Localized Lama Gradual Typing

V.S. Kryshatapovich, ORCID: 0000-0002-3941-6201 <kry127@yandex.ru>
ITMO University,
Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia

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.

Keywords: programming languages; gradual typing; type safety; cast calculus

For citation: Kryshatapovich V.S. Localized Lama Gradual Typing. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 61-76. DOI: 10.15514/ISPRAS-2021-33(3)-5.

Локализованное применение частичной типизации

В.С. Крыштапович, ORCID: 0000-0002-3941-6201 <kry127@yandex.ru>
Университет ИТМО
197101, г. Санкт-Петербург, Кронверкский проспект, д.49

Аннотация. Частичная типизация — это современный подход для сочетания преимуществ статической и динамической типизации. Но несмотря на то, что научные исследования направлены на корректировку систем типов, многие языки намеренно делают систему некорректной для ускорения производительности. Данная работа посвящена реализации диалекта языка Лама, который поддерживает частичную типизацию для явно указанных участков кода. Целью реализации является сочетание двух подходов: обеспечение типобезопасности в одних участках кода и производительность языка в других участках кода. Статья раскрывает дetailи реализации и свойства полученной системы, а также рассматривает способы улучшения полноты и корректности полученной системы.

Ключевые слова: языки программирования; частичная типизация; системы типов; исчисление преобразований

Для цитирования: Крыштапович В.С. Локализованное применение частичной типизации. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 61-76 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(3)–5.

1. 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 is 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 us imagine typical Python code, and most probably it would be some untyped piece of code. Surprisingly or not, only 3.9% 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 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.

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 programs at all)
2) If program typechecks, it can produce only dynamic type error or cast errors. No errors involving immutability 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 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 ×, likely rendering them unusable for their actual purpose. To increase performance many of them reduce number 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.

2\*x\*y

\[\text{2\*x\*y}\]

If \(x\) and \(y\) have known at compile time types, then type of the functions can be inferred: inner function has type \(\text{Int} \to \text{Int}\), and outer function has type \(\text{Int} \to \text{Int} \to \text{Int}\).

Moreover, Lama nowadays supports operations only with integer constants (\(\text{Int}\)). If we take a closer look to the untyped example, it can be inferred that \(x\) should have type \(\text{Int}\), \(y\) should have type \(\text{Int}\), because they are used in expression like \(2 \times x \times y\), 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 compatibility. So we can bring up type inference features with awareness on backward compatibility.

Another example of typing Lama programs is pattern matching

\[
\begin{cases}
\text{case } a \text{ of } \\
A(0) \rightarrow "1" \\
A(x) \rightarrow "2" \\
\text{esac}
\end{cases}
\]

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, number of values is not bounded. Two S-expression labels are considered equal in Lama if their five first letters are the same, so \(\text{Branch} (\text{Leaf}, \text{Leaf}, 3)\) and \(\text{Branch} (\text{Leaf}, \text{Leaf}, 3)\) are equal S-expressions. By the way, \(\text{Leaf}\) is nested S-expression with zero values in it, so brackets are optional for zero-arity S-expressions.

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

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

\[
\text{local foo = fun } (x :: A(\text{Int})) \{ \\
\text{case } x \text{ of } \\
\text{A(0) } \rightarrow "1" \\
\text{A(x, y) } \rightarrow "3" \text{ anything } \\
\text{esac} \};
\]

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 \(x :: A(\text{Int})\).

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

\[
\text{public fun id2 (Abc (x, y)) :: } ? \{
\}
\]

Kryshapovich V.S. Localized Lama Gradual Typing. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 61-76.
At first glance it is unclear, where is the problem, because `dynt("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:

```plaintext
def fun intStringer(x :: Int) {
  x.string
}
```

Of course, if we try to reduce `dynt("input")` we get `"input" + 1`, and then we will 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 is why we should check function arguments wrapping them with appropriate dynamic casts. So, if follow blame ideology in both implementations `dynt("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 number 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 section we will define syntax of gradual types and their semantics.

### 3. 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 will 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 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 section “Concrete syntax and semantics” of specification [10].

