Localized Lama Gradual Typing*

*Implementing a dialect of untyped Lama language which supports user-controlled static verification sections of code

Victor Kryshtapovich
Joint master degree program of ITMO and JetBrains
2 Kantemirovskaya Str., St. Petersburg, 197342, Russia
kry127@yandex.ru

***Abstract*—Gradual typing is a modern approach for combining benefits of static typing and dynamic typing. Although scientific research aim for soundness of type systems, many of languages intentionally make their type system unsound for speeding up performance.**

This paper describes an implementation of a dialect for Lama programming language that supports gradual typing with explicit annotation of dangerous parts of code. The target of current implementation is to grant type safety to programs while keeping their power of untyped expressiveness. This paper covers implementation issues and properties of created type system. Finally, some perspectives on improving precision and soundness of type system are discussed.

***Index Terms*—programming languages, gradual typing, type safety, cast calculus**

I. Introduction

There are different approaches of type system implementation. Static type systems are well-known for preventing many undesired behaviors of the program at compile time by reasoning about possible values that expression may or may not take (e.g. Java, Haskell, ...). On the opposite side, dynamic type systems are well-known to be the most flexible type systems – low compilation prerequisites and delegation type safety to runtime allows rapid development and prototyping (e.g. Python, Racket, ...).

There is a combination of both mentioned approaches named "Gradual Typing". This technique of program typing drained a lot of attention since the article of Siek and Taha [1] was published. Article presents sound type system for Lisp dialect which represents partially typed functional language. The presence of sound system for this model language gave rise to lots of research in this field.

But practical application of sound gradual systems are still questionable because of the performance issues [2]. The key purpose of this article is to see how gradual typing and explicit unsafe code annotations can be integrated with each other as native language syntax. The desired result is to acquire language that allows programmer to control trade-off between performance and type safety. The Lama [3] version 1.00 will be used as our target language of research.

Let's imagine typical Python code, and most probably it would be some untyped piece of code. Surprisingly or not, only 3.8% of repositories have type annotations by 2020 year [4]. But the idea of gradual typing is powerful: let programmers add static type information expression by expression in the code. Thus, we can step-by-step convert untyped code into fully statically typed code with corresponding static guarantees.

This is so called *gradual* typing: on the one hand we have power of static annotations preventing us from misusing functions, modules and preserving contracts. On the other hand we shut down static type system whenever we choke down with abyss of static type errors.

The most important result of original article [1] was soundness of gradual type system. This was reached by exploiting *cast calculus* and rewriting original program with casts. The cast can be imagined as the bridge that value surpass during runtime from untyped part of code to typed part of code. This kind of "bridge" is annotated with static type and value should conform to it while moving from less typed part of code to more typed part of code. So, the main idea is to correctly insert casts and yield a program with *soundness* property.

1) If program does not typecheck, the program execution path may stuck with static type error emerged at runtime. (if there is a possibility to launch untyped programms at all)
2) If program typechecks, it can produce only dynamic type error or cast errors. *No errors involving incompatibility of static types may occur at runtime.*

In other words, if program is accepted by sound typechecker it can never fail contracts that was given to expression by the programmers in the form of types. For instance, you cannot acquire string value in variable statically typed as integer.

Gradual typing has been presented in several languages and in various forms, such as:

1) Python [5] [6] (MyPy [7] and PyType [8] projects)
2) Typed Racket [9]
3) JavaScript: TypeScript
4) C♯ 4.0 with dynamic keyword

Although they are all have gradual typing property (in the sense, that not all objects have known type at compile time), their implementation of gradual type system has strong differences. Some of them are compiled into dynamic target language, such as TypeScript program is converted to pure JavaScript after compilation. Some of them are static by the nature as C♯ and then bring up a "dynamic" keyword

which marks that object has unknown type until runtime. Some of them incorporate optional typing annotations and leave them alone for documentation and external tools (linters, typecheckers, IDE) as Python do.

The most noticeable state-of-the-art of gradual typing: every industrial-level language doesn't care much about soundness of the type system. This is because of the performance issues. Some real programs exhibit slowdown over $20\times$, likely rendering them unusable for their actual purpose [2]. To increase performance many of them reduce amount of dynamic casts, or remove them at all. This leads to trade-off between soundness and performance of gradually typed language.

To sum up, gradual typing provides mechanism to check program correctness having this pros and cons:

- Types can be added ad hoc by the programmers
- Gradual type system can be sound in certain languages (more frequently academic ones)
- Dynamic typechecks is giving significant overhead at runtime

No doubt: looking at the diversity of implementation and approaches it is interesting to look at the result of implementation of gradual typing in the language with different model of computation and semantics. We will test some new syntax conceptions experimenting with Lama programming language.

$\lambda^{\alpha}\mathcal{M}^{\alpha}$ is a programming language developed by JetBrains Research for educational purposes as an exemplary language to introduce the domain of programming languages, compilers and tools [3]. The most noticeable property of this language that it is fundamentally untyped. The reference manual says, that the lack of a type system is an intentional decision which allows to show the unchained diversity of runtime behaviors. But at the same time manual says that the language can be used in future as a raw substrate to apply various ways of software verification (including type systems) on [10]. So why wouldn't we try to implement some kind of type system upon it?

In our work we will test new approach of combining parts of code where different rules of static verification are applied: some parts of code will be gradually typed, and some parts of code will be left untyped. The expected result is programming language that can mix two types of code:

- with semantics that respects type safety in necessary parts of the code (e.g. sound)
- with original semantics without overheads

This should allow programmer to choose what parts of program should be gradually typed, and what parts of program should not be typed.

Another expected result is producing a program with decreasing speed of execution of gradually typed code . The slowdown may be arbitrary, but we will try to reproduce results from article [2] (at least $\times 2$ slowdown).

II. Examples

To give reader a proof of concept we should consider concrete syntax and pragmatics of the pieces of code written in Lama and describe how to introduce types into our language and what they expected to do.

Normally, code in Lama looks as follows. No types, just anarchy of undefined behaviors:

```
fun closure(x) {
  fun (y) {
    2*x*y
  }
}
```

In this example we see function that takes `x` as an argument and returns function that multiplies input argument by `2 * x`. One expects it to be used upon integers, but Lama won't restrict to call function like `closure("Hello, ")("world!")` and pray for runtime not to fall. We can use type annotations to designate our intentions about the code like so:

```
fun closure(x :: Int) :: Int -> Int {
  fun (y :: Int) :: Int {
    2*x*y
  }
}
```

What do we expect from introduced type annotations?

