Jsonnet Type Inference

Summary

Mainly, this document contains design ideas of how to perform type inference for certain Jsonnet constructions. It also includes basic info about Jsonnet, type theory, and the type inference algorithm, which are necessary to understand the content of this document.

Author: Iryna Dobrovolska
Contributors: Pippijn van Steenhoven
Reviewers:
Pippijn van Steenhoven (pippijn@google.com),
Dan Sheridan (dsheridan@google.com),
Dave Cunningham (dcunnin@google.com),
Stanisław Barzowski (sbarzowski@google.com)

Status:[1] Draft | WIP | **In Review** | Final
Approved: YYYY-MM-DD, announced: https://groups.google.com/g/jsonnet
Implementation: https://github.com/Dobryk15/jsonnet/tree/jsonnettype/core/type_inference

Created: 2020-08-11
Last updated: 2020-10-30
Self Link: bit.ly/2JbMr0N

Username	Role	Status	Last Change
dcunnin	Reviewer	PENDING	2020-10-05
dsheridan	Reviewer	PENDING	2020-10-05
pippijn	Reviewer	PENDING	2020-10-05

Objective

The goal of this project is to create a framework for static type inference of structurally typed data languages, and apply it to Jsonnet to create a statically typed dialect with better error messages for certain classes of programming errors.

It is a **non-goal** to support the full flexibility of Jsonnet's dynamic types. For example, we will not support arbitrary custom overloading of functions using `isNumber`/`isString`/etc. Where complexity of the type system would explode if we were to support a Jsonnet feature, we will choose to restrict the feature instead of complicating the type system, because a more complex type system has more complicated type error messages, reducing the usefulness of this project.

[1] **Draft** = Don't use or rely on any of this material. **WIP** = Somewhat reliable content, but expect changes/deletions/additions. **In Review** = Reviewers are reviewing this document, authors are making suggested changes. **Final** = The content of this document will not change anymore and has been approved by the team.

Background

1. Hindley-Milner Type Inference Implementation in Python as a reference approach: https://github.com/rob-smallshire/hindley-milner-python
2. Type rules standard notation https://en.wikipedia.org/wiki/Type_rule
3. Type rules for HM system http://dev.stephendiehl.com/fun/006_hindley_milner.html#constraint-generation
4. Jsonnet specification https://jsonnet.org/ref/spec.html
5. Dependent types
   1.https://medium.com/background-thread/the-future-of-programming-is-dependent-types-programming-word-of-the-day-fcd5f2634878 (types as values; idea with induction)
   2. https://en.wikipedia.org/wiki/Dependent_type
6. Improving Jsonnet ecosystem (In particular, chapter 3 “Linter”, p. 21.) https://barzowski.com/msc_thesis.pdf
7. Jsonnet to Java https://github.com/google/jsonnet/tree/master/java_comparison

Detailed design

Notation

We use Jsonnet syntax here to express objects, functions, different expressions. In type theory, there is common notation `x : number`, which means that `x` has type `number`. However, in Jsonnet `:` is used for the field definition. Thus, we will use `∈` instead and interpret a type as a set of values, so this ‘belonging’ symbol makes sense.

Jsonnet syntax

Object

Object is the main component of the Jsonnet program and basically the whole program is an object.

Any object consists of key-value pairs. Values can be not only scalars but also other (nested) objects, arrays. One object can inherit another one by overriding its fields and/or adding new fields. Some fields can be hidden, which is identified by `::` symbol. However, it has no influence on the type inference, it is just about visibility of the field in the final JSON object. Also, some local variables can be defined inside an object. Below is an example:

```
local T = {
        basic:: 1,
        nested_obj: {str: "abc", num: 1 }
};
{
        B: T {
                local awards = 3,
                own_quantity: super.basic + awards,
                basic_quantity: super.basic,
        },
        C: T {
                basic: 5,
                own_quantity: self.basic,
                basic_quantity: super.basic,
        }
}
```

Reference tools

1. **Self** - reference to the object itself (like in python).
   Example:

   ```
   {
       x: 1,
       y: self.x
   }
   ```

2. **Super** - reference to the inherited (base) object .
   Example:

   ```
   {
       x: {
           x1: 1
       },
       y: self.x {
           y1: super.x1
       }
   }
   ```

3. **$** - reference to most outer object
   Example:

   ```
   {
       x: 1,
       y: {
           x: 2,
           z: {
               t: $.x
           }
       }
   }
   ```
   In this example, t will have value 1 because $.x refers to the most outer x.

Core syntax of Jsonnet

```
e ∈ Core  ::=  null | true | false | self | super | string | number
               | { { assert e } { [ e ] h e } }
               | { [ e ] : e for id in e }
               | [ { e } ]
               | e [ e ]
               | e ( { e } { id = e } )
               | id
               | local id = e { id = e } ; e
               | if e then e else e
               | e binaryop e
               | unaryop e
               | function ( { id = e } ) e
               | error e
```

Currently, only some syntactic structures are supported. Actually, the goal was not to support everything but only a useful subset of syntax.

Types and kinds

Let's clarify the difference between types and kinds. According to Wikipedia, `type` is "an attribute of data" while `kind` is "a type of types". Let's see what it means. Actually, we can generalize/group some values by a certain type. For example, `1`, `1.5`, `1e3` belong to the type `number` or we can say that type `number` is inhabited by those values.

In the case of kinds, their inhabitants are not values but other types. The simplest kind is denoted `*` and can be inhabited by any monotype, e.g. `Number`, `Boolean`, or by composed type like `List Number`. Let's call this kind `*` a **primitive type**.
Below there are some examples, where names of kinds are capitalized and names of types begin from the lower-case letter.

Type constructors:

1. **Primitive** types are of kind `*` and can be inhabited by values, e.g. `Number ∈ *`.
2. **Parameterized** types (= TypeOperator) take monotypes as parameters, e.g.:
   a. `Function ∈ * -> (* -> *)`,
      i. `Function Number ∈ * -> *`,
      ii. `Function Number String ∈ *`,
   b. `List ∈ * -> *`,
      i. `List Number ∈ *`, (so, List Number is of kind *, thus a primitive type)
      ii. `List (Function Number String) ∈ *`,
   c. `Identity ∈ (* -> *) -> (* -> *)`,
      i. `Identity List ∈ (* -> *)`,
      ii. `Identity List Number ∈ *`,
   d. `WithNumber ∈ (* -> *) -> *`,
      i. `WithNumber (Op ∈ * -> *) = Op Number` (type function)
      ii. `WithNumber List ∈ *`,
      iii. `f ∈ WithNumber List` ⇒ `f ∈ List Number`,
      iv. `[3] ∈ WithNumber List`,

e. `singleton ∈ (m ∈ * -> *) -> a -> m a`
f. `(singleton List) 3 ∈ List Number`
g. `(singleton (Function String)) 3 ∈ Function String Number`
h. `HigherOrder ∈ (* -> *) -> (* -> *)`
i. `[3] ∈ List Number`

We will support at the moment primitive types and maybe later we will use parameterized types for errors.

