяв имя квантификатора, сгенерировав для него новую переменную типа и заменив его экземпляры в квантифицированном типе новой сгенерированной переменной типа. Итак, это даст (T1 -> T1). Теперь необходимо определить тип аргумента, который, как обычно, даст Int вместе с заменой (T1: Int). Итак, тип (id (1)) будет Int, а тип всего выражения let будет таким же.

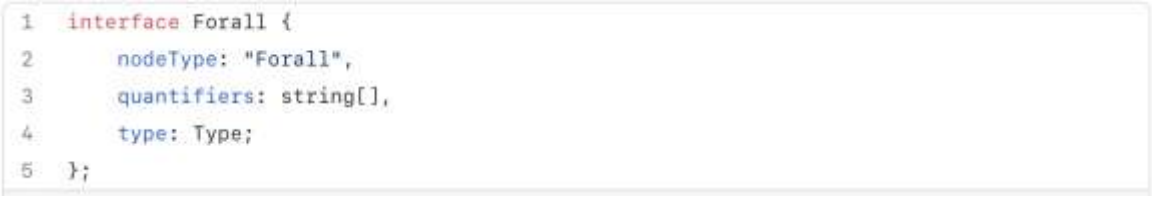
С помощью кода на TypeScript добавим новый тип для выражений let:



```
type Expression = ... | ELet;
type ELet = {
    nodeType: "Let",
    name: string;
    rhs: Expression;
    body: Expression;
}
```

Рис. 1. Описание типа для let выражения

Узел let имеет имя, выражение RHS, которое привязано к имени, и тело. Затем определим тип для квантифицированных типов Forall:

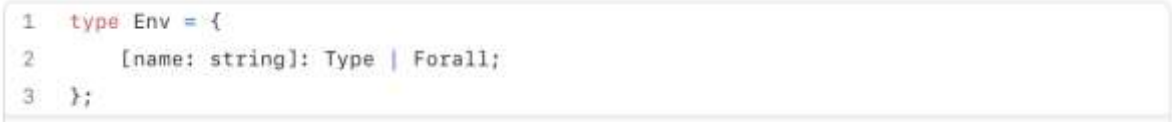


```
interface Forall {
    nodeType: "Forall",
    quantifiers: string[],
    type: Type;
};
```

Рис. 2. Описание интерфейса Forall

Он может количественно определять по нескольким типам переменных (квантификаторам). Обратите внимание, что мы не расширяем наше объединение Type, чтобы включить тип Forall.

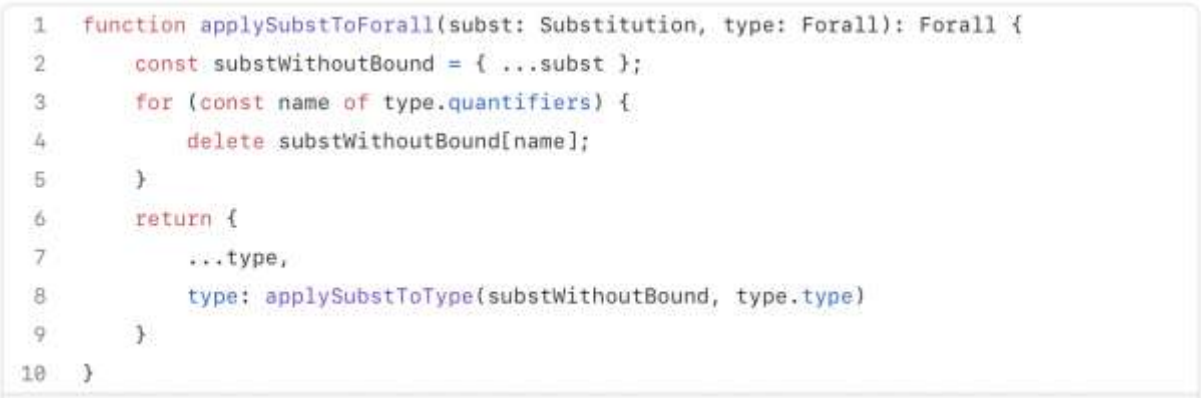
Окружение теперь может содержать типы, а также Forall. Поэтому необходимо обновить тип Env, чтобы отразить это:



```
type Env = {
    [name: string]: Type | Forall;
};
```