The definition of type annotations `typeExpression` is presented on the fig. 2. It semantic (see 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 `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.

### 4. 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.

- `Int` corresponds to integers `t = TInteger;`
- `Str` corresponds to strings `t = TString;`
- `Void` corresponds to empty set of values `t = 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.

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

Fig. 2. Typing expression syntax

Fig. 3. Typing expression semantics

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 `t = TInteger;`
- `Str` corresponds to strings `t = TString;`
- `Void` corresponds to empty set of values `t = 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.
• $v$ 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.

Let us denote set of variables by $V$, which represented by OCaml string $s$, and set of types $T$. We should think about $T$ wider, that types induced by type constructors of fig. 3. In other words, some type $\tau \in T$ 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 $\text{Cast}$ constructors, i.e. pure untyped Lama expression. Notice, that expression can also contain patterns $p$ due to pattern matching in $\text{Case}$ expression. 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 trying to statically typecheck expression. If we succeed to acquire static type of program represented as whole expression, we can conclude that there is 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 $\tau(p) \in T$ for lower bound inferred type for pattern and $\tau(p) \in T$ 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:

• $\tau_p$ is representing type that covers all possible values captured by pattern (upper bound);
• $\tau_i$ is representing type that is covered by all possible values captured by pattern (lower bound).

But the good news is that no type intersections $\text{TIntersection}$ or type subtractions $\text{TSubstraction}$ are coming – we try to avoid them when building type system for Lama. Now we can make an analogy of $\sim$ relation for expression $e$ and type $\tau$. 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 types is crucial for analyzing case expressions. Let’s denote $\text{Cast}(\tau_{\text{lower}}, \tau_{\text{upper}})$ for lower bound inferred type for pattern and $\text{Cast}(\tau_{\text{lower}}, \tau_{\text{upper}})$ for upper bound inferred type for pattern. 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 trying to statically typecheck expression. If we succeed to acquire static type of program represented as whole expression, we can conclude that there is 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 $\tau(p) \in T$ for lower bound inferred type for pattern and $\tau(p) \in T$ 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:

• $\tau_p$ is representing type that covers all possible values captured by pattern (upper bound);
• $\tau_i$ is representing type that is covered by all possible values captured by pattern (lower bound).

For example, value $\text{Suc}(1)$ has type $\text{TSexp}("\text{Suc}", \text{TConst})$, but this value alone covers almost nothing, so $\text{TVoid} \sqsubseteq \{\text{Suc}(1)\}$ but this value alone covers almost nothing, so $\text{TVoid} \sqsubseteq \{\text{Suc}(1)\} \sqsubseteq \text{TSexp}("\text{Suc}", \text{TConst})$. Now we are ready to describe our main part of algorithm: type inference and cast insertion for Lama expressions. We will use such notation: $e \rightarrow e': \tau$. That means that expression $e$ has type $\tau$, and cast insertion into that expression produces expression $e'$, which has the same type $\tau$. In addition, we have two types of contexts: $\Gamma : \Psi \Rightarrow \tau$ for typing context of variables (which assigns types to variable typenames) and set of types $\Delta \in T$ 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:

$\Gamma, \Delta \vdash e \Rightarrow e': \tau$. 

Fig. 4. Lama expression class

Fig. 5. Lama value class

Fig. 6. Lama pattern class

Fig. 7. Rules of conformance to the other type
Крыштапович В.С. Локализованное применение частичной типизации. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 61-76.

69

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

Fig. 9 and 10 presenting all set of rules for type inference of Lama expression with \( e \Rightarrow e \Rightarrow e \Rightarrow e \Rightarrow e \) notation used. Let us highlight some features about presented algorithm.

The set of return types for expression \( e \) is initialized with \( \emptyset \). Note, that initial context \( \Gamma \) maps every variable occurrence to type \( T_{\text{Any}} \). 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.

Notation \( \phi \in \langle \chi \rangle \) in rule \([\text{InferLength}]\) means that \( \phi \)'s top level constructor should be one of the listed in angle brackets.