Let's continue with types. We can classify types as **type variables** and **type operators**. Type variables stand for an arbitrary type. Type operators can be simple **monotypes** (often called nullary operators because they don't take any parameters) or more complex types which take monotypes as parameters. When types take type variables as parameters, they are called **polymorphic types**.

Types:

1. **Type variable (universal type)**
   We use small lower-case Latin letters to denote type variables. For example: `a`, `b`, `c`, …
   In the current implementation, null has a universal type.
2. **Number** (includes all numeric types): `5`, `6.66`
3. **String**: `‘abc’`, `“abc”` , `|||abc|||` (for multiple lines)
4. **Boolean**: `true`, `false`
5. **Array** in Jsonnet can contain elements of different types. However, we only consider array with elements of the same type:
   - `[1,2,3,4,5], [“abc”, “google”], [{x: 1', y: ‘abs’}, {x: 2}]` - valid
   - `[‘one’, 1, true, 2]` - invalid
6. **Record (aka row[2])**
   Record is a dictionary where `key` is field name, `value` is the type of the corresponding field.
   Initially, record is a polymorphic type because we use type variables to construct it.
   Later on, during type inference, we infer those types if it is possible. But sometimes a record can remain fully or partially polymorphic if there are no restrictions on all or some fields respectively.
   Let's consider the following Jsonnet object:
   `{x: 1, y: {y1: ‘smth’, y2: $.x + 2}}`
   The type of this object is the following record:
   `{x ∈ number, y ∈ {y1 ∈ string, y2 ∈ number}}`
7. **Function** type notation: `f ∈ (a) -> b`
   Example of function definition in Jsonnet: `local f(x) = {y: x.t}`

Remark:
To understand how type inference deals with records, apart from `record` we should consider `record_ctor`. The difference between them is that `record` is an instance of type row operator and is basically what we described here. `record_ctor` is actually a function with recursively built return type which contains the corresponding `record`. For example, `record` = **{x ∈ number, y ∈ a}** and its corresponding `record_ctor` **∈ (number) -> (a) -> {x ∈ number, y ∈ a}**. In other words,

```
record_ctor ∈ Function(x_type, Function(y_type, {x ∈ x_type, y ∈ y_type}) )
```

[2] In this document, **row** and **record** have the same meaning and can be used interchangeably.

Initial type environment (Type schemes)

Binary operations
Here: a - universal type variable (any type), b - number | array | object, b - number | string,

- `(+)       : b -> b -> b | a -> string -> string | string -> a -> string`
- `(-)       : number -> number -> number`
- `(*)       : number -> number -> number`
- `(/)       : number -> number -> number`
- `(|)       : number -> number -> number`
- `(&)       : number -> number -> number`
- `(^)       : number -> number -> number         (xor operator)`
- `(%)       : number -> number -> number         (modulo operator)`
- `(!)       : boolean -> boolean`
- `(<)       : c -> c -> boolean`
- `(<=)      : c -> c -> boolean`
- `(>)       : c -> c -> boolean`
- `(>=)      : c -> c -> boolean`
- `(&&)      : boolean -> boolean -> boolean`
- `(||)      : boolean -> boolean -> boolean`
- `(==)      : a -> a -> boolean`
- `(!=)      : a -> a -> boolean`

Unary operators:

- (+) `: number -> number`
- (-) `: number -> number`
- (!) `: boolean -> boolean`
- (~) `: number -> number`

Remark 1

Bitwise operators (|), (&), (^) work with float number `5.5` like with `5`, so they accept any possible number but work with its integer part.

Remark 2

(+) binary operation is implemented in Jsonnet in such a way that if one operand has type string, another one will be implicitly casted to string. It causes some ambiguity and difficulties for type inference. Different approaches can be used to tackle this problem. But we will consider one described below.

The idea is to preprocess (+) expressions when one of its arguments is a string literal. So, `‘Age’ + 1` becomes `‘Age’ + to_string(1)`. Thus, the type of second argument (`1` in this case) doesn’t matter anymore since we erase it by applying `to_string()`. So, the type of (+) expression where at least one of the arguments is string, will be **string -> string -> string**.

So, we can assume the following:

- (+) :: `a` -> `a` -> `a` where `a` = `string` | `number` | `object` | `array`

- to_string :: `a` -> `string`

Examples:

- If string operands are literal_strings and type `string` is explicit, then we can apply to_string().
  `{ x: "Version_" + 1 + "_beta" }` => `{ x: "Version_" + to_string(1) + "_beta" }`
- If operands from the previous example are local variables `x` and `y`, we don't know if they are strings. So, by requiring that (+) operands should have the same type, we will get a type error in this case.

```
local x = "Version_";
local y = "_beta";
{ z: x + 1 + y }
```

=> `{ z: x + 1 + y }`

We can solve this problem by inferring types of local variables during translation of Jsonnet AST to HM AST. But it may be not easy to do. However, this situation opens a question that maybe we would like to perform type inference for Jsonnet AST.

Remark 3

(==) is deep value equality. Thus, the result of this operation `{x: {y: 1}} == {x: {y: 2}}` is `false`.

- (==) for jsonnet:
  - a -> b -> boolean
  - `local t1 = { x: self.y == 3, z: self.y };` // `{ x ∈ boolean, y ∈ a, z ∈ a }`

Remark 4

`%` is desugared to `std.mod()` and does Python-style string formatting if the left-hand side (lhs) of expression is a string.

Example: `“Age: %d” % 1`.

As can be seen from the example, the type of right-hand side (rhs) argument depends on the value (here, on specificator `d`) of the lhs argument (formatted string). String formatting operator has not been implemented yet but there are a couple of approaches which can be used:

- use dependent types;
  then, the type of expression is “string -> a(lhs) -> string”, which means that type of rhs argument depends on the lhs argument.
- preprocess the whole expression before analysing;
  then, the example above will be converted to smth like: `“Age: ” + format_d(“”, 1)`, where `format_d()` corresponds to the specificator `d`
- let rhs argument have universal type at that point and defer it to dynamic type check.

Interesting case to think about is when rhs argument is an object. Example:

```
{
    age: 5,
    info: "She is %(age)s" % self
}
```

Language semantics

null

null can be expressed using LetrecAnd:
null = LetrecAnd({"null": Identifier("null")}, Identifier("null"))

But we just map ‘null’ to the new type variable in the type environment, which we use during type inference.

error

- { x: !true }
- (!) :: Bool -> Bool
- local t1 = { x: error "hello", y: self.x } // { x ∈ Error a, y ∈ a }
  t1 { x: "hello" } // OK: Error a => string
  t1 { x: 3 } // OK: Error a => number
  t1 { x: 3, y: "hello" } // FAIL
- (error) :: string -> Error a
- error "a" + error "b" // a + a -> a

Row polymorphism

In Jsonnet we deal with objects which can be parameters of functions. Since we want the same function to deal with objects of different but compatible types we need row polymorphism. Then, if there are no restrictions on the argument of function func(obj), it can take any obj. For example, if `func(obj) = obj {}`, then it can operate on any obj: empty object `{}`, object with different number of fields: `{x: 1, y: 2}` or `{x: 1, y: 2, z: 1}`, or object with different types: `{x: 1}` or `{x: ‘1’}`.