- Backward compatibility with existing untyped source code
- Static compile-time checks
- Dynamic runtime checks

Moreover, we would like something like type inference.

```
fun closure(x :: Int) {
  fun (y :: Int) {
    2*x*y
  }
}
```

If `x` and `y` have known at compile time types, then type of the functions can be inferred: inner function has type `Int -> Int`, and outer function has type `Int -> Int -> Int`.

Moreover, Lama nowadays supports operations only with integer constants, i.e. `Int`. If we take a closer look to the untyped example, it can be inferred that `x` should have type `Int`, `y` should have type `Int`, because they are used in expression like `2 * x * y`, and further infer function types, which makes this concrete piece of code fully typed.

At first glance type inference seems to be contradictory with backward compatibility. That is because some of the untyped expressions become implicitly typed, as first example do. Thus, runtime typechecks are inserted in parts of code that were initially untyped, which affects their semantics. Thankfully, the developers of Lama left regression tests that check backward compability. So we can bring up type inference features with awareness on backward compability.

Another example of typing Lama programs is pattern matching

```
fun processA(a) {
  case a of
    A (0) -> "1"
  | A (x) -> "2"
  esac
}
```

The `A(0)` notation is so called S-expression [11]. *Quick Lama-specific introduction*: you can consider S-expression as labeled array of arbitrary values. Name should be capitalized, amount of values is not bounded. Two S-expression labels are considered equal in Lama if their five first letters are the same, so `Branch(Leaf, Leaf, 3)` and `Branc(Leaf, Leaf, 3)` are equal S-expressions. By the way, `Leaf` is nested S-expression with zero values in it, so brackets are optional for zero-arity S-expressions.

Side note: S-exprs like `Int` and `Str` has type `Int :: Int()` and `Str :: Str()` to distinguish them from integers (`3 :: Int`) and strings (`"smoothie" :: Str`) type.

Back to our `processA` function, we can see, that if `a` matches `A(0)`, then "1" produced, for other value `A(smth)` where `smth` is not `0` we would get "2" produced by the function. If we call `processA(B(0))` we would get runtime error from pattern matching. So, other things that we would like from our type system are:

- Check that all branches cover matching expressions. E.g. no runtime error would occur in pattern matching
- Check branches that would never succeed: either covered by previous branch or just don't conform to matching expression

For example, type system should reject this Lama program:

```
local foo = fun (x :: A(Int)) {
case x of
    A (0) -> "1"
  | A (x, y) -> "3" anything
esac
};
```

Here type system can check two things. First of all, `x = A(1)` won't meet any branch, so not whole possible values of `x` are covered. And the second: `A(x, y)` would never match values with type `x :: A(Int)`.

Also note, that functions in Lama has beautiful sugar that combines pattern matching, that can be used to check input arguments:

```
public fun id2 (Abc (x, y)) :: ? {
  x
}
write(id2(Abc(6, 8)));
write(id2(Xyz(6, 8))); -- static fail
```

The last example that we should consider is connected with runtime checks. Let's look at this simple piece of code:

```
fun intStringer(x :: Int) {
  x.string
}
local dyn :: ? = "Can be anything";
dyn := intStringer; -- forget type
dyn("input") -- should it fail?
```

variableDefinitionItem : LIDENT [:: *typeExpression*] [= *basicExpression*]
functionDefinition : [public] fun LIDENT (*functionArguments*) *functionBody* [:: *typeExpression*]
functionArguments : [*funArgItem* (, *funArgItem*)*]
funArgItem : *simplePattern* [:: *typeExpression*]

Fig. 1. Syntax extension: scope expressions with type annotations.

At first glance it is unclear, where is the problem, because `dyn("input")` would reduce to `"input".string` and then to `"input"`. Do we actually care about function, that originally takes `Int` and store it at runtime?. The answer is yes:

```
fun intStringer(x :: Int) {
  (x + 1).string
}
```

Of course, if we try to reduce `dyn("input")` we get `"input" + 1`, and then we'll now end up with runtime error of casting `"input"` to `Int`. But what is the real cause of this error, whom to *blame* [12] [13] [14] for this mess – a plus operator, or input to the `intStringer`? That's why we should check function arguments wrapping them with appropriate dynamic casts. So if follow blame ideology in both implementations `dyn("input")` would fail with the same reason: function expected `Int`, but given `Str`. But this solution could lead to extra checks and execution speed decrease.

After seeing quite a bit of examples we conclude that these features would be handful in untyped Lama language. Typechecker would decrease amount of errors in code made by programmers and runtime casts would inform programmer when untyped code does not conform contracts of the typed code. In next chapter we will define syntax of gradual types and their semantics.

III. Type annotations definition and semantics

Gradual typing assumes that user annotates parts of the program with certain type. So we should provide this feature in Lama compiler.

Syntax rules have been described in Lama specification. We'll fix them a little bit, because we only change variable definition (global and scope), function definition and their input parameters, look at p. 10 [10] for more detailed language syntax specification.

We slightly modified this nonterminals on the "Fig. 1": just put static type annotations to variable definition and function definition. Also, nonterminal *functionArguments* was slightly

```
typeExpression : typeUnion
               | typeArrow
               | typeSexp
               | typeArray
               | typeAny
               | ( typeParser )
typeAny        : ?
typeArray      : [ typeExpression ]
typeSexp       : UIDENT[ ( typeList0, ) ]
typeArrow      : typeExpression → typeExpression
               | ( typeList0, ) → typeExpression
typeUnion      : Union [ typeList0; ]
typeList0,     : [typeExpression( , typeExpression)*]
typeList0;     : [typeExpression( ; typeExpression)*]
```

Fig. 2. Typing expressions syntax.

$$
\begin{aligned}
\tau := \;& \texttt{TAny} \mid \texttt{TConst} \mid \texttt{TString}(s) \\
& \mid \texttt{TArr}(\tau) \mid \texttt{TRef}(\tau) \mid \texttt{TLambda}(\overline{\tau}, \tau) \\
& \mid \texttt{TSexp}(s, \overline{\tau}) \mid \texttt{TUnion}(\overline{\tau}) \mid \texttt{TVoid}
\end{aligned}
$$

Fig. 3. Typing expressions semantics.

changed in comparison to specification to respect pattern matching sugar. This sugar is not included in concrete syntax definition for some reason. Other nonterminals assumed taken from chapter "Concrete syntax and semantics" of specification [10].

The definition of type annotations *typeExpression* is presented on figure 2. It semantic (see τ in "Fig. 3") is almost straightforward: syntax rule *typeAny* corresponds to dynamic type `TAny`, which can hold arbitrary value. Syntax rule *typeArray* corresponds to the array `TArr` of certain type. Syntax rule *typeSexp* corresponds to `TSexp` with parsed `UIDENT` as the name of S-expression and list of types forming type of S-expression. Syntax rule *typeArrow* corresponds to arrow `TLambda`. Note that input arguments can vary from zero to arbitrary amount. Syntax rule *typeUnion* corresponds to `TUnion` and lists all types that value can conform.