In rule \([\text{InferCase}]\) cast to \( T_{\text{Any}} \) is optional. It is used in inference rules to be consistent with \([\text{InferCall3}]\) rule which process call of the union type object.

Many of the rules can be simplified by removing \( \Delta \) because they do not change it, such as \([\text{InferLambda}],[\text{InferReturn1}],[\text{InferReturn2}]\), etcetera. That is because they recompute \( \Delta \) for expressions that never change \( \Delta \) in correct Lama expressions. There are a few places where \( \Delta \) is useful: it is \([\text{InferLambda}],[\text{InferReturn1}],[\text{InferReturn2}]\) rules. Notice, that we are 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 the user, the inferred type for variable will be used implicitly.

Also notice rules in \([\text{InferCase}]\). First, we collect return types from the branches while dragging \( \Delta \) through the computation pipeline. The second one, look at notation \( \Gamma \cup \tau : \{x: \tau(p_i)\} \) – it fulfills typing context with mapping of \( P_{\text{Named}} \) named pattern to its types. The \( \tau_F \) can be defined via \( \tau_r \) as follows:

\[
\tau_r(p_i) \cup \{x: \tau_r(p_i)\} = P_{\text{Named}}(s, p_r) = \bigcup_{n} \tau_r(p_i) = P_{\text{Exp}}(s, p_r, \ldots, p_n)
\]

otherwise

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

\[
\tau_r(p_i) \sim \tau_r(p_i) \Rightarrow \tau(p_i) \subset \tau_r(p_i) \subset \tau_r(p_i) \subset (p_i).
\]
Kryshtapovich V.S. Localized Lama Gradual Typing. *Tradey ISP RAN/Proc. ISP RAS*, vol. 33, issue 3, 2021, pp. 61-76.

In other words, when expression holds, it is certain that pattern \( \cdot \) was covered by more recent cover \( \cdot \). 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 [17] or [18] for example of polymorphic type system that handles that.

The most complex is \([\text{InferScope}]\) rule. It is intentionally simplified, because its implementations is more subtle. Here it simply overwrites variable or function definition and updates context. 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:

```plaintext
f g (D (0)) \; \text{-- ok}
\{
  f \cdot \text{Int} \rightarrow \text{Str};
  g \cdot \text{Int} \rightarrow \text{Int};
  f (g (0)) \; \text{-- ok}
  f (g (D (0))) \; \text{-- error}
\}
```

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

5. Cast Performance Analyzing

It is obvious that rules presented at fig.9 introduce new kind of expression \( \text{Cast} \left( \cdot, \cdot \right) \). It’s runtime semantics is simple: when expression \( \cdot \) evaluates to value \( \cdot \), we should check that value \( \cdot \) corresponds to type \( \cdot \). If \( \cdot \) conforms to \( \cdot \), the result of evaluation of \( \text{Cast} \left( \cdot, \cdot \right) \) is \( \cdot \), otherwise cast error \( \bot \) produced as the result.

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

```plaintext
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 type checking procedure:

- \( \#\text{NoTypecheck} \) – drops AST from type checking at all;
- \( \#\text{StaticTypecheck} \) – disable cast insertion into AST, but static checks are still enabled;
- \( \#\text{GradualTyping} \) – enable cast insertion into AST.

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

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

```plaintext
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 2 we have noticed that + operator coerces both its arguments to \textit{Const} at runtime, so appropriate casts to \textit{TConst} types from unknown type are inserted. Hence, this code is modeling 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 1,100s |
| 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 | 0m14,188s |
| 34      | 0m15,981s | 0m24,799s |
| 35      | 0m26,933s | 0m43,855s |
| 36      | 0m42,326s | 1m 7,766s |
| 37      | 1m32,161s | 1m49,319s |
| 38      | 1m53,534s | 3m 0,748s |
| 39      | 3m18,464s | 4m54,461s |
| 40      | 5m17,664s | 7m54,811s |

The average of slowdown: $$s_d = \frac{m}{\text{typ}}$$ from the point of actual slowdown registered $$n = 21$$ is: $$\frac{1}{20} \times 40 \times 1.10 \times 1.10 \times 1.45 \approx 1.45.$$ As we can see, section of code with active gradual typing runtime type checking exhibit almost × 1.5 slowdown. Thus, we have reproduced the result of an article [2] but in the case Lama semantics using this artificially small example.