Row polymorphism notation and remarks:

- ‘x?’ - means that the field ‘x’ is optional. It can be used to add restrictions on the fields of the base object during inheritance. Let’s assume that `{x: 1}` inherits from object `base`. But we don’t know the type of `base` at that point. After inheritance, the type of `base` may be expressed as `{x? ∈ number}`, which means that the field `x` is not required to be in the `base` but when this field is present its type should be `number`.
- ‘x!’ - means that field ‘x’ is required. Namely, an object with a field marked ‘!’ can inherit from the object that contains this field. For more information, see Required fields.
- ‘x<’ - means that the field ‘x’ is required on the left. In other words, an object whose field is marked with ‘<’ can inherit only from objects which contain the field ‘x’. For example, `{y: super.x}` can inherit from `{x: 1}` but not `{y: 2}` because the `x` should be present in the inherited by `{y: super.x}` object.
- ‘...’ represents other fields that may be present in the record; inside one type we can have different rest ‘...’, so we can name it like ‘r1’, ‘r2’, ‘r3’.
- When we refer to some fields of record in the body of function, they must be in this record; e.g. `local f(obj) = {a: obj.b}` → we assume that `b` exists in `obj`.
- We don’t allow to change the type of the field, so if we assigned 1 to the field x, we assume that x had type `number` in the record if it existed before the new assignment.

- Let's consider types of two different objects: o1 ∈ {a ∈ number, ...} and o2 ∈ {a ∈ number, b ∈ string}. Then, o1 can be considered as a supertype w.r.t. o2 since o2 ⊆ o1.

Some examples:

- Field extension or update:
  `local f(obj) = obj {m: obj.n.o}`
  type: `f ∈ ({n ∈ {o ∈ a, r1...}, m? ∈ a, r2...}) ->`
  `{m ∈ a, n ∈ {o ∈ a, r1...}, r2...}`
- Field selection:
  `local f(obj) = obj.x`
  type: `f ∈ ({x ∈ a, ...}) -> a`
- local `f(obj) = obj {x: 5}`
  type: `f ∈ ({x? ∈ number, ...}) -> {x ∈ number, ...}`
- `local f(obj, var) = obj {x: var}`
  type: `f ∈ ({x? ∈ a, ...}, a) -> {x ∈ a, ...}`
- `{y: super.x}`
  type: `{x< ∈ a, y ∈ a}`

Algorithm

Description

For type inference, we use Hindley-Milner(HM) algorithm. Originally, it was applied to HM language, based on Lambda Calculus. For compatibility with Jsonnet, we introduced some modifications of HM language and corresponding changes to the algorithm itself. Particularly, we extended the language by adding new nodes and types (see Extended HM language). The type system is described in Types and kinds section.

Type inference is a recursive process, based on tree traversal. When we are going down AST we put some constraints on initially free type variables, assigned to identifiers. Then, when going up, we use already collected information and restrictions to infer other types and to unify types which should be the same. To see how it works, look at Example.

To see the whole program flow from Jsonnet program to the type inferring, take a look at this diagram "Type inference stages" .

Components of HM algorithm:

- Type environment
  Mapping between identifiers and their corresponding types.
- Non_generic
  List of type variables that don't occur in other type expressions.
- Unification
  The function unify accepts a constraints set as input and produces a substitution (or fails causing an inference error).
- Instantiation
  In the beginning, we operate with universal type variables. They are assigned to some identifiers or are parts of more complex polymorphic types. Later, during unification of a type variable with another type this type is assigned to the instance of that type variable.

- Pruning
  The goal is to reduce the type variable to its instance if this type variable was already instantiated.

Introduced changes to HM algorithm:

- Unification was extended to deal with a new type - TypeRowOperator. Here, the concept of row polymorphism is used when two row types are unified. So, we only require that the fields that occur in both row1 and row2 have the same type. However, row1 and row2 don't need to have the same number of fields.

Unification. Row unification

Unification together with instantiation of type variables is a very powerful mechanism to store connections/dependencies between type variables. Let's consider the following example:

```
{
  x: self.y,
  y: self.z,
  z: {z1: 1}
}
```

In the beginning, we set in the type environment that types of fields are just type variables `x ∈ a`, `y ∈ b`, `z ∈ c`.

Let's say that each type variable has an attribute called 'instance'. When the instance of the type variable is defined, then we say that type variable is instantiated. Also, let's say we have a prune() function which returns an instance of type variable or type variable when its instance is not defined.

Now, let's consider some type inference steps for the example above:

- Analysis of the type of `x`:
  Type of `x`'s body is `b` (type of `y`). Then, we unify type variable `a` with type variable `b`, and as a result we instantiate a: `a.instance = b`
- Analysis of the type of `y`:
  Type of `y`'s body is `c` (type of `z`). Then, we unify type variable `b` with type variable `c`, and as a result we instantiate b: `b.instance = c`. Now, when we prune `a`, we won't get `b` but `c` since pruning is recursive. So, both `a` and `b` are instantiated with and refer to type variable `c`.
- Analysis of the type of `z`:
  Type of `z`'s body is `{z1 ∈ number}`. Then, we unify type variable `c` with record `{z1 ∈ number}`, so `c.instance = {z1 ∈ number}`. And following the logic above `a.instance = b.instance = {z1 ∈ number}`.
- Thus, knowing the types of object's fields, we know the type of object:
  `{x ∈ {z1 ∈ number}, y ∈ {z1 ∈ number}, z ∈ {z1 ∈ number}}`

With this example, I wanted to show that it is important to keep those connections between type variables and use unification to substitute a type variable with an inferred type.

Row unification happens in different scenarios. For example:

- When type of the field was partially inferred before its definition
  Example:

  ```
  {
    x: self.y {x1: 1},
    y: {x2: 3}
  }
  ```

  Before y was defined we set requirements on its type during inheritance. So, we unify taken from the environment type of y (with optional fields) and type of its definition (`{x2 ∈ number}`).

- When types of base and child objects are known and we are trying to unify them. If they are unifiable, we can fully infer or set some constraints on the type of inheritance result as well as on the types of base and child objects. These 2 stages may be considered as a **mixin unification**.

Suggestion: maybe it is better only to check if rows are unifiable inside unify() function and create separate functions for cases where row unification needs to perform some changes in row1 and/or row2. So, we can check if rows can be unified within one function but type updates happen somewhere else and are different depending on the AST node.

Below there are some examples which cover different cases of unification (not only unification between two rows but also row and type variable). I don't consider functions here.

Case 1. Unification of partially inferred rows with optional fields.

```
1. {
2.   t1: self.y {k1: 1},
3.   t2: self.z {k2: true},
4.   z: self.y,           // types of z and y are partially inferred records with optional
5.                        // fields
6.   y: {k3: 1}
7. }
```

How to unify z and y when their types are instantiated type variables? - We can modify the type of y by info about z and then set an instance corresponding to type variable z equal to y's type. Then, by inferring the type of y, we take its type from the type environment and record `{k3 ∈ number}` and do the following:

- check if they can be unified by comparing types of fields from both sides,
- update `{k3 ∈ number}` and instantiate y's type variable with obtained type.