Only *typeSexp* rule with zero arity has non straightforward semantics. If type parameters of S-expression type are not presented, and UIDENT is one of the

- `Int` – corresponds to integers $\tau =$ `TInteger`
- `Str` – corresponds to strings $\tau =$ `TString`
- `Void` – corresponds to empty set of values $\tau =$ `TVoid`
- otherwise it corresponds to S-expression with specified name and no arguments.

If *typeSexp* is specified with brackets, it has straightforward semantics of S-expression. So, for example, `Cons` and `Cons()` has the same semantics of `TSexp("Cons")`, but semantics of `Int` and `Int()` are different as integer and S-expression types: `TConst` and `TSexp("Int")` correspondingly.

$$
\begin{aligned}
e := \;& \\
& \texttt{Const}(i) \mid \texttt{Arr}(\overline{e}) \mid \texttt{String}(s) \mid \texttt{Sexp}(s, \overline{e}) \\
& \mid \texttt{Var}(s) \mid \texttt{Ref}(s) \mid \texttt{Cast}(e, \tau) \mid \texttt{Binop}(s, e, e) \\
& \mid \texttt{Elem}(e, e) \mid \texttt{ElemRef}(e, e) \mid \texttt{Length(e)} \\
& \mid \texttt{StringVal}(e) \mid \texttt{Call}(e, \overline{e}) \mid \texttt{Assign}(e, e) \\
& \mid \texttt{Seq}(e, e) \mid \texttt{Skip} \mid \texttt{If}(e, e, e) \mid \texttt{While}(e, e) \\
& \mid \texttt{Repeat}(e, e) \mid \texttt{Case}(e, \overline{(p, e)}) \mid \texttt{Return}(e) \\
& \mid \texttt{Ignore}(e) \mid \texttt{Scope}(\overline{(s, e)}, e) \mid \texttt{Lambda}(\overline{(s, \tau)}, e, \tau)
\end{aligned}
$$

Fig. 4. Lama expression class.

$$
\begin{aligned}
v := \;& \\
& \texttt{VVar}(s) \mid \texttt{VElem}(v, i) \mid \texttt{VInt}(i) \mid \texttt{VString}(s) \\
& \mid \texttt{VArray}(\overline{v}) \mid \texttt{VSexp}(s, \overline{v}) \mid \texttt{VClosure}(\overline{s}, e, \sigma) \\
& \mid \texttt{VFunRef}(s, \overline{s}, e, i) \mid \texttt{VBuiltin}(s) \mid \texttt{VCast}(v, \tau)
\end{aligned}
$$

Fig. 5. Lama value class.

IV. Typechecking rules

The typechecking is inserted in the compilation pipeline directly after AST (Abstract Syntax Tree) representation of the program has been built (see "src/Language.ml" and "src/Driver.ml" in Lama source code [3]). The typechecking simultaneously performs the following procedures with AST: type checking, type inference and cast insertion.

For detailed description of this three type system problems we need to describe such classes as *expressions*, *values*, *patterns* and *types* of the language.

- τ is class of type expressions (see "Fig. 3")
- e is class of expressions (see "Fig. 4")
- v is class of values (see "Fig .5")
- p is class of patterns (see "Fig. 6")

There is also additional classes that are built-in of implementation language (OCaml). They can be considered as value class:

- i – integer
- s – string

The class σ in `VClosure` represents state which captured by closure and is not interesting in our research.

Let's denote set of variables by $\mathbb{V}$, which represented by OCaml string s, and set of types $\mathbb{T}$. We should think about $\mathbb{T}$ wider, that types induced by type constructors of "Fig. 3". In other words, some type $\gamma \in \mathbb{T}$ may not be expressed with type constructors.

If we simplify process of compilation a little bit and ignore external symbol resolvance, Lama parser generates expression of e class without `Cast` constructors, i.e. pure untyped Lama expression. Notice, that expression can also contain patterns p due to pattern matching in `Case` expression.

$$
\begin{aligned}
p := \;& \texttt{PWildcard} \mid \texttt{PConst}(i) \mid \texttt{PString}(s) \\
& \mid \texttt{PArray}(\overline{p}) \mid \texttt{PSexp}(s, \overline{p}) \mid \texttt{Named}(s, p) \\
& \mid \texttt{PBoxed} \mid \texttt{PUnBoxed} \mid \texttt{PStringTag} \\
& \mid \texttt{PSexpTag} \mid \texttt{PArrayTag} \mid \texttt{PClosureTag}
\end{aligned}
$$

Fig. 6. Lama pattern class.

$$\frac{}{\tau \sim \texttt{TAny}} \;\texttt{[ConfTAny]}$$

$$\frac{}{\texttt{TAny} \sim \tau} \;\texttt{[ConfTAny2]}$$

$$\frac{\tau \sim \tau'}{\texttt{TArr}(\tau) \sim \texttt{TArr}(\tau')} \;\texttt{[ConfTArr]}$$

$$\frac{\tau \sim \tau'}{\texttt{TRef}(\tau) \sim \texttt{TRef}(\tau')} \;\texttt{[ConfTRef]}$$

$$\frac{s = s' \wedge \left[\bigwedge_{i=1}^{n} \tau_i \sim \tau_i'\right]}{\texttt{TSexp}(s, \tau_1 \dots \tau_n) \sim \texttt{TSexp}(s', \tau_1' \dots \tau_n')} \;\texttt{[ConfTSexp]}$$

$$\frac{\begin{array}{c}\texttt{[ConfTLambda]} \\ \tau_r \sim \tau_r' \wedge \left[\bigwedge_{i=1}^{n} \tau_i' \sim \tau_i\right]\end{array}}{\texttt{TLambda}(\tau_1 \dots \tau_n, \tau_r) \sim \texttt{TLambda}(\tau_1' \dots \tau_n', \tau_r')}$$

$$\frac{\bigwedge_{i=1}^{n} \tau_i \sim \tau'}{\texttt{TUnion}(\tau_1 \dots \tau_n) \sim \tau'} \;\texttt{[ConfTUnion1]}$$

$$\frac{\bigvee_{i=1}^{n} \tau \sim \tau_i'}{\tau \sim \texttt{TUnion}(\tau_1' \dots \tau_n')} \;\texttt{[ConfTUnion2]}$$

$$\frac{\tau = \tau'}{\tau \sim \tau'} \;\texttt{[ConfTGround]}$$

Fig. 7. Rules of type conformance to the other type.

