Implementing System T in Haskell

Jingren Wang

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

Dec 12th, 2021

Abstract

Computers are general-purpose computing devices, and programming languages enable programmers to talk to them. In programming languages, we study the meta-theory of programming languages, such as, their expressivity, robustness, and efficiency.

In this project, we perform a case study in the design and implementation of programming languages. We design a toy language based on System T, and implement it using Haskell. Our language has natural numbers, booleans, higher-order functions and recursion. We design a bidirectional type system and an operational semantics for it. We evaluate our language implementation by verifying important properties of the type system and operational semantics. We write several example programs in our language and run them, comparing the performance against their native implementation.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Jingren Wang Signature: Jingren Wang

All source code for this project can be found at: `https://github.com/wjrforcyber/SystemT`.

Acknowledgements

Firstly, I want to thank my supervisor Vikraman Choudhury. This project literally opened my eyes in programming language. And it is the first time I build a language in Haskell. It changes the way I think about programming. My supervisor was very patient to me during the past few months, and we met in person every week, without his help I cannot reach this far.

Also, I want to say thanks to my parents, although they are seven hours ahead of me, they are always patient and try to spend time talking to me when I'm under pressure.

Due to this Covid-19, this is a very hard but special year not only for me, but for all of us. So I also want to say thanks to my friend Ella Kohn, thanks for the study time we spent together, nothing could compare to those precious time. And also, thanks to all those friends who showed up during my solo hiking journey at amazing Balloch and helpful and friendly strangers at University of Glasgow.

Contents

1 Introduction **5**

1.1 Design overview . . . 5

1.2 Outline . . . 6

2 Background **7**

2.1 Typing . . . 7

2.2 Bidirectional type checking . . . 8

2.3 Operational Semantics . . . 8

2.4 Tasty . . . 9

2.5 Criterion . . . 10

2.6 Prettyprinter . . . 10

2.7 GADTs . . . 10

2.8 System T . . . 10

3 L3: A language with numbers and booleans **12**

3.1 Grammar . . . 12

3.1.1 Syntax of L3 . . . 12

3.1.2 Type checking . . . 12

3.1.3 Semantics . . . 13

3.1.4 Implementation . . . 13

3.1.5 Testing . . . 16

3.2 Intrinsic L3 . . . 16

4 L6: A language with many types 18

4.1 Grammar . . . 18

4.1.1 Syntax of L6 . . . 18

4.1.2 Type checking L6 . . . 19

4.1.3 Symantics L6 . . . 20

4.1.4 Implementation . . . 20

4.1.5 Function Testing . . . 24

5 Evaluation 28

5.1 Tests . . . 28

5.2 Benchmark . . . 29

6 Conclusion 31

A Benchmark result 34

Chapter 1

Introduction

In the study of programming languages, various techniques are used in implementing and formalizing meta-theory of programming languages [Har16]; type systems and operational semantics are the typical forms which provides an objective foundation for such analyses [Har05]. In this project, Haskell will be used as a host language to design a programming language similar to System T [Göd58], with higher-order functions and recursion, and a bidirectional type-checker and an evaluator to ensure its coherence and consistency with expectations.

A programming language provides notation or syntax for developers to write programs [Aab96]. There are lots of different types of programming languages, such as Python, C, Haskell, Rust, Ocaml, etc. Nowadays, functional language is becoming more and more popular. The main reason is developers like neat, friendly, easier-to-debug ways to construct programs, and functional programming languages are powerful in doing recursive programming. Haskell [Hal+92], as one of the leading fast-growing programming languages, has been selected as the host programming language for our implementation.

The semantics are used to describe how a system works, that is what behaviour does the system has. In our project, system specifically based on system T, generally speaking, semantics are the system's analysis [Gog75]. Semantics are also needs to be very rigorous because they avoid ambiguity and clear complicity in defining documents as well as form basis for implementation, analysis and verification [NN07].

1.1 Design overview

We start by designing a simple language L1 with a single type – natural numbers with primitive operations for addition and multiplication.