The problem is that because of the current implementation of analyse() function, the type of `self.y` (line 4) will be not a type variable but just its instance. So, we cannot bind the type variable of z to the type variable of y. As a result, we cannot bind the type of z to the type of y which will be inferred later.
One potential solution is to store somewhere else (e.g., in another type env) the partial types until they will be completely inferred and instantiate type variables, which correspond to the fields, only with type of field's definition. So, when some field is used before definition, we don't instantiate its type variable from the initial type environment but store received type restrictions in another dictionary for example.
Another option is to create an additional attribute in the class TypeVariable which corresponds to 'type restrictions'. So, the type variable can get some restrictions and before instantiation we should check if a potential instance satisfies those restrictions.

Case 2. Unification of type variable and partially inferred type with optional fields

```
{
  y: self.z {x: 1},
  z: self.t,           // type of t is type variable, type of z is record with optional field
  t: {k: true}
}
```

Potential solution for row unification

Let’s consider the same example as in the case 1:

```
1. {
2.   t1: self.y {k1: 1},
3.   t2: self.z {k2: true},
4.   z: self.y,           // types of z and y are partially inferred records with optional
5.                        // fields
6.   y: {k3: 1}
7. }
```

Algorithm (by line):

1. n/a
2. y : {k1? : number}@0x1
   t1 : {k1 : number, ->0x1}@0x2
3. y : {k1? : number}@0x1
   t1 : {k1 : number, ->0x1}@0x2
   z : {k2? : bool}@0x3
   t2 : {k2 : bool, ->0x3}@0x4
4. z = y
   unify {k1? : number}@0x1 and {k2? : bool}@0x3
   ⇒
   y : {k1? : number, k2? : bool}@0x1
   z : {k1? : number, k2? : bool}@0x1
   t1 : {k1 : number, ->0x1}@0x2
   t2 : {k1 : number, ->0x3}@0x4
   {==0x1}@0x3
5. n/a
6. unify {k1? : number, k2? : bool}@0x1 with {k3 : number}
   ⇒
   y : {k1? : number, k2? : bool, k3 : number}@0x1
   z : {k1? : number, k2? : bool, k3 : number}@0x1
   t1 : {k1 : number, ->0x1}@0x2 == {k1 : number, k2? : bool, k3 : number}
   t2 : {k2 : bool, ->0x3}@0x4 == {k2 : bool, k1? : number, k3 : number}
7. n/a

Inheritance. Challenges and ideas

Inheritance in Jsonnet is just a special case of plus operation between two objects, when an RHS object has the form `{...}`.

Inheritance rules in Jsonnet

You can find a description of inheritance rules in Jsonnet here. Below, I just want to point out some cases.

Example 1:

```
{
  x: {
    t: 2,
  },
  y: self.x {
    k: self.t
  }
}
```

Valid example

Example 2:

```
{
  x: {
    k: self.t,
  },
  y: self.x {
    t: 2
  }
}
```

This example causes manifestation error because it is expected that `t` will be defined inside `x`'s body or inside the base object. But `x` has no base object and no field `t`.

Example 3:

```
{
  local x = {
    k: self.t,
  },
  y: x {
    t: 2
  }
}
```

However, when the base object is a local variable we get a valid Jsonnet program.

Assumptions

- We assume that fields with the same names in inherited and inheriting objects have the same type. So, basically the type value (if we can talk about such one) should be the same for self.<field_name> and super.<field_name>. The motivation is that although we may need to have different values for the field in base and child class, having different types seems to be strange and confusing. So, the example below will cause an error:

```
{
    local x = {k: 'str'},
    y: x {m: super.k, k: 2}
}
```

  However, if the field in the base object is defined as null, we can override it in different child classes with values of different types. So, the following program is valid due to type inference:

```
{
    local base = {
        z: null
    },
    x: base {
        z: 3
    },
    y: base {
        z: "str"
    }
}
```

- Inheriting objects don't influence the base object unless base is a function parameter.
  Thus, the type of the following object:

```
{
    local x = self.y {
        z: 3
    },
    y: {
        z: null
    }
}
```

  is {y ∈ { z ∈ a } }

- We have 2 different type variables in the different instantiations.
  Thus, the type of the following object:

```
{
    local base = {
        m: {z: null}
    },
    x: base {},
    y: base {}
}
```

is { x ∈ { m ∈ {z ∈ a }}, y ∈ {m ∈ { z ∈ b }}}

Idea of inheritance representation

Idea: represent inheritance as function application, where:

- base object - function (i.e. function's body),
- child object - function parameter.

So, literally, we apply a base object to a child object.

Example:

```
local base = {res: self.x+1};
{y: base {x: 3}}
```

Equivalent representation in terms of function application:

```
local f(arg) = {res: arg.x+1}
y: f({x: 3})
```

We can also do the other way around: transform any function application to the inheritance form.

For example:
local f(x) = x + 1; f(3) --> local f = { ret: self.x + 1 }; (f + { x: 3 }).ret.
In the part (f + { x: 3 }).ret, plus sign represents inheritance.
The body of the function is wrapped up into an object:
x + 1 -->{ ret: self.x + 1 },
as well as the passed value:
3 --> { x: 3 }

As a result, it will be easier to deal only with function application or only with inheritance.

Visibility challenge

The fields of base class have to be visible to the child class without overriding them in the child class. The opposite should also hold: fields of child class have to be visible to the base class. Thus, in the next two examples field 'k' should be recognized:

```
{
    local base = {
        z: self.k
    },
    x: base {k: 1}
}
```

```
{
    local base = {
        k: 1
    },
    x: base {z: self.k}
}
```

The possible way to handle this situation is to preprocess AST before inferring types in such a way that we extend objects with fields that are used inside the object but not defined. Then, for the first example, we will add `k: null` to the base object:

```
{
    local base = {
        z: self.k,
        k: null
    },
    x: base {k: 1}
}
```

Required fields

Idea: to set requirements on fields that need to be presented in a child or base object, we introduce a concept of the `!` flag for the field. All such fields have to be overridden before materialisation. So, after type inference, we check if there are types that contain `!`. If yes, then we obtain an error.

Object comprehension

Currently, object comprehension isn't supported.

Let's consider the following example:

```
local f(x) = { ["f" + i]: true for i in std.range(0, x) };
f(10)
```

Here type of `f` depends on the value of `x`:
f ∈ `x -> record_type_that_depends_on_x`

When an object is created during object comprehension, we need to evaluate the expression inside `{}` to be able to reason about the type of created object (for instance, to know the field names of the object). Thus, the type of object created in such a way depends on some expressions that need to be calculated.