Then, we have some options how to deal with generated AST. The trivial option is to left expression untouched and get the semantics of classic Lama language. The first option is try to statically typecheck expression. If we succeed to acquire static type of program represented as whole expression, we can conclude that there are no static misuse of typed expressions. The second option is to transform AST to insert casts where values are passing from untyped parts of code to typed one. We will build up an algorithm that makes static typechecking and dynamic cast insertion simultaneously.

For type checking we need to answer a question: does some type $\tau_1 \in \mathbb{T}$ conforms to other type $\tau_2 \in \mathbb{T}$? That answer is given by $\sim$ relationship named "conforms" which is constructed by axioms presented at "Fig. 7".

We should put additional attention to `TUnion` type and it's rules. It denotes type that holds all possible values which can hold its constituent types. It is naturally comes from such language expressions as `If`, `Case` and `Return`. We've chosen set-theoretic approach on typing such expressions. Although there is an algorithm for union contraction, set-theoretic approach for type combination may lead to certain drawback in correctness and decreased performance during compile time.

Speaking about correctness: rules `ConfTUnion1` and `ConfTUnion2` generally cannot proof that two type representation conform to each other if they really do. Thus, the lack of completeness is reflected in false positives generated by static typechecker. That means correct type-annotated Lama expressions can be rejected by typechecker with such relationship definition $\sim$. This is a common illness of every static typechecker, because we would like to check nontrivial property of the code: to be statically correct [15].

But the good news is that no type intersections `TIntersection` or type substractions `TSubstraction` are coming – we try to avoid them when building type system for Lama.

Now we can make an analogy of $\sim$ for expression e and type τ. But instead we will be inferring type of expression. To start with something simple let's define type inference for patterns (see "Fig. 8").

Notice, that we infer both lower and upper bound for pattern type. This interval style inference of patterns is crucial for analyzing case expressions. Let's denote $\tau_l(p) \in \mathbb{T}$ for lower bound inferred type for pattern and $\tau_r(p) \in \mathbb{T}$ for upper bound inferred type for pattern. Notation $\tau(p)$ means theoretic set of all possible values that are captured by pattern p. With the chosen type constructors and their semantics we can conclude:

1) τ_r is representing type that covers all possible values captured by pattern (upper bound)
2) τ_l is representing type that is covered by all possible values captured by pattern (lower bound)

For example, value $\texttt{Suc}(1)$ has type $\texttt{TSexp}(\text{"}\texttt{Suc}\text{"}, \texttt{TConst})$, but this value alone covers almost nothing, so $\texttt{TVoid} \sqsubset \{\texttt{Suc}(1)\} \sqsubset \texttt{TSexp}(\text{"}\texttt{Suc}\text{"}, \texttt{TConst})$.

Now we are ready to describe our main part of algorithm: type inference and cast insertion for Lama expressions. We will use such notation: $e \mapsto e' : \tau$. That means that expression e has type τ, and cast insertion into that expression produces expression e', which has the same type τ. In addition, we have two types of contexts: $\Gamma : \mathbb{V} \to \mathbb{T}$ for typing context of variables (assigns types to variable typenames) and set of types $\Delta \subset \mathbb{T}$ for collecting information about function return type. Then, typechecker by given context and collected return types produce another collection of return types (probably, bigger than the original), expression rewritten with casts and it's type. So, the full notation of this algorithm should be:

$$\Gamma, \Delta \vdash e \mapsto \Delta' \vdash e' : \tau$$

"Fig. 9" and "Fig. 10" presenting all set of rules for type inference of Lama expression with $\Gamma, \Delta \vdash e \mapsto \Delta' \vdash e' : \tau$ notation used. Let's highlight some features about presented algorithm.

The set of return types for expression Δ is initialized with $\varnothing$. Note, that initial context Γ maps every variable occurrence to type `TAny`. The typechecker does not check, is symbol is defined in upper scopes or correctly imported, but context is called to provide correct surrounding type information for expressions.

$$\frac{}{\texttt{TAny} \sqsubset \tau(\texttt{PWildcard}) \sqsubset \texttt{TAny}}\ \texttt{[InferPWildcard]}$$

$$\frac{\tau_l \sqsubset \tau(p) \sqsubset \tau_r}{\tau_l \sqsubset \tau(\texttt{PNamed(s, p)}) \sqsubset \tau_r}\ \texttt{[InferPNamed]}$$

$$\frac{}{\texttt{TVoid} \sqsubset \tau(\texttt{PConst}(i)) \sqsubset \texttt{TConst}}\ \texttt{[InferPConst]}$$

$$\frac{}{\texttt{TVoid} \sqsubset \tau(\texttt{PString}(s)) \sqsubset \texttt{TString}}\ \texttt{[InferPString]}$$

$$\frac{}{\texttt{TConst} \sqsubset \tau(\texttt{PUnboxed}) \sqsubset \texttt{TConst}}\ \texttt{[InferPUnboxed]}$$

$$\frac{}{\texttt{TString} \sqsubset \tau(\texttt{PStringTag}) \sqsubset \texttt{TString}}\ \texttt{[InferPStrTag]}$$

$$\frac{}{\texttt{TVoid} \sqsubset \tau(\texttt{PSexpTag}) \sqsubset \texttt{TAny}}\ \texttt{[InferPSexpTag]}$$

$$\frac{}{\texttt{TVoid} \sqsubset \tau(\texttt{PClosureTag}) \sqsubset \texttt{TAny}}\ \texttt{[InferPClosureTag]}$$

[InferPBoxed]

$$\frac{}{\texttt{TUnion(TString, TArr(TAny))} \sqsubset \tau(\texttt{PBoxed}) \sqsubset \texttt{TAny}}$$

[InferPArrTag]

$$\frac{}{\texttt{TArr(TAny)} \sqsubset \tau(\texttt{PStringTag}) \sqsubset \texttt{TArr(TAny)}}$$

[InferPSexp]

$$\frac{\tau_i \sqsubset \tau(p_i) \sqsubset \tau_i'}{\begin{array}{c}\texttt{TSexp}(s, \tau_1 \dots \tau_n) \sqsubset \\ \tau(\texttt{PSexp}(s, p_1 \dots p_n)) \sqsubset \texttt{TSexp}(s, \tau_1' \dots \tau_n')\end{array}}$$

[InferPArray]

$$\frac{\tau_i \sqsubset \tau(p_i) \sqsubset \tau_i'}{\begin{array}{c}\texttt{TArr}(\texttt{TUnion}(\tau_1 \dots \tau_n)) \sqsubset \\ \tau(\texttt{PArr}(p_1 \dots p_n)) \sqsubset \texttt{TArr}(\texttt{TUnion}(\tau_1' \dots \tau_n'))\end{array}}$$

Fig. 8. Rules of lower and upper bound type inference for patterns.