L6 is the final language based on system T which is gradually developed step by step from L1. To reach L6, the new type constructors are added one after another. The overview of the project is shown in figure 1.1.

In L1 and L2, natural numbers are used from the host language, that is, we define natural numbers in Haskell, and use them for the expressions in our language. In L3, we have first-class expressions

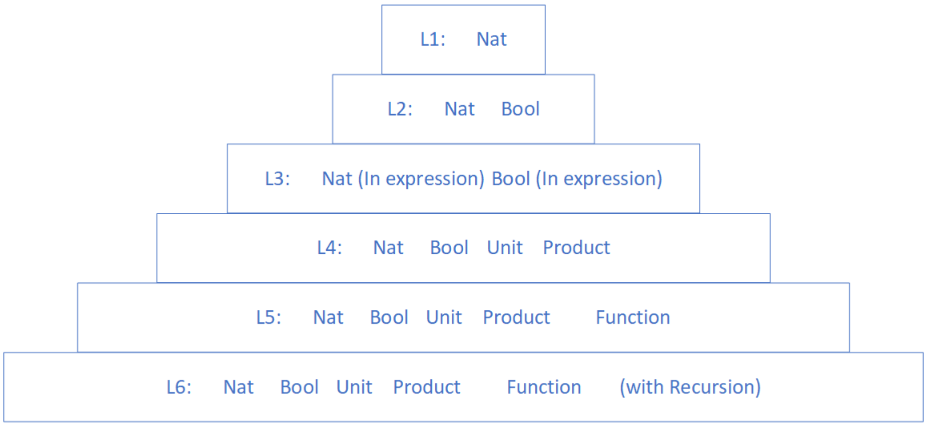

L1: Nat
L2: Nat Bool
L3: Nat (In expression) Bool (In expression)
L4: Nat Bool Unit Product
L5: Nat Bool Unit Product Function
L6: Nat Bool Unit Product Function (with Recursion)

Figure 1.1: Overview of L1 to L6 type structure

for constructing natural numbers, that is, we have an expression form for zero, and an expression form for successor. In L4, Unit and Product types are added to L3, so if A is a type and B is a type, then A × B is a type, which means we now have infinitely many types which can be formed recursively, such as, Nat × Nat, Nat × Bool, (Nat × Bool) × Bool, (Nat × (Nat × Nat)) × ((Nat × Bool) × (Bool × Bool)) etc. We add higher-order functions, variables, lambda abstractions, and application in L5, means now we could now do functions. In L6, recursion is added based on L5 to enable recursion using natural numbers, which makes our language much more powerful. Recursive expression in L6 take three parameters, first expression and second expression based on third expression, second expression always applies to the third expression unless the third expression reduced to 0, which is the time to return the first expression.

1.2 Outline

- In Chapter 2, we give a background of the method, library, framwork we used in our design and experiment, which includes typing, testing framework Tasty, benchmarking framework Criterion, friendly-format building tool Prettyprinter.
- In Chapter 3, we describe the language L3, its syntax, grammar, type system, and Big-step semantics. We also describe and intrinsic and extrinsic implementation of it in Haskell.
- In Chapter 4, we describe the final language implementation L6, in a similar structure as chapter 3, but with products, function, iteration, which is important in constructing functions. And both Big-step and Small-step semantics are in L6. Reduction rules and several testing functions are implemented in L6, too.
- In Chapter 5, we demonstrate the evaluation result from Tasty and Criterion to give explanation on the correctness and performance of our language.
- In Chapter 6, we conclude by discussing the potential future work on the language.

Chapter 2

Background

2.1 Typing

We want to make sure the programmer doesn't write bad programs, that is, we don't want programs to get stuck at runtime. A type system weeds out invalid programs, that is, we assign types to programs, and only allows the programmer to execute programs that are well-typed using our typing rules. To do this, we design a type system, and implement a type-checker.

There are many ways of designing type systems. In this work, we use bi-directional typing, which consists of a type-checker and type-synthesiser (or type-inferencer). A type system is described using judgements and inference rules.