Optional fields in inheritance

Let’s consider 2 different scenarios during inheritance:

1. Type of the base object is already inferred
   In this case, we only check the common fields of base and child objects. And don’t add any optional fields to the base object. So, `?` flag won’t be used here.
2. Type of the base class is unknown (so, it is just type variable)
   In this situation, we instantiate the type variable of base class with new record_type (or maybe record), which consist of fields from child class, which are marked as optional in the base class. For example, if the type of child class is `{x ∈ a, y ∈ bool}`.

Implementation details:

- Optional fields can be as a separate attribute of the class TypeRowOperator. Then, we don’t need to explicitly mark a field with the flag ‘?’ but just add it into the special dictionary which contains only optional fields. Actually, we can even have a list of dictionaries where each dictionary contains only fields added during one usage of the object before its definition. For example:

```
{
    x: self.z {x1: 1},  // z's optional_fields = [{x1: {x1: number}}]
    y: self.z {x2: 2},  // z's optional_fields = [{x1: {x1: number}}, {x2: number}]
    z: {x1: 3, x3: 4}
}
```

  We need to decide if we want to delete optional fields after the field is analysed and its type is inferred or keep them. If the field’s body is just a simple object like the body of `z` in the example above, then we don’t need to store optional fields since the type of `z`’s body is fully defined and doesn’t contain any special fields. If `z` would be defined as `self.foo` (just imagine that field `foo` exists), then if the type of the `foo` is unknown, we need to propagate optional fields of `z` further to the type of `foo`. But the type of `foo` is known, we only need to check if there are no conflicts between types (actually, it is a similar case to the current definition of that in the example above).

- Optional fields can have an additional purpose. We can use them to keep a connection between the type of inherited object (let’s name it `base`) and the type of inheriting object (`child`). It is useful when during inheritance the type of `base` is unknown but we need to extend `child` with fields of `base` which are not defined in `child`.
  Alternative (and maybe even better approach) is to apply topological sorting of fields. Then, we should always know the type of `base` before inheritance (assuming no cyclic definitions like `{x: self.y, y: self.x}`).

Index node (Jsonnet AST)

self.<field_name>

Currently, self.<field_name> is translated just into lambda_ast.Identifier(<field_name>). Additionally, let's assume that we already support the Required fields concept. This allows us to distinguish between the same field_names that should belong to different objects. For example:

```
{
    x: 1,
    y: {
        y1: self.x
    }
}
```

Since `x` isn't defined within the inner object (`{ y1: self.x}`), we should get an error. In our naive implementation (without 'Required fields'), this example will be considered as a valid one because the field, defined in the outer object, will be visible in the nested objects as well. But, because of the 'Required fields' concept, we will extend the nested object above with 'x: null' and its type with the special flag `!`. So, actually, we get the following object:

```
{
    x: 1,
    y: {
        y1: self.x,
        x: null
    }
}
```

As a result, we don't mix up the outer and inner `x`'s and their types. The disadvantage of such implementation is that we cannot immediately say that self.x is not defined inside `y`-object but need to analyse the final type and by seeing this special unresolved flag `!` we will return an error about an undefined field.

Alternative

Idea:

- create a new HM AST node 'Index' and translate Jsonnet AST Index to that new node;
- unify the type of 'target' with type of wrapped into object 'index' within analyse() function,
- Example:
  obj.x.s --> Index('target'=obj, 'index'=x.s) //HM AST node
  When analyse Index, call unify(analyse(obj), {x ∈ analyse(x.s)})

Import

There is no implementation of 'import' yet but let's consider its potential (and maybe desired) behaviour.

At first, let's look at an example below:

```
local foo = import 'foo.libsonnet';
{
    local l1 = foo {x: 1, y: 'y', z: 'z'},
    local l2 = foo {x: 1, y: true, z: false},
    res: {s: l1, t: l2}
}
```

Let’s say we precompiled the ‘foo.libsonnet’ and inferred that the type of its object `foo` is, for example, `{x ∈ number, y ∈ a, z ∈ a}`.

And we can define the following behaviour:

- treat an imported object as a template so it is possible to override its `null`-fields with different types;
- satisfy the type restrictions obtained in imported object, e.g. `y` and `z` should have the same type, `x` should be the number.

Something to think about:

- `import` statement can also be assigned directly to the field: `{x: import 'foo.libsonnet'}`, do we want the same behaviour as if it will be assigned to a local variable (like above)?
- can desired `import` behaviour coexist with inheritance concepts, which involve adding restrictions on the type of base object during inheritance (e.g. optional fields)?
  Let’s say we assign `foo` to another local variable `bar`. And `bar` is inherited before its type is analysed. So, we end up in the second scenario for inheritance, described here. Then, we need to process optional fields in the right way, taking into account this situation.
- as an example of `import` behaviour, we can consider diamond import, when we import the same file from two different files and then have both of them in another file again:
  ```
     ‘a’
    /   \
  ‘b’   ‘c’
    \   /
     ‘d’
  ```
  Within the file `d`, we want to treat imported objects from `b` and `c` as separate entities, and thus, every imported object within them as a copy/instantiation of the original object (in this case, this original object is from file `a`).

Type errors

Type errors occur when type inference fails. An error message contains error’s kind, incompatible types, location information, and in some cases the context with more details.

Error example:
`Type mismatch: string != number, lines 6-8, field 'name'`

The goal is to make error messages more accurate and precise

Extended HM language

To translate Jsonnet AST to HM language, we added new necessary nodes and types. Below is extended HM system that we use:

AST nodes:

- Let
- Letrec
- Lambda
- Identifier
- Apply
- LiteralNumber,
- LiteralString,
- LiteralBoolean
- LetrecAnd
- Inherit

Types:

- TypeVariable
- TypeOperator
- Function
- Integer->Number
- Bool
- String
- TypeRowOperator

where:

- <...> - unchanged nodes and types
- <...> - changed nodes and types
- <...> - added nodes and types

Short description of HM AST nodes:

- **LiteralNumber, LiteralString, LiteralBoolean** are just simple objects that contain literals of corresponding types.
- **Let** binds a value to a name inside a body. Let creates a local type environment for the specified body and adds mapping between the name and the analysed type of its value into this type environment. So, later we can search for the type of Identifier with corresponding name in that local type environment.
- **Letrec** binding is the same as Let binding but it allows a name to appear in its definition. This allows us to define recursive functions like factorial.
- **Lambda** node correspond to 'abstraction' $\lambda x.t$ in Lambda Calculus. It represents an anonymous function that is capable of taking a single input 'x' and substituting it into the expression 't'
- **Identifier** is a name that associates with some value in the type environment. We can build such an association using Let/Letrec/LetrecAnd or just put name-type binding into the environment directly.
- **Apply** represents application of function (given as first parameter) to the argument (second parameter)
- **LetrecAnd** was introduced to represent Object and Local nodes of Jsonnet AST in such a way that fields and bindings on the same level are simultaneously defined, so they are visible to definition of each other independently of the order in which they appear inside the same object.
- **Inherit** node represents inheritance. Apart from location, it has 2 attributes: base and child, which correspond to the inherited and inheriting object respectively. Currently, there are some problems