Notation $\tau \in \langle \texttt{TSexp}, \texttt{TString}, ... \rangle$ in rule `[InferLength]` means that τ's top level constructor should be one of the listed in angle brackets.

In rule `InferCall` cast to `TAny` is optional. It is used in inference rules to be consistent with `InferCall3` rule which process call of the union type object.

Many of the rules can be simplified by removing Δ because they don't change it, such as `InferArr` and `InferSexp`, et cetera. That is because they recompute Δ for expressions that never change Δ in correct Lama expressions. There are a few places where Δ is really useful: it is `InferLambda`, `InferReturn1` and `InferReturn2` rules. Notice, that we inferring return type of the function just to acknowledge that it fits type declared by the user, the declared interface is not changing. *But if the type is not specified by user, the inferred type for variable will be used implicitly*.

Also notice rules in `InferCase`. First of all, we collect return types from the branches while dragging Δ through the computation pipeline. The second one, look at notation $\Gamma \cup \tau_\Gamma(p_i)$ – it fulfills typing context with mapping of `PNamed` named pattern to its types. The τ_Γ can be defined via τ_r as follows:

$$\tau_\Gamma(p) = \begin{cases} \tau_\Gamma(p') \cup \{s : \tau_r(p)\} & p = \texttt{PNamed}(s, p') \\ \bigcup_{i=1}^{n} \tau_\Gamma(p_i) & p = \texttt{PArr}(p_1, \dots, p_n) \\ \bigcup_{i=1}^{n} \tau_\Gamma(p_i) & p = \texttt{PSexp}(s, p_1, \dots, p_n) \\ \varnothing & \text{otherwise} \end{cases}$$

The third one about `InferCase` is that there is a check that all branches cover target type: $\omega \sim \texttt{TUnion}(\overline{\tau_l(p_i)})$. And the fourth: notice that each pattern is checked for code execution availability $\tau_r(p_i)\ \omega$, and at the same time we check that branch is not hidden by earlier branch $\tau_r(p_i) \nsim \tau_l(p_j)$. According to inequalities

$$\tau_r(p_i) \sim \tau_l(p_j) \Longrightarrow \tau(p_i) \sqsubset \tau_r(p_i) \sqsubset \tau_l(p_j) \sqsubset (p_j)$$

In other words, when expression holds, it is certain that pattern p_i was covered by more recent cover p_j. In that way we eliminated the need of introduction of intersection or difference types in our type system. But it doesn't mean we cannot deal with intersection and difference types, see [18] or [19] for example of polymorphic type system that handles that.

The most complex is `[InferScope]` rule. It is intentionally simplified, because it's implementations is more subtle. Here it simply overwrites variable or function definition and updates context Γ. But implementation also checks, that previous usage is corresponding with current typing when no expression is provided to variable. But to describe that strictly we would need to introduce a class for declarations and this rule would get even more complex.

So, this rule lead to new language feature – type usage of expression inside the scope:

```
{
  f :: Int -> Str;
  g :: Int -> Int;
  f(g(0)); -- ok
  f(g(D(0))) -- error
};
{
  f :: D(Int) -> Str;
  g :: D(Int) -> D(Int);
  f(g(0)); -- error
  f(g(D(0))); -- ok
  {
    f :: [Int] -> Int;
    g :: Str -> [Int];
    f(g("hello, world"))
  }
};
```

Other type checking rules either trivial or common in corresponding field of study [16] [17], so we wouldn't dive too deep into them. In next chapter we will discuss performance issues of our typechecking algorithm.

V. Cast performance analyzing

It is obvious that rules presented at "Fig. 9" introduce new kind of expression `Cast`(e, τ). It's runtime semantics is simple: when expression e evaluates to value v, we should check that value v corresponds to type τ. If v conforms to e, the result of evaluation `Cast`(e, τ) is v, otherwise cast error $\perp$ produced as the result.

Consequently, checking that value corresponds to some type may be time consumptive, especially when type and expression are complex and have big nesting. Thus, we can introduce an explicit syntax for parts of code where we wish not to insert casts like this:

```
fun mod(x :: ?, m :: ?) :: ? {
  #NoTypecheck {
    (if x < 0 then 0-x else x fi) % m
  }
}
```

Typechecker will see this annotation and completely ignore annotated part of code.The implementation of gradual typing for Lama offers us three options to maintain typechecking procedure:

- `#NoTypecheck` – drops AST from typechecking at all
- `#StaticTypecheck` – disables cast insertion into AST, but static checks are still performing
- `#GradualTyping` – enables cast insertion into AST

You can nest `#StaticTypecheck` and `#GradualTyping` annotations in order to disable or enable cast insertion while type checking. But there is no point to nest type related information into `#NoTypecheck` annotation, because they would be completely ignored by typechecker.

Having all power of gradual types and unchained diversity of undefined behaviors, let's use interpretation mode of Lama compiler to see the slowdown in the code execution. We will use sample code:

```
fun fibonacci(k) {
  if   k == 0 then return  0
  elif k == 1 then return  1
  elif k <  0 then return -1
  else return fibonacci(k-1)
            + fibonacci(k-2) fi
}
write(fibonacci(read()))
```

It is not obvious where are the casts in this example, but in section II we have noticed, that + operator coerces both it's arguments to `Const` at runtime, so appropriate casts to `TConst` types from unknown type are inserted. Hence, this code models situation of frequent value passage from untyped part of code to typed part of code.