A judgement is a relation, connecting expressions to types. It is a statement, sometimes states one or more syntax trees have some properties or stand in some relation [Har16]. For example,

$$e : \tau$$

means e has the type τ.

$$e \Downarrow v$$

means e reduce to the value v.
In bidirectional type checking, we split $e : \tau$ into check:

$$e \Leftarrow \tau$$

and infer:

$$e \Rightarrow \tau$$

As for type inference, an inference rule lets us use our hypotheses to infer new information. In this case, for example, knowing the type of a sub-expression, allows us to infer the type of the whole expression. Inference always needs judgments, collection in premises sufficient for the conclusion.

If premise is empty, the conclusion stands, then the conclusion stands unconditionally [Har16]. For example,

$$\frac{e \Leftarrow \mathsf{Nat}}{\mathsf{Suc}(e) \Rightarrow \mathsf{Nat}}$$

To infer $\mathsf{Suc}(e)$ has type Nat, the expression e must be checked to have a Nat type too.

2.2 Bidirectional type checking

In bidirectional type systems, there are two kinds of judgements, one for type-checking, and one for type-inference. Bidirectional type checking, as its name shows, involves types, bidirectional checking, which means check and synthesis of types(in this project we use infer instead of synthesis). So firstly, a completed type system should be built, our project, which is different from the original system T, Boolean, Unit, Product, and Function types are added to the system and then rules are written to make sure type checker and infer works as we expected. To make a type system bidirectional, a bottom-up type information derivation rule construction is much easier to think[Pfe04]. Detailed construction of the bidirectional type checking will be shown in L3 and L6 in this thesis.
Bidirectional type checking is used into the implementation from L1 to L6, we also used notation in bidirectional type checking rules in lambda expression, and then it could help type system syntax-directed[Chr19].

Bidirectional checking rules can be evaluated by Lecture Notes on Frank Pfenning's Bidirectional Type Checking[Pfe04], there is a four-steps rule:
1. Assume each input constituent of the conclusion is known.
2. Show that each input constituent of the premise is known, and each output constituent of the premise is still free (unknown).
3. Assume that each output constituent of the premise is known.
4. Show that each output constituent of the conclusion is known.
Whenever a rule has been constructed, this four-steps rule can be applied to check if it is constructed in a correct way.
For a general understanding of bidirectional rules, infer is always much stronger than check, it needs more bounds and premises for conclusion to stand.

2.3 Operational Semantics

An operational semantics for a language explains how to execute programs written in that language. There are many styles of doing operational semantics. In this work, we design and implement small-step and big-step operational semantics.

An operational semantics is presented as a reduction system, that is, a binary relation on the set expressions of the language.

The two ways of doing operational semantics have been used in L6, Big-step and Small-step, each of them has strength and weakness. We can use them both in our design from L1 to L6. Big-step is mostly used in an interpreter in simple language, which makes it efficiently evaluates.

Small-step is used to step through the example program in complicated language, which will help evaluate each step of the example program to test the language's robustness and soundness. From L1 to L5, big-step evaluation interpreter has been used, in L6, Small-step and Big-step method is implemented. Big-step semantics or Small-step semantics, which one to choose depends on the size, complicity, programming language itself and more importantly, what to prove[Owe+16]. As for Big-step, expressions can be reduced to values, for example,

$$\frac{e \Downarrow \mathsf{Suc}^{n}(0)}{\mathsf{Suc}(e) \Downarrow \mathsf{Suc}^{n+1}(0)}$$

If e can be evaluated to a value of applying n times Suc to Zero, then $\mathsf{Suc}(e)$ should be evaluated to a value of applying $n+1$ times Suc to Zero.
As for Small-step, on the other hand, for example,

$$\frac{e \mapsto e'}{\mathsf{Suc}(e) \mapsto \mathsf{Suc}(e')}$$

evaluated expressions in small step, one step of evaluation a time. $\mapsto$ is from transition system, for example, $e \mapsto e'$ means state e may transition to state e'. In the example above, if e step evaluate to another expression e', then $\mathsf{Suc}(e)$ evaluates to $\mathsf{Suc}(e')$. In this case, e' may have the chance to continue the next evaluation step.