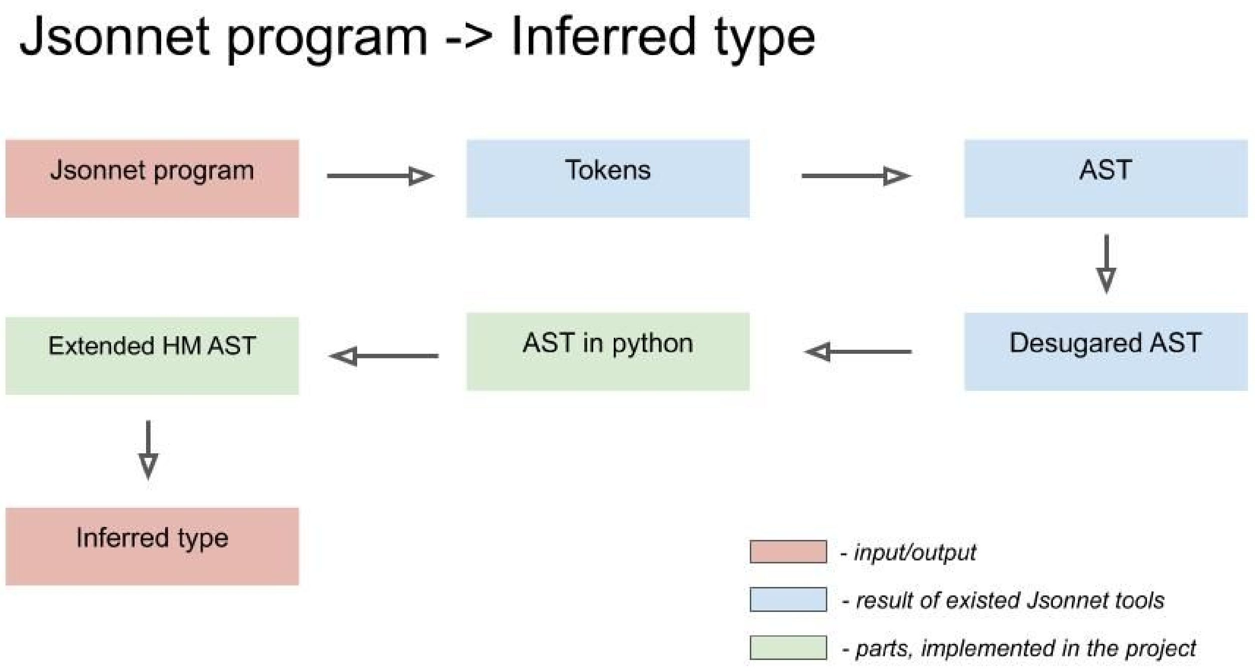
with over-approximation of a base object. Also, we may want to represent it as Apply node (Representation idea)

General information about types can be found here. In addition, a few more words about type operators in terms of type classes:

Type operators:

1. TypeOperator(name, types)
   TypeOperator is used to construct a new type from given types.
   For example, Function is TypeOpertator, where:
   - name = '->',
   - types = [from, to]
2. TypeRowOperator(fields, flags)
   TypeRowOperator is used to construct rows/records. Flags is a dictionary to keep specific info about fields, like '?' or '!' flags from row polymorphism concept. Maybe it will be better to combine flags and fields in one dictionary with instances of the new class Field, which will keep type and flags info about fields.

Diagram "Type inference stages"


Jsonnet program -> Inferred type
Jsonnet program
Tokens
AST
Desugared AST
AST in python
Extended HM AST
Inferred type
- input/output
- result of existed Jsonnet tools
- parts, implemented in the project

Warnings:

- print function that prints Desugared AST uses overloaded operator '<<'; but it will be good to introduce our own version of operator '<<' to avoid problems with double overloading of '<<' since it may also appear in the internal code of Jsonnet.

Modules

Short description of some important modules:

- `print.cpp`
  Prints desugared Jsonnet AST in such a way that printed string can be evaluated to the python version of Jsonnet AST implemented in `jsonnet_ast.py`.
- `rename.py`
  Adds prefix 'local_' to the names of local variables to distinguish them from field names.
- `translate_jsonnet_to_lambda.py`
  Translates Jsonnet AST to the extended HM AST.
  For example, the Object node is translated to the LetrecAnd node.
- `jsonnet_ast.py`
  Implementation of Jsonnet AST in python.
- `lambda_ast.py`
  Implementation of extended HM AST.
- `lambda_types.py`
  Implementation of HM types and added custom time.
- `hm_algo.py`
  Implementation of HM algorithm.
- `hm_algo_test.py`
  Contains a few unit tests with different inputs for analyse() function.
- `infer.py`
  Runs type inference program.

Open problems

1. Let's consider an example:

   ```
   {
       x: self.y {t: 1},
       y: {z: 2}
   }
   ```

   If the type of `y` was not inferred before inheritance, i.e. the type of `y` is just type variable, then the result of type unification of `self.y` and `{t: 1}` would be just type of `{t: 1}`.

   Potential solutions:

   1. During the traversal of HM AST, create a smth like temporary type that contains references on the base and child classes.Then, analysing the result type, we should be able to get rid of those temporary types.
   Potential risk: recursive dependencies that cannot be resolved during the single traversal of the type.
   2. Topological sorting

2. Analyse `std` library. Currently, `std` variable is defined as null.
3. Inheritance with more than 2 objects in a chain.
4. Translate other Jsonnet AST nodes into HM AST nodes (Array, ObjectComprehension, SuperIndex, etc.).

5. Think about an alternative way to store location information. Maybe store it in TypeVariable instead of AST nodes because type errors occur within types, not nodes. Another way to store it in the Field structure which is used to represent a field in `fields` dictionary in TypeRowOperator.
6. Undefined but used inside object fields (like `y` inside `{x: self.y}`) will be defined as null and added to LetrecAnd's bindings. But they **will** contain an empty location. Maybe, we would like to add the location of the Index node to them.
7. Visibility scope of local variables

```
1. local base = 1;
2. {
3.    local base = base,
4.    x: base
5. }
```

Second `base` in line 3 refers not to the `base` definition in line 1 but to the `base` defined in line 3. Thus, self-recursive arises which is not allowed in Jsonnet. Jsonnet interpreter will output "max stack frames exceeded" error. But we don't catch this error during static type inference because we consider the second `base` from line 3 as reference on the `base` from line one. It is not type inference error that we need to catch but this example shows that we interpret Jsonnet syntax in the wrong way. So, as a solution we can check during preprocessing if the name of the local variable is not in its definition. Actually, it is the similar case to mutual recursion which was supported by us but not allowed in Jsonnet:

```
{
   x: self.y,
   y: self.x
}
```