We will compare this code wrapped in `#GradualTyping` which is the default, and `#NoTypecheck` annotations. The time measurement is performed with Unix `time` utility, thus compile time included in both measures.

n	Untyped	Typed
10	0m 0,119s	0m 0,092s
11	0m 0,097s	0m 0,079s
12	0m 0,088s	0m 0,087s
13	0m 0,094s	0m 0,093s
14	0m 0,091s	0m 0,095s
15	0m 0,086s	0m 0,090s
16	0m 0,092s	0m 0,094s
17	0m 0,095s	0m 0,088s
18	0m 0,093s	0m 0,100s
19	0m 0,102s	0m 0,105s
20	0m 0,106s	0m 0,125s
21	0m 0,124s	0m 0,124s
22	0m 0,132s	0m 0,154s
23	0m 0,162s	0m 0,192s
24	0m 0,208s	0m 0,279s
25	0m 0,284s	0m 0,389s
26	0m 0,416s	0m 0,581s
27	0m 0,593s	0m 0,878s
28	0m 0,909s	0m 1,363s
29	0m 1,467s	0m 2,179s
30	0m 2,326s	0m 3,561s
31	0m 3,659s	0m 5,796s
32	0m 5,977s	0m 9,469s
33	0m 9,477s	0m14,108s
34	0m15,981s	0m24,799s
35	0m26,933s	0m43,855s
36	0m42,236s	1m 7,766s
37	1m12,161s	1m49,319s
38	1m53,534s	3m 0,748s
39	3m18,046s	4m54,461s
40	5m17,664s	7m54,811s

The average of slowdown $sd_n = \frac{t_n^{\texttt{Typed}}}{t_n^{\texttt{Unyped}}}$ from the point of actual slowdown registered $n = 21$ is:

$$\frac{1}{20}\sum_{n=21}^{40} sd_n = \frac{1}{20}\sum_{n=21}^{40} \frac{t_n^{\texttt{Typed}}}{t_n^{\texttt{Unyped}}} \approx 1.45$$

As we can see, section of code with active gradual typing static checks exhibit almost $\times 1.5$ slowdown. Thus we have reproduced the results of an article [2], even on this artificially small example.

VI. Conclusion

We introduced type system with following properties:

- Monomorphic
- Gradual

It would be nice to introduce such features in type system as:

- Polymorphism
- Recursive types

In the future work it is desired to use type equations and Hindley-Milner style inference with unification algorithm as presented in [18] and [20].

It is worth to mention the reproduction of the result of a recent article about industrial-level languages that use gradual types unsoundly [2]. We have modeled the situation of values constantly transiting from untyped part to typed parts of program and expectedly acquired slowdown of execution.

In addition, we have provided a simple and powerful, yet dangerous, method of maintaining trade-off between type safety and execution performance: let programmer choose areas of code where he needs extra performance and where he needs static and runtime type safety guaranties, either with `#NoTypecheck`, or better with `#StaticTypecheck` and `#GradualTyping` annotations.

The idea goes further. It would be nice to introduce some other sections of static verification that programmers can apply at their taste. For instance, live variable analysis `#LiveVarAnalysis`, or memory access safety. Thus, programmer acquire framework with bunch of static verifiers and the ability to choose what guaranties is the most important at applied piece of code. To sum up, programmer mantains compilation time and acquires code with the needed guarantees unified in one syntax.

Even though the type system soundness is still questionable and should be proved or improved, several tests are added to codebase to check type system, including not compiling tests, runtime error tests and positive example tests. Introduced type system enhances coding experience and points out at least silly and obvious errors that programmers are frequently making. More over, Lama's facility has been extended by logger to generate warning messages, mostly for case expression coverage.

The implementation of gradual typing for Lama language resides in personal repository within branch named "GraduLama" [21].

REFERENCES

[1] Siek, Jeremy G.. "Gradual Typing for Functional Languages." (2006).

[2] Cameron Moy, Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. Proc. ACM Program. Lang. 5, POPL, Article 53 (January 2021), 28 pages. DOI:https://doi.org/10.1145/3434334

[3] D. Boulytchev. " JetBrains-Research / Lama" source code https://github.com/JetBrains-Research/Lama Request timestamp: 27/03/2021 20:42

[4] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2020). Association for Computing Machinery, New York, NY, USA, 57–70. DOI:https://doi.org/10.1145/3426422.3426981

[5] Guido van Rossum, Ivan Levkivskyi. "PEP 483 – The Theory of Type Hints"https://www.python.org/dev/peps/pep-0483/ Request timestamp: 27/03/2021 22:10

[6] Guido van Rossum, Jukka Lehtosalo, Łukasz Langa. "PEP 484 – Type Hints" https://www.python.org/dev/peps/pep-0484/ Request timestamp: 27/03/2021 22:08

[7] Jukka Lehtosalo et al. "Mypy: Optional Static Typing for Python" https://github.com/python/mypy Request timestamp: 27/03/2021 22:15

[8] "Pytype: A static type analyzer for Python code" https://github.com/google/pytype Request timestamp: 27/03/2021 22:17

[9] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, Asumu Takikawa. "The Typed Racket Guide". https://docs.racket-lang.org/ts-guide/index.html Request timestamp: 27/03/2021 22:19

[10] D. Boulytchev. "Lama language specification v. 1.10." https://github.com/JetBrains-Research/Lama/blob/1.10/lama-spec.pdf Request timestamp: 27/03/2021 19:39

[11] R. Rivest, "S-Expressions", 4/05/1997 http://people.csail.mit.edu/rivest/Sexp.txt Request timestamp: 29/03/2021 17:48

[12] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. Proc. ACM Program. Lang. 1, ICFP, Article 39 (September 2017), 28 pages. DOI:https://doi.org/10.1145/3110283

[13] Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The root cause of blame: contracts for intersection and union types. Proc. ACM Program. Lang. 2, OOPSLA, Article 134 (November 2018), 29 pages. DOI:https://doi.org/10.1145/3276504

[14] Wadler, P. (2015). A Complement to Blame. SNAPL.

[15] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. Proc. Trans. Amer. Math. Soc. 74, pages 358-366. DOI:https://doi.org/10.1090/S0002-9947-1953-0053041-6

[16] Benjamin C. Pierce. Types and Programming Languages. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142 ISBN:0-262-16209-1

[17] Tipy v jazykah programmirovanija / Perevod s angl. M.: Izdatel'stvo "Ljambda press": "Dobrosvet", 2011 656+xxiv s. ISBN 978-5-9902824-1-4, 978-5-7913-0082-9

[18] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. Proc. ACM Program. Lang. 3, POPL, Article 16 (January 2019), 32 pages. DOI:https://doi.org/10.1145/3290329

[19] Karla Ramírez Pulido, Jorge Luis Ortega-Arjona, Lourdes del Carmen González Huesca. 2020. Gradual Typing Using Union Typing With Records. Electronic Notes in Theoretical Computer Science, Volume 354, Pages 171-186, ISSN:1571-0661, https://doi.org/10.1016/j.entcs.2020.10.013.