In this project, we need an evaluator to run our program on our computer, the evaluator will help evaluate if the system T constructed in L3 or L6 is correct or not by comparing evaluated values to expected values, and it will help check if all the condition is covered. Normally, evaluation will end up with a final value result which is actually a subset of the expression, and this design can later be tested by Tasty framework. Evaluator is also important when the benchmark evaluates the efficiency of the program. Accurate sample values should be passed into the benchmark, so the evaluator could run the program properly in the system T we designed.

2.4 Tasty

Tasty is a test framework which combines unit testing, property testing, etc in a single test package. It will automatically generate a report for your recent test, and the number of the tests, test cases and test time can be customized by user. Tests are run in parallel but report results are in a deterministic order.
Unit tests allow the user to write single inputs when a specific feature needs to be tested. Quick check helps create property tests which could tests several cases at the same time, but the cases generated are random and sometimes cannot find the potential problem in our program, but large amount of passed tests can always make sure it works well. The better way to test is shown in this project, we combined unit tests and property tests together to make sure as many cases as possible should test are tested.
The detail of this package can be found here: https://github.com/UnkindPartition/tasty

2.5 Criterion

Criterion can help users write micro-benchmark in Haskell. This library helps analyse performance of specified programs and compare the results. A report will be automatically generated and in this paper is attached at the appendix part.
The report will start with an overview of all the performances which can show data interactively if you move the mouse on the chart, and you could even choose the showing order when selected a specific feature in the drop-down menu. After the overview, detailed result of every each one of the groups you have compared will be shown in two charts, on the left is kernel density estimate (KDE) and on the right is the raw data from which the kernel density estimate is built. Under each pair of charts, there are also OLS regression, R^2 goodness-of-fit, Mean execution time and Standard deviation details. These results of our project will be attached in Appendix B.
The detail of this package can be found here: https://github.com/haskell/criterion

2.6 Prettyprinter

As a text rendering engine, Prettyprinter can print programs written in our language, PrettyPrinter uses Text data type, it supports Unicode and all kinds of expressions can be self-defined.
When there are too many iterative expressions, it it much easier to read if a more common mathematic expression can replace the expressions defined in the expression abstract syntax tree.

2.7 GADTs

Two different views of types which are Haskell Curry and Alonzo Church late refer to extrinsic and intrinsic types. Extrinsic semantics system means every phrase that satisfying typing judgment has the meaning, these typing judgments collected to give properties to this meaning.
However, there is no need to consider untyped expressions in intrinsic semantics, every expression defined in the abstract syntax tree has an independent and unique meaning which supports by the type judgement[Rey00].

In this project, we use GADTS to construct our intrinsically typed Abstract Syntax Trees. GADTs is short for Generalized Algebraic Data Types. It helps constructors have a non-uniform return type, which allows us to encode invariants about a data structure in its own type[Zil15]. It simply adds type annotations to the constructor defined in the project. After this type signature, the type checker is no longer required, because the signature bounds the type of the constructor, the constructor can only return the type it belongs to.

2.8 System T

Original System T, which was introduced by Gödel in 1958[Göd58] has simply typed lambda calculus with natural numbers and recursion. Gödel use System T dealt with Heyting Arithmetic, which leads to the availability of interpret Peano Arithmetic.
System T provides a primitive recursion, it can express to total higher-order primitive recursive

functions, all programs terminate.

In this project, Haskell is used as a base language to implement System T with more interesting features.

Chapter 3

L3: A language with numbers and booleans

In L3, System T with natural number and boolean will be constructed. In the syntax, we describe Types, Expressions, Values and Judgements. We also describe the statics typing rules and dynamic operational semantics.