Рис. 3. Описание типа Env

Далее необходимо добавить функцию applySubstToForall для применения подстановки к типу Forall:



```
function applySubstToForall(subst: Substitution, type: Forall): Forall {
    const substWithoutBound = { ...subst };
    for (const name of type.quantifiers) {
        delete substWithoutBound[name];
    }
    return {
        ...type,
        type: applySubstToType(substWithoutBound, type.type)
    }
}
```

Рис. 4. Описание функции applySubstToForall

Из кода видно, что мы не заменяем типы, количественно определяемые методом forall, потому что привязки forall «затеняют» предыдущие привязки с тем же именем. Это похоже на то, как работают привязки функций. Переменная параметра функции скрывает переменные с тем же именем от внешних областей видимости. Таким образом, мы клонируем данную замену и удаляем из нее количественные имена, а затем применяем замену к типу forall. Стоит обратить внимание на использование здесь синтаксиса spread {… Subst} эквивалентно записи Object.assign ({}, subst).

Далее необходимо изменить подпись функции addToContext, чтобы принимать Forall вместе с Type.

function addToContext(ctx: Context, name: string, type: Type | Forall): Context

После этого необходимо добавить несколько функций для получения переменных свободного типа из разных типов. Стоит помнить, что свободные переменные - это переменные типа, которые не связаны квантификатором. Итак, для типов Forall свободные переменные - это свободные переменные тела без кванторов. Вот эти функции.

```
type FreeVars = {
  [name: string]: true;
}

function union(a: FreeVars, b: FreeVars): FreeVars {
  return { ...a, ...b };
}

function difference(a: FreeVars, b: FreeVars): FreeVars {
  const result = { ...a };
  for (const name in b) {
    if (result[name]) {
      delete result[name];
    }
  }
  return result;
}

function freeTypeVarsInType(t: Type): FreeVars {
  switch (t.nodeType) {
  case "Named": return {};
  case "Var": return {[t.name]: true};
  case "Function":
    return union(
      freeTypeVarsInType(t.from),
      freeTypeVarsInType(t.to)
    );
  }
}

function freeTypeVarsInEnv(env: Env): FreeVars {
  let result: FreeVars = {};
  for (const key in env) {
    const t = env[key];
    const freeVars = t.nodeType === "Forall"
      ? freeTypeVarsInForall(t)
      : freeTypeVarsInType(t);
    result = union(result, freeVars);
  }
  return result;
}

function freeTypeVarsInForall(t: Forall): FreeVars {
  const quantifiers: FreeVars = {};
  for (const name of t.quantifiers) {
    quantifiers[name] = true;
  }
  const freeInType = freeTypeVarsInType(t.type);
  return difference(freeInType, quantifiers);
}
```

Создание экземпляров

Нам нужна функция для создания экземпляров типов Forall. Она включает в себя создание переменных нового типа для каждой количественной переменной, их замену в теле forall и возвращение ее:

```
function instantiate(ctx: Context, forall: Forall): Type {
    const subst: Substitution = {};
    for (const name of forall.quantifiers) {
        const tVar = newTVar(ctx);
        subst[name] = tVar;
    }
    return applySubstToType(subst, forall.type);
}
```

Рис. 5. Описание функции instantiate

Вывод для выражений переменных var

Теперь, поскольку среда может содержать как все, так и обычные типы, необходимо обновить вывод для выражений переменных. Если имя переменной привязано к типу, то оно такое же, как и раньше, однако, если это Forall, то необходимо создать его экземпляр, а затем вернуть:

```
function inferVar(ctx: Context, e: EVar): [Type, Substitution] {
    const env = ctx.env;
    if (env[e.name]) {
        const envType = env[e.name];
        if (envType.nodeType === "Forall") {
            return [instantiate(ctx, envType), {}];
        } else {
            return [envType, {}]
        }
    } else {
        throw `Unbound var ${e.name}`;
    }
}
```