TODOs

1. Add unit tests (now there are unit tests only for hm_algo module).
2. Process exceptions.
3. Reimplement print_ast.cpp by introducing a new class that will play a role own implementation of '<<' operator. Then, we can overload this operator instead of '<<'.
4. Maybe, substitute 'lambda' with 'hm' in module names to highlight that we are using HM language.
5. Currently, we rename local variables by adding prefix `local_` to them. We use `replace('_', 'U_')` to make unique mapping. But this replacement is applied only to the local variables, so we cannot rely later on that only local variables have the prefix `local_`. So, apply `replace('_', 'U_')` to the fields as well if they contain `local_` prefix in their names.
   What's more, currently the names of local variables are not translated to the original name without prefix. But we need to do it before the name of the local variable will be passed into the error message (and later not only local variable but field as well).
   Examples (what is ideal to have in the end):
   1. Initially, field_name = "x",
      after renaming, field_name = "x",
      after translation to original name, field_name = "x"

2. Initially, field_name = "local_x",
   after renaming, field_name = "'localU_x",
   after translation to original name, field_name = "'local_x"
3. Initially, local_var_name = "x",
   after renaming, local_var_name = "local_x",
   after translation to original name, local_var_name = "x"
4. Initially, local_var_name = "local_x",
   after renaming, local_var_name = "local_localU_x",
   after translation to original name, local_var_name = "local_x"

Examples

Example 1

```
{
    x: 1,
    y: null,
    z: self.x + self.y
}
```

Jsonnet object:
Initially, we assign to all field names new type variable and put this information into our context (type environment) where we store the information about types of identifiers:

x ∈ **a**
y ∈ **b**
z ∈ **c**
Then, we try to unify these type variables with types of field values
x: 1
=> 1 is LiteralNumber => 1 ∈ **number**
=> **a** = **number** (here we instantiate type variable **a**) => x ∈ **number**
y: null
=> the type of null is new type variable, let's say **d**
=> **b** = **d**
z: self.x + self.y
=> we look for the type of x and y in the context and apply constraints on x and y, caused by binary plus operation, which says that type(x)=type(y)=type(z) => **b**=**d**=**a**=**c**=**number**
=> y ∈ **number**
=> z ∈ **number**
Inferred type of the whole object: **{ x ∈ number, y ∈ number, z ∈ number }**. The pattern for an object's type is defined by TypeRowOperator which is used to construct record types, like one above.

Example 2. Inheritance

So far, Inheritance is implemented in such a way that base class is kind of over-approximated. It means that we can override the base object's fields that don't have a type so far with different types in different child objects.

```
{
    local base = {
        t: null
    },
    x: base {
        t: 3
    },
    y: base {
        t: "str"
    }
}
```

We won't get an error in the example above because we don't store information after the first inheritance that a has to be of number type. As a result, we can later overload a with value of type string.

Thus, inferred type of the whole object: `{x ∈ {t ∈ number}, y ∈ {t ∈ string}}`

Example 3

Invalid Jsonnet program because function cannot be manifested in JSON, so we can assign it to the field.

```
{
  local f(base) = base {
    a: "str"
  },
  x: f
}
```

Risks

Main risk

Some Jsonnet features cannot be supported with HM type system and static analysis.

Mitigation: *we just accept the fact that we may end up with a restricted subset of Jsonnet features supported by type inference framework.*

Testing & QA

Now, there are 3 modules for testing. Two of them (*type_inference_test* and *parametrized_test*) contain integration tests, which take as an input Jsonnet program, run type inference on it, and check if the returned type is the same as the expected one. The main module with a variety of integration tests is *type_inference_test*. *parametrized_test* contains only a few tests and is aimed to show how they can be parametrized.

Currently, there is only one module with unit tests - *hm_algo_test* (remark: it needs to be extended with more test cases). As an input it takes extended HM AST and runs HM algorithm on it. The output is inferred type or the error. The goal is to check if the algorithm has expected behaviour for all HM AST nodes in different cases.

In the future, other parts of the program should also be covered with unit tests.

Alternatives

Doing nothing

Actually, Jsonnet existed without static type inference but some error messages contained a long unclear traceback so they were hard to read and to understand where the error actually occurred.

Without HM AST

Currently, we translate Jsonnet AST to extended HM AST and then perform HM algo on extended HM AST.

Alternative: implement HM algo directly for Jsonnet AST. Then, we can avoid additional translation.
The reasons against:

- current solution with translation seems to be simpler,
- initially HM algo was used for HM language so there are a lot of implementations that we used as a start point.

Use non-desugared Jsonnet AST

Currently, we use desugared Jsonnet AST before evaluating it to the corresponding python object.

Alternative: work with original Jsonnet AST (the result of Jsonnet parser without desugaring). We may need information which is desugared in the future.

The reasons against:

- desugared AST is easier to handle since some more advanced language constructs are expressed in terms of simpler constructs, so we don't need to process the AST nodes representing those advanced constructs.

Dependent types

There are a couple of things which we don’t support because we have not implemented this concept. For example, object comprehension, string formatting via %. In some cases, we can find alternatives but maybe this concept will be critical at some point.

The reasons against:

- dependent types are quite difficult because they depend on the value of expression, so we additionally need to evaluate an expression to reason about type.

Gradual typing

According to Wikipedia, “Gradual typing is a type system in which some variables and expressions may be given types and the correctness of the typing is checked at compile time (which is static typing) and some expressions may be left untyped and eventual type errors are reported at runtime (which is dynamic typing).” To perform static typing as a part of gradual typing, we need to allow type declaration (or maybe even require type declaration in some cases) in Jsonnet program. We can deal with unannotated variables by assigning them the **dynamic** type and allowing type checker to make implicit conversions:

- from the **dynamic** type to any other type (like number, string),
- from another type to the **dynamic** type.

Gradual typing has some problems:

- it has a flavor of dynamic typing and because of type conversion is unsafe,
- the problem with subtyping (see this article) which, in my opinion, will appear quite often.

Some thoughts about gradual typing in Jsonnet and its problems can be found here (see sections 3.7.1-3.7.2).

Topological sorting (TS)

The reason for applying TS to the nodes of HM AST[3] is that we want to know the type of the field or local variable before it is used.

Assumptions:

1. No cycles (add check if there is a mutual recursion or self recursion before applying TS)
2. Only local variables have a prefix 'local_'. (see *rename.py* module and this remark). This assumption is needed at the certain point of the algorithm to make difference between fields and local variables since in HM AST they both are represented as bindings of LetrecAnd.
3. Let's assume we added undefined-but-required fields, those with '!' flag. (Think if we really need this requirement)

Let's consider the following example:

```
// Example 1.
{
    x: self.y {},
    y: {y1: 3}
}
```

The type of the field y is unknown during inheritance and because of that we need to deal with partially known types and store somewhere the restrictions on the type (too early instantiation of the corresponding type variable may cause problems). So, we want to change the order of fields and local variables in such a way that ones without dependencies will be computed first. In the example above, the type of y should be inferred first, and then type of x because x depends on y.

In the next example, we see that v2 should be analysed before v1 because v1 depends on v2. And v1 should be analysed before z[4].