3.1 Grammar

3.1.1 Syntax of L3

In L3, seven expressions are shown in the syntax. In simple L1 and L2, which are not displayed in this thesis, natural numbers and boolean are not constructed by ourselves but using the original ones in Haskell. In L3 every natural number can be written by EZero and ESucc, and True can be written in ETrue and False can be written in EFalse.

$$
\begin{array}{llll}
\textsc{Types} & T & ::= & \mathsf{Nat} \mid \mathsf{Bool} \\
\textsc{Expressions} & e & ::= & \mathsf{Zero} \mid \mathsf{Suc}(e) \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1 + e_2 \mid e_1 * e_2 \\
\textsc{Values} & v & ::= & \mathsf{Suc}^n(\mathsf{Zero}) \mid \mathsf{true} \mid \mathsf{false} \\
\textsc{Judgements} & \mathcal{J} & ::= & \vdash e : T
\end{array}
$$

Figure 3.1: Grammar of L3

3.1.2 Type checking

Bidirectional checking rules used in L3 has natural numbers and boolean types in expression, rules are written in two directions with synthesis and check, which synthesis is stricter in premises than check.

We could use if e_1 then e_2 else e_3, the most complicated one in L3 as an illustration example. In

Type Infer/Synthesis

$$\frac{}{\vdash \mathsf{Zero} \Rightarrow \mathsf{Nat}}\;\textsc{Zero} \qquad \frac{\vdash n \Leftarrow \mathsf{Nat}}{\vdash \mathsf{Suc}(n) \Rightarrow \mathsf{Nat}}\;\textsc{Suc}$$

$$\frac{}{\vdash \mathsf{true} \Rightarrow \mathsf{Bool}}\;\textsc{True} \qquad \frac{}{\vdash \mathsf{false} \Rightarrow \mathsf{Bool}}\;\textsc{False}$$

$$\frac{e_1 \Leftarrow \mathsf{Nat} \qquad e_2 \Leftarrow \mathsf{Nat}}{\vdash e_1 + e_2 \Rightarrow \mathsf{Nat}}\;\textsc{EAdd} \qquad \frac{e_1 \Leftarrow \mathsf{Nat} \qquad e_2 \Leftarrow \mathsf{Nat}}{\vdash e_1 * e_2 \Rightarrow \mathsf{Nat}}\;\textsc{EMul}$$

$$\frac{e_1 \Leftarrow \mathsf{Bool} \qquad e_2 \Rightarrow T \qquad e_3 \Rightarrow T' \qquad T = T'}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow T}\;\textsc{EIf}$$

Type check

$$\frac{}{\vdash \mathsf{Zero} \Leftarrow \mathsf{Nat}}\;\textsc{Zero} \qquad \frac{\vdash n \Leftarrow \mathsf{Nat}}{\vdash \mathsf{Suc}(n) \Leftarrow \mathsf{Nat}}\;\textsc{Suc}$$

$$\frac{}{\vdash \mathsf{true} \Leftarrow \mathsf{Bool}}\;\textsc{True} \qquad \frac{}{\vdash \mathsf{false} \Leftarrow \mathsf{Bool}}\;\textsc{False}$$

$$\frac{e_1 \Leftarrow \mathsf{Nat} \qquad e_2 \Leftarrow \mathsf{Nat}}{\vdash e_1 + e_2 \Leftarrow \mathsf{Nat}}\;\textsc{EAdd} \qquad \frac{e_1 \Leftarrow \mathsf{Nat} \qquad e_2 \Leftarrow \mathsf{Nat}}{\vdash e_1 * e_2 \Leftarrow \mathsf{Nat}}\;\textsc{EMul}$$

$$\frac{e_1 \Leftarrow \mathsf{Bool} \qquad e_2 \Leftarrow T \qquad e_3 \Leftarrow T}{\vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Leftarrow T}\;\textsc{EIf}$$

Figure 3.2: Bidirectional checking rules of L3

synthesis rule, the input is "if e_1 then e_2 else e_3", the output is type T. In the premises, e_1 has to checked to a Bool type and if e_2 can be inferred to type T, e_3 can be inferred to type T', the conclusion can only stand if T equals to T'.