Рис. 6. Описание функции inferVar

Обобщение

Привязки Let могут связывать полиморфные значения, поэтому можно обобщить тип RHS привязки let на тип forall:

```
function generalize(env: Env, type: Type): Type | Forall {
    const envFreeVars = freeTypeVarsInEnv(env);
    const typeFreeVars = freeTypeVarsInType(type);
    const quantifiers = Object.keys(difference(typeFreeVars, envFreeVars));
    if (quantifiers.length > 0) {
        return {
            nodeType: "Forall",
            quantifiers: quantifiers,
            type: type
        };
    } else {
        return type;
    }
}
```

Рис. 7. Описание функции generalize

Он берет свободные переменные данного типа и помещает их в левую часть forall. Обратите внимание, что сначала проверяется, не является ли переменная типа свободной в среде. Мы не можем

обобщать свободные переменные в среде, потому что позже они могут быть привязаны к определенным типам.

Заключение

В данной статье был проанализирован и разработан полиморфизм форм (let-полиморфизм). Это делает систему типов более выразительной. Теперь мы можем написать вывод для выражений let следующим образом:

```
function inferLet(ctx: Context, expr: ELet): [Type, Substitution] {
    const [rhsType, s1] = infer(ctx, expr.rhs);
    const ctx1 = applySubstToCtx(s1, ctx);
    const rhsPolytype = generalize(ctx1.env, rhsType);
    const ctx2 = addToContext(ctx1, expr.name, rhsPolytype);
    const [bodyType, s2] = infer(ctx2, expr.body);
    const s3 = composeSubst(s1, s2);
    return [bodyType, s3]
}
```

Рис. 8. Описание функции inferLet

Таким образом, здесь выводится правая часть let, обобщается тип и добавляется обобщенный тип в контекстную среду. Затем возвращается предполагаемый тип тела и составные замены из предыдущих шагов.

Теперь есть возможность проверить вывод на наличие полиморфных типов и, подводя итоги, можно написать несколько вспомогательных функций:

```
function forall(quantifiers: string[], type: Type): Forall {
    return {
        nodeType: "Forall",
        quantifiers,
        type
    };
}

function eLet(
    name: string,
    _rhs: string | Expression,
    _body: string | Expression
): Expression {
    const rhs = e(_rhs),
          body = e(_body);
    return {
          nodeType: "Let",
          name, rhs, body
    }
}
function e(expr: Expression | string): Expression {
    if (typeof expr === "string") {
        return v(expr);
    } else {
        return expr;
    }
}
```

Рис. 9. Описание функций forall, eLet и e

Список литературы / References

1. Статья Функциональное программирование на TypeScript: паттерн «класс типов». [Электронный ресурс], 2021. Режим доступа: https://habr.com/ru/post/534998/ (дата обращения: 17.09.2021).
2. Документация TypeScript / Type Inference. [Электронный ресурс], 2021. Режим доступа: https://www.typescriptlang.org/docs/handbook/type-inference.html/ (дата обращения: 21.09.2021).
3. Документация TypeScript / Advanced Types. [Электронный ресурс], 2021. Режим доступа: https://www.typescriptlang.org/docs/handbook/advanced-types.html/ (дата обращения: 21.09.2021).
4. Статья Основные принципы ООП: инкапсуляция, наследование, полиморфизм. [Электронный ресурс], 2021. Режим доступа: https://gos-it.fandom.com/wiki/Основные_принципы_ООП:_инкапсуляция,_наследование,_полиморфизм/ (дата обращения: 16.09.2021).
5. Репозиторий TypeScript. [Электронный ресурс], 2021. Режим доступа: https://github.com/microsoft/TypeScript/#readme/ (дата обращения: 20.09.2021).
6. *Appel Rachel*. Статья “Write Object-Oriented TypeScript: Polymorphism”. [Электронный ресурс], 2021. Режим доступа: https://blog.jetbrains.com/webstorm/2019/03/write-object-oriented-typescript-polymorphism/ (дата обращения: 17.09.2021).
7. *Черный Борис*. Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений [Текст], 2021. 212 с.
8. *Макконнелл Стив*. Совершенный код [Текст], 2017. 111-113 с.