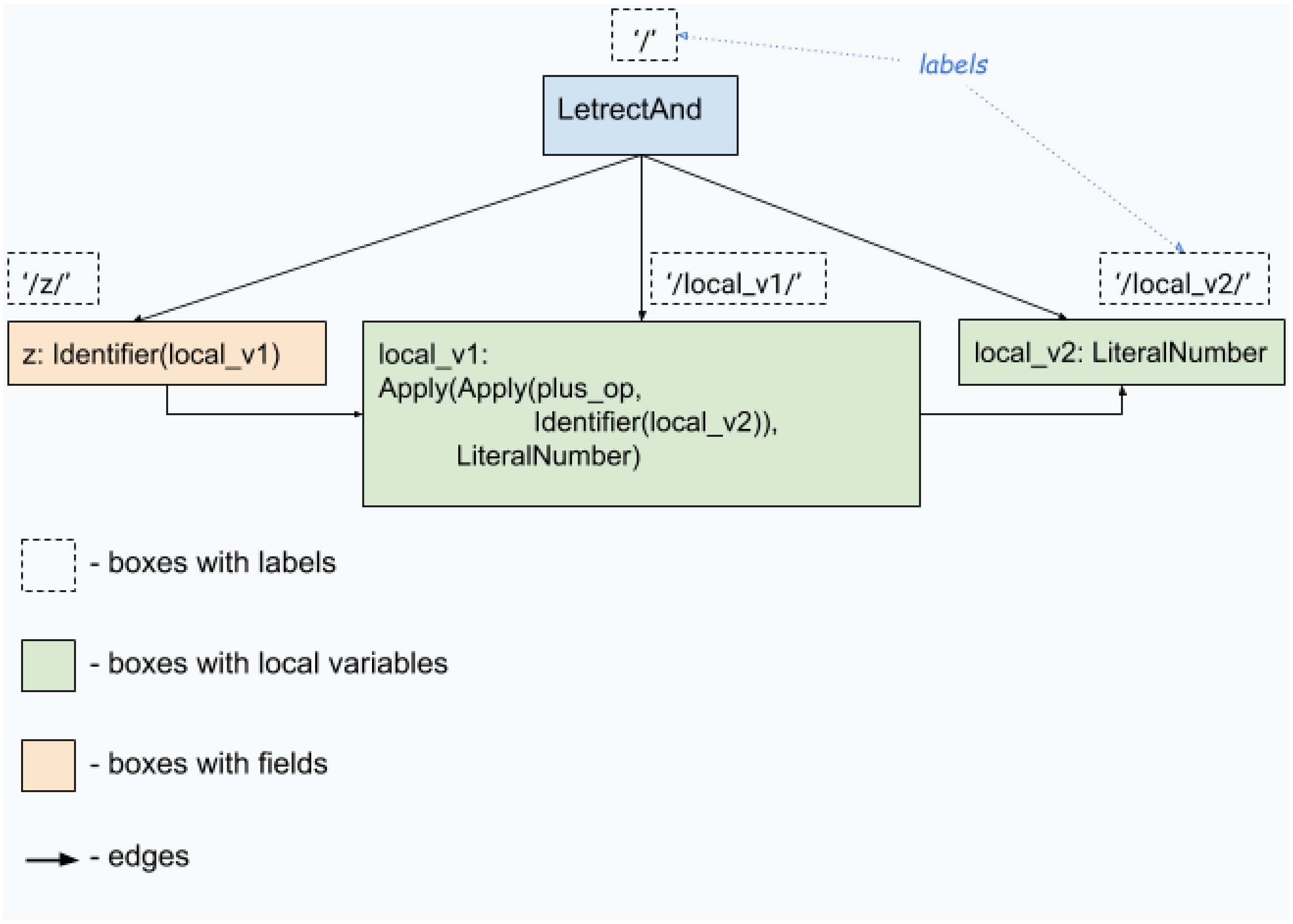
```
// Example 2.
```

[3] More precisely, we want to order bindings (which represent fields and local variables) of the LetrecAnd node.

[4] From this piece of code, it seems that v1 and v2 should be analysed before z, but due to implementation details they all are bindings within one LetrecAnd node, so we need to define proper order, in which their types will be inferred.

```
local v1 = v2 + 1, v2 = 3;
{
    z: v1
}
```

Before applying TS, we need to have a **unique label** for each AST node. Since names of fields or local variables may repeat, they cannot be used. But we can consider a unique path from the node to the AST root as possible label[5]. This path starts with '/' - root label, and ends with adding '<node_name>/'. An advantage of the 'path' label is the information about all node's predecessors up to the root[6]. Together with labels, we need to store mapping between labels and node references to know which AST part should be analysed first. Let's consider Example 2. Its corresponding HM AST with labeled nodes (labels are in the dash-border boxes):



Apart from a list of **node labels,** we need a list of **edges** - dependencies between nodes, which are built during AST traversal. Edges examples:

- (`'/'`, `'/z/'`)
  edge between labels of LetrecAnd node and one of its bindings,

[5] Alternatively, the node itself can be used as a unique label.

[6] I am not sure if we need this extra information. I thought about one case when it can be useful - if the local variable is `self` object. Then, we can use the path to go up in the tree and take the correct label of the object corresponding to `self`.

- (`‘/z/’`, `‘/local_v1/’`)
  edge between labels of the field `z` and local variable on which `z` depends.

Our edges are directed from dependent nodes to their children. However, in the original algorithm of TS, it is the other way around - an independent node (leaf node) is a node without incoming edges. So, edges need to be redirected before applying TS. For example, (`‘/’`, `‘/z/’`) should be changed to (`‘/z/’`, `‘/’`).

Since there can be dependencies between bindings of LetrecAnd, we need to know the labels of all bindings before we go deeper in any of them. Then, when we see `Identifier(local_v1)` in the body of `z`, we search in some dictionary (where names of variables and fields are mapped to labels) for the path to variable `local_v1`. So, we need to simultaneously create paths for all bindings of LetrecAnd and only then traverse their bodies.

With data we collected so far, we can perform TS. One thing to think about is that if we start to analyse leaf nodes of AST, can we apply correctly the type of fields to the record?

I thought to start from the root of AST and only change the order in which bindings of each LetrecAnd are computed. But it won’t be enough to shuffle only bindings on the same level. Using this partial sorting won’t work for the example below:

```
{
  local l1 = self.x.x1.x2.x3,
  x: {x1: {x2: {x3: 1}, y: l1}}
}
```

Also, TS won’t help in the case of functions because function’s parameters are independent nodes during function definition but when we analyse them - it is just variables which types will be known only during application of function. One idea was to analyse the function's body every time when the function is applied.

Alternative approaches for implementation of (+) binary operation

- Haskell (+):
  - class Addable a
  - instance Addable String
  - instance Addable Int
  - (+) :: Addable a, Addable b => a -> b -> AddResult a b
  - type AddResult a b
  - type AddResult String Int = String
  - type AddResult Int String = String
  - type AddResult Int Int = Int
- OCaml (=):
  - (=) : a -> a -> bool
- C++ (overloading):
  - operator+(string, int)
  - operator+(string, string)
  - operator+(int, string)

- Allow (+) operands have different types (when one of them will be inferred to be string). Then, use special type rules for (+) during HM AST analysis.