[20] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic type inference for gradual Hindley–Milner typing. Proc. ACM Program. Lang. 3, POPL, Article 18 (January 2019), 29 pages. DOI:https://doi.org/10.1145/3290331

[21] V. Kryshtapovich. " GraduLama" source code https://github.com/kry127/Lama/tree/gradulama Request timestamp: 27/03/2021 07:35

$$\frac{}{\Gamma, \Delta \vdash \texttt{Const}(i) \mapsto \Delta \vdash \texttt{Const}(i) : \texttt{TConst}} \ [\texttt{InferConst}]$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{String}(s) \mapsto \Delta \vdash \texttt{String}(s) : \texttt{TString}} \ [\texttt{InferString}]$$

$$\frac{\Delta_0 := \Delta \qquad \Gamma, \Delta_{i-1} \vdash e_i \mapsto \Delta_i \vdash e_i' : \tau_i}{\Gamma, \Delta \vdash \texttt{Arr}(e_1 \dots e_n) \mapsto \Delta_n \vdash \texttt{Arr}(\overline{e_i'}) : \texttt{TArr}(\texttt{TUnion}(\overline{\tau_i}))} \ [\texttt{InferArr}]$$

$$\frac{\Delta_0 := \Delta \qquad \Gamma, \Delta_{i-1} \vdash e_i \mapsto \Delta_i \vdash e_i' : \tau_i}{\Gamma, \Delta \vdash \texttt{Sexp}(s, e_1 \dots e_n) \mapsto \Delta_n \vdash \texttt{Sexp}(s, \overline{e_i'}) : \texttt{TSexp}(s, \overline{\tau_i})} \ [\texttt{InferSexp}]$$

$$\frac{\Gamma(s) = \tau}{\Gamma, \Delta \vdash \texttt{Var}(s) \mapsto \Delta \vdash \texttt{Var}(s) : \tau} \ [\texttt{InferVar}]$$

$$\frac{\Gamma(s) = \tau}{\Gamma, \Delta \vdash \texttt{Ref}(s) \mapsto \Delta \vdash \texttt{Ref}(s) : \texttt{TRef}(\tau)} \ [\texttt{InferRef}]$$

$$\frac{\Gamma, \Delta \vdash e_1 \mapsto \Delta_1 \vdash e_1' : \tau_1 \qquad \tau_1 \sim \texttt{TConst} \qquad \Gamma, \Delta_1 \vdash e_2 \mapsto \Delta_2 \vdash e_2' : \tau_2 \qquad \tau_2 \sim \texttt{TConst}}{\Gamma, \Delta \vdash \texttt{Binop}(s, e_1, e_2) \mapsto \Delta_2 \vdash \texttt{Binop}(s, \texttt{Cast}(e_1', \texttt{TConst}), \texttt{Cast}(e_2', \texttt{TConst})) : \texttt{TConst}} \ [\texttt{InferBinop}]$$

$$\frac{\Gamma, \Delta \vdash e_1 \mapsto \Delta_1 \vdash e_1' : \texttt{TAny} \qquad \Gamma, \Delta_1 \vdash e_2 \mapsto \Delta_2 \vdash e_2' : \tau_2 \qquad \tau_2 \sim \texttt{TConst}}{\Gamma, \Delta \vdash \texttt{Elem}(e_1, e_2) \mapsto \Delta_2 \vdash \texttt{Elem}(e_1', \texttt{Cast}(e_2', \texttt{TConst})) : \texttt{TAny}} \ [\texttt{InferElem}]$$

$$\frac{\Gamma, \Delta \vdash e_1 \mapsto \Delta_1 \vdash e_1' : \texttt{TStr} \qquad \Gamma, \Delta_1 \vdash e_2 \mapsto \Delta_2 \vdash e_2' : \tau_2 \qquad \tau_2 \sim \texttt{TConst}}{\Gamma, \Delta \vdash \texttt{Elem}(e_1, e_2) \mapsto \Delta_2 \vdash \texttt{Elem}(e_1', \texttt{Cast}(e_2', \texttt{TConst})) : \texttt{TConst}} \ [\texttt{InferElemOfStr}]$$

$$\frac{\Gamma, \Delta \vdash e_1 \mapsto \Delta_1 \vdash e_1' : \texttt{TArr}(\tau_1) \qquad \Gamma, \Delta_1 \vdash e_2 \mapsto \Delta_2 \vdash e_2' : \tau_2 \qquad \tau_2 \sim \texttt{TConst}}{\Gamma, \Delta \vdash \texttt{Elem}(e_1, e_2) \mapsto \Delta_2 \vdash \texttt{Elem}(e_1', \texttt{Cast}(e_2', \texttt{TConst})) : \tau_1} \ [\texttt{InferElemOfArr}]$$

$$\frac{\Gamma, \Delta \vdash e \mapsto \Delta_1 \vdash e' : \tau \qquad \tau \in \langle \texttt{TAny}, \texttt{TArr}, \texttt{TString}, \texttt{TSexp} \rangle}{\Gamma, \Delta \vdash \texttt{Length}(e) \mapsto \Delta_1 \vdash \texttt{Length}(e') : \texttt{TConst}} \ [\texttt{InferLength}]$$

$$\frac{\Gamma, \Delta \vdash e \mapsto \Delta_1 \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{StringVal}(e) \mapsto \Delta_1 \vdash \texttt{StringVal}(e') : \texttt{TString}} \ [\texttt{InferStringVal}]$$

$$\frac{\Gamma, \Delta \vdash f \mapsto \Delta_0 \vdash f' : \texttt{TAny} \qquad \Gamma, \Delta_{i-1} \vdash e_i \mapsto \Delta_i, e_i' : \tau_i}{\Gamma, \Delta \vdash \texttt{Call}(f, e_1 \dots e_n) \mapsto \Delta_n \vdash \texttt{Cast}(\texttt{Call}(f', \overline{e_i'}), \texttt{TAny}) : \texttt{TAny}} \ [\texttt{InferCall}]$$

$$\frac{\Gamma, \Delta \vdash f \mapsto \Delta_0 \vdash f' : \texttt{TLambda}(\gamma_1 \dots \gamma_m, \tau) \qquad m = n \qquad \Gamma, \Delta_{i-1} \vdash e_i \mapsto \Delta_i, e_i' : \tau_i \qquad \tau_i \sim \gamma_i}{\Gamma, \Delta \vdash \texttt{Call}(f, e_1 \dots e_n) \mapsto \Delta_n \vdash \texttt{Cast}(\texttt{Call}(f', \overline{\texttt{Cast}(e_i', \gamma_i)}), \tau) : \tau} \ [\texttt{InferCall2}]$$

$$\frac{\Gamma, \Delta \vdash f \mapsto \Delta_0 \vdash f' : \texttt{TUnion}(\gamma_1 \dots \gamma_m) \qquad \Gamma \cup \{x : \gamma_i\}, \Delta_{i-1} \vdash \texttt{Call}(x, \overline{e_j}^{i-1}) \mapsto \Delta_i, \texttt{Cast}(\texttt{Call}(x', \overline{e_j}^{i}), \tau_i) : \tau_i}{\Gamma, \Delta \vdash \texttt{Call}(f, e_1^0 \dots e_n^0) \mapsto \Delta_m \vdash \texttt{Cast}(\texttt{Call}(f', \overline{e_j}^{m}), \texttt{TUnion}(\overline{\tau_i})) : \texttt{TUnion}(\overline{\tau_i})} \ [\texttt{InferCall3}]$$