3.1.3 Semantics

Big-step oprational semantics

Figure 3.3 shows the Big-step semantics in L3.

3.1.4 Implementation

We implement L3 in Haskell, using Haskell data types to encode the abstract syntax trees for types, expressions, and values. Then, we implement the bidirectional type-checker, and big-step evaluator.

$$\frac{}{\mathsf{Zero} \Downarrow \mathsf{Zero}}\ \textsc{Zero} \qquad \frac{e \Downarrow \mathsf{Suc}^{n}(\mathsf{Zero})}{\mathsf{Suc}(e) \Downarrow \mathsf{Suc}^{n+1}(\mathsf{Zero})}\ \textsc{Suc}$$

$$\frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ \text{E}\textsc{True} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ \text{E}\textsc{False}$$

$$\frac{e_1 \Downarrow \mathsf{Suc}^{n_1}(0) \qquad e_2 \Downarrow \mathsf{Suc}^{n_2}(\mathsf{Zero})}{e_1 + e_2 \Downarrow \mathsf{Suc}^{n_1+n_2}(\mathsf{Zero})}\ \text{E}\textsc{Add} \qquad \frac{e_1 \Downarrow \mathsf{Suc}^{n_1}(\mathsf{Zero}) \qquad e_2 \Downarrow \mathsf{Suc}^{n_2}(\mathsf{Zero})}{e_1 * e_2 \Downarrow \mathsf{Suc}^{n_1 * n_2}(\mathsf{Zero})}\ \text{E}\textsc{Mul}$$

$$\frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow v_2}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_2}\ \text{E}\textsc{If}\text{-TRUE} \qquad \frac{e_1 \Downarrow \mathsf{false} \qquad e_3 \Downarrow v_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_3}\ \text{E}\textsc{If}\text{-FALSE}$$

Figure 3.3: Big-step semantics of L3

Type

L3 has two data type, TNat and TBool, as defined in type syntax.

```
data Ty
  = TNat
  | TBool
```

Figure 3.4: Type definition in L3

Expression

An abstract syntax tree(AST) is wildly used in defining expressions, it is an ordered tree, and whose interior nodes are operators whose arguments are its children[Har16].
AS figure 3.5 shows, Expression in L3 is defined using AST, which is written using an algebraic data type(ADT) in Haskell, it is defined iteratively and can capture essential input while at the same time omit unnecessary syntactic details like tree nodes[Jon03].

```
data Exp
  = EZero
  | ESucc Exp
  | ETrue
  | EFalse
  | EAdd Exp Exp
  | EMul Exp Exp
  | EIf Exp Exp Exp
```

Figure 3.5: Expression definition in L3

Value

Value in figure 3.6 is the evaluation result of all the expressions in L3, it is a subset of expressions but defined separately here in L3. Here Nat is directly used to consist all the natural numbers, VSuccN(Nat) means Suc^n(Zero) apply Suc n times. notice that VSuccN 0 equals to 0, which is quite different from the 'successor 0', which equals to 1. VTrue means true and VFalse means false.

```
data Val
  = VSuccN Nat
  | VTrue
  | VFalse
```

Figure 3.6: Value defination of L3

Type checking

In type checking, Monad has been derived in a new defined TC type in figure 3.7:

```
newtype TC a = TC {runTC :: Either TCError a}
  deriving (Eq, Show, Functor, Applicative, Monad)
```

Figure 3.7: TC type

An either type is defined in the TC type where left is an error message which has the type of String and right is the value of the Unit type returns.
tccheck is used to check if the expression has a proper given type: while tcinfer synthsis a type

```
tccheck :: Exp -> Ty -> TC ()
```

Figure 3.8: tccheck

from expression, as reference discussed in 2.1:

```
tcinfer :: Exp -> TC Ty
```

Figure 3.9: tcinfer

Big-step evaluation

We implemented Big-step rules in L3, since values are defined and available in L3, a direct big step evaluation is constructed by:

```
eval :: Exp -> Maybe Val
```

Figure 3.10: Big-step evaluation

The benefit of using Maybe is it can easily return nothing or just the value. Do notation has been used in the eval function to cover all the possibilities in expressions.