### 6. 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 [17] and [19].

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 guarantees, 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 guarantees is the most important at applied piece of code. To sum up, programmer maintains 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. Moreover, 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” [20].

### References

[1] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In Proc. of the Seventh Workshop on Scheme and Functional Programming, 2006, pp. 81–92.

[2] Camron Moy, Phuc C. Nguyen et al. Corpse reviver: sound and efficient gradual typing via contract verification. Proceedings of the ACM on Programming Languages, vol. 5, issue POPL, 2021, Article 53, 28 p.

[3] D. Boultychev. JetBrains-Research/Lama source code. Available at https://github.com/JetBrains-Research/Lama, accessed 27/03/2021.

[4] Ingkarat Rak-amnouykit, Daniel McCrevan et al. Python 3 types in the wild: a tale of two type systems. In Proc of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2020), 2020, pp. 57-70.

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

[6] Guido van Rossum, Jukka Lehtosalo, Ł. Łukasz, J. St-Amour. PEP 484 – Type Hints. Available at https://www.python.org/dev/peps/pep-0484/, accessed 27/03/2021.

[7] Jukka Lehtosalo et al. MyPy: Optional Static Typing for Python. Available at https://github.com/python/mypy, accessed 27/03/2021.

[8] Pytype: A static type analyzer for Python code. Available at https://github.com/google/pytype, accessed 27/03/2021.

[9] Sam Tobin-Hochstadt, Vincent St-Amour et al. The Typed Racket Guide. Available at https://docs.racket-lang.org/tsguide/index.html, accessed 27/03/2021.

[10] D. Boultychev. Lama language specification v. 1.10. Available at https://github.com/JetBrains-Research/Lama/blob/1.10/lama-spec.pdf, accessed 27/03/2021.

[11] R. Rivest. S-Expressions., 4/05/1997. Available at http://people.csail.mit.edu/rivest/Sexp.txt, accessed 29/03/2021.

[12] Asma Ahmed, Dustin Jannner. Theorems for free for free: parametricity, with and without types. Proceedings of the ACM on Programming Languages, vol. 1, issue ICFP, 2017, Article 39, 28 p.

[13] Jack Williams, J. Garrett Morris, and Philip Wadler. The root cause of blame: contracts for intersection and union types. Proceedings of the ACM on Programming Languages, vol. 2, issue OOPSLA, 2018, Article 134, 29 pages.
[14] P. Wadler. A Complement to Blame. In Proc. of the 1st Summit on Advances in Programming Languages (SNAPL 2015), 2015, pp. 309-320.

[15] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, vol. 74, no. 2. 1953, pp. 358- 366.

[16] Benjamin C. Pierce. Types and Programming Languages. The MIT Press, 2002, 648 p.

[17] Giuseppe Castagna, Victor Lanvin et al. 2019: Gradual typing: a new perspective. Proceedings of the ACM on Programming Languages, vol. 3, issue POPL, 2019, Article 16, 32 p.

[18] Karla Ramírez Pulido, Jorge Luis Ortega-Arjona et al. Gradual Typing Using Union Typing with Records. Electronic Notes in Theoretical Computer Science, vol.354, 2020, pp. 171-186.

[19] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic type inference for gradual Hindley–Milner typing. Proceedings of the ACM on Programming Languages, vol. 3, issue POPL, 2019, Article 18, 29 pp.

[20] V. Kryshtapovich. GraduLama source code Available at https://github.com/kry127/Lama/tree/gradulama, accessed 27/03/2021.

Информация об авторах / Information about authors

Виктор Сергеевич КРЫШТАПОВИЧ, студент магистратуры второго курса. Научные интересы: системы типов, базы данных.

Viktor Sergeevich KRYSHTAPOVICH, second year master's student. Research interests: type systems, databases.