$$\frac{\Gamma, \Delta \vdash r \mapsto \Delta_1 \vdash r' : \texttt{TAny} \qquad \Gamma, \Delta_1 \vdash e \mapsto \Delta_2 \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{Assign}(r, e) \mapsto \Delta_1 \vdash \texttt{Assign}(r', e') : \texttt{TAny}} \ [\texttt{InferAssign1}]$$

$$\frac{\Gamma, \Delta \vdash r \mapsto \Delta_1 \vdash r' : \texttt{TRef}(\rho) \qquad \Gamma, \Delta_1 \vdash e \mapsto \Delta_2 \vdash e' : \tau \qquad \tau \sim \rho}{\Gamma, \Delta \vdash \texttt{Assign}(r, e) \mapsto \Delta_1 \vdash \texttt{Assign}(r', e') : \rho} \ [\texttt{InferAssign2}]$$

Fig. 9. Rules of type inferrence and cast insertion for expressions.

$$\frac{\Gamma, \Delta \vdash e_1 \mapsto \Delta_1 \vdash e_1' : \sigma \qquad \Gamma, \Delta_1 \vdash e_2 \mapsto \Delta_2 \vdash e_2' : \tau}{\Gamma, \Delta \vdash \texttt{Seq}(e_1, e_2) \mapsto \Delta_2 \vdash \texttt{Seq}(e_1', e_2') : \tau} \ \texttt{[InferSeq]}$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{Skip} \mapsto \Delta \vdash \texttt{Skip} : \texttt{TVoid}} \ \texttt{[InferSkip]}$$

$$\frac{\Gamma, \Delta \vdash c \mapsto \Delta_1 \vdash c' : \sigma \sim \texttt{TConst} \qquad \Gamma, \Delta_1 \vdash e_t \mapsto \Delta_2 \vdash e_t' : \tau_t \qquad \Gamma, \Delta_2 \vdash e_f \mapsto \Delta_3 \vdash e_f' : \tau_f}{\Gamma, \Delta \vdash \texttt{If}(c, e_t, e_f) \mapsto \Delta_3 \vdash \texttt{If}(\texttt{Cast}(c', \texttt{TConst}), e_t', e_f') : \texttt{TUnion}(\tau_t, \tau_f)} \ \texttt{[InferIf]}$$

$$\frac{\Gamma, \Delta \vdash c \mapsto \Delta_1 \vdash c' : \sigma \sim \texttt{TConst} \qquad \Gamma, \Delta_1 \vdash e \mapsto \Delta_2 \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{While}(c, e) \mapsto \Delta_2 \vdash \texttt{While}(\texttt{Cast}(c', \texttt{TConst}), e') : \texttt{TVoid}} \ \texttt{[InferWhile]}$$

$$\frac{\Gamma, \Delta \vdash c \mapsto \Delta_1 \vdash c' : \sigma \sim \texttt{TConst} \qquad \Gamma, \Delta_1 \vdash e \mapsto \Delta_2 \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{Repeat}(e, c) \mapsto \Delta_2 \vdash \texttt{Repeat}(e', \texttt{Cast}(c', \texttt{TConst})) : \texttt{TVoid}} \ \texttt{[InferRepeat]}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash m \mapsto \Delta_0 \vdash m' : \omega \\ \Gamma \cup \tau_\Gamma(p_i), \Delta_{i-1} \vdash e_i \mapsto \Delta_i \vdash e_i' : \tau_i \qquad \tau_r(p_i) \sim \omega \qquad \omega \sim \texttt{TUnion}(\overline{\tau_l(p_i)}) \qquad \forall j < i \ . \ \tau_r(p_i) \nsim \tau_l(p_j)\end{array}}{\Gamma, \Delta \vdash \texttt{Case}(m, (p_1, e_1) \ldots (p_n, e_n)) \mapsto \Delta_n \vdash \texttt{Case}(m', (p_1, e_1') \ldots (p_n, e_n')) : \texttt{TUnion}(\overline{\tau_i})} \ \texttt{[InferCase]}$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{Return} \mapsto \Delta \cup \{\texttt{TVoid}\} \vdash \texttt{Return} : \texttt{TVoid}} \ \texttt{[InferReturn1]}$$

$$\frac{\Gamma, \Delta \vdash e \mapsto \Delta' \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{Return}(e) \mapsto \Delta' \cup \{\tau\} \vdash \texttt{Return}(e') : \texttt{TVoid}} \ \texttt{[InferReturn2]}$$

$$\frac{\Gamma, \Delta \vdash e \mapsto \Delta' \vdash e' : \tau}{\Gamma, \Delta \vdash \texttt{Ignore}(e) \mapsto \Delta' \vdash \texttt{Ignore}(e') : \texttt{TVoid}} \ \texttt{[InferIgnore]}$$

$$\frac{\Delta_0 := \Delta \qquad \Gamma \cup \{\overline{(s_i, \tau_i)}_{k=1}^{i-1}\}, \Delta_{i-1} \vdash e_i \mapsto \Delta_i \vdash e_i' : \tau_i \qquad \Gamma \cup \{\overline{(s_i, \tau_i)}_{k=1}^{n}\}, \Delta_n \vdash e \mapsto \Delta' \vdash e' : \omega}{\Gamma, \Delta \vdash \texttt{Scope}(\overline{(s_i, e_i)}_{i=1}^{n}, e) \mapsto \Delta' \vdash \texttt{Scope}(\overline{(s_i, e_i')}_{i=1}^{n}, e') : \omega} \ \texttt{[InferScope]}$$

$$\frac{\Gamma \cup \{\overline{(s_i, \sigma_i)}_{i=1}^{n}\}, \varnothing \vdash e \mapsto \Delta', e' : \delta \qquad \texttt{TUnion}(\Delta' \cup \{\delta\}) \sim \tau}{\Gamma, \Delta \vdash \texttt{Lambda}(\overline{(s_i, \sigma_i)}_{i=1}^{n}, e, \tau) \mapsto \Delta, \texttt{Lambda}(\overline{(s_i, \sigma_i)}_{i=1}^{n}, e', \tau) : \texttt{TLambda}(\overline{\sigma_i}, \tau)} \ \texttt{[InferLambda]}$$

Fig. 10. Rules of type inferrence and cast insertion for expressions (part 2).