3.1.5 Testing

The test suite is one of the main parts in our project. In L3, tccheck and tcinfer have been tested by the property tests in all kinds of expressions:

- if a type can be checked with Nat, then it will also be inferred to Nat.
- if a type can be checked with Bool, then it will also be inferred to Bool.
- The text in the entries may be of any length.

are the two cases ran by the Qucikcheck.
A new type defined for another two test cases

```
newtype TcTyExp = TcTyExp {tcgetExp :: Exp}
```

Figure 3.11: Well typed expression

- every well-typed expression can be inferred.
- every well-typed expression can be checked for its type.

which bounded expression with ”well-typed”, well-typed programs cannot go wrong [Mil78], means these generated expressions are meaningful and evaluable. We ensure the expression generated has a type, by using tcinfer, it is well-typed if tcinfer success.
Another test is about evaluate function,

- Well-typed expressions reduced to a value

which means every expression generated by the arbitrary defined will be reduced to the value defined.

3.2 Intrinsic L3

In intrinsic L3, type definition still remains the same as the extrinsic ones. However, the expressions are changed since every constructor includes a type signature now, details are in figure 3.12.

```
data Exp :: Ty -> Type where
  EZero :: Exp 'TNat
  ESucc :: Exp 'TNat -> Exp 'TNat
  ETrue :: Exp 'TBool
  EFalse :: Exp 'TBool
  EAdd :: Exp 'TNat -> Exp 'TNat -> Exp 'TNat
  EMul :: Exp 'TNat -> Exp 'TNat -> Exp 'TNat
  EIf :: Exp 'TBool -> Exp ty -> Exp ty -> Exp ty
```

Figure 3.12: Intrinsic expression

```
data Val :: Ty -> Type where
  VSuccN :: Nat -> Val 'TNat
  VTrue :: Val 'TBool
  VFalse :: Val 'TBool
```

Figure 3.13: Intrinsic value

As figure 3.13 shows, Values are changed in the same way with the type signature.
In the evaluation part of intrinsic one in L3, evaluation uses case to make sure the value is correct, the type of eval is:

```
eval :: Exp ty -> Val ty
```

Figure 3.14: eval in Intrinsic L3

Take if e_1 then e_2 else e_3 for example, we don't have to check if e_2 and e_3 has the same type, because it cannot be wrong since type signature has bound what type it can use. So we only have to use $case$ to check if e_1 is VTrue or VFalse.

$$\frac{e_1 \Downarrow \text{true}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_2}\ \textsc{EIf}\text{-TRUE} \qquad \frac{e_1 \Downarrow \text{false}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_3}\ \textsc{EIf}\text{-FALSE}$$

Chapter 4

L6: A language with many types

L6 is based on L3 and there are actually L4 and L5 implementation in between, omitted details of L4 and L5 can be found in source code.
In L6, a much larger type system has been created. In L6, type includes natural number, boolean, unit, product and function. Since lambda recursion is implemented in L6, higher order recursive functions can work well in L6. An extra context is put into use in L6 to make sure every time variables can be looked up and checked in the context. In addition, reduction rules are defined in L6 too to make the evaluation much more efficient.

4.1 Grammar

4.1.1 Syntax of L6

L6 is much more complicated and supports functions, which EAdd and EMul were dropped from the expression because they are no longer needed since they can be constructed by primitive recursion. We have context in L6 which can be understood as a list of variable, which we use to lookup when doing calculations, extend when new variables arrive. Judgements here have variables have type T belong to Γ or with the judgments and context Γ we can derive the expression has the type T, or we could check whether the expression is a value defined by judgement e val [Har16].

$$
\begin{array}{llll}
\textsc{Types} & T & ::= & \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{Unit} \mid T \times T \mid T \to T \\
\textsc{Expressions} & e & ::= & \mathsf{Zero} \mid \mathsf{Suc}(e) \mid \mathsf{true} \mid \mathsf{false} \mid \\
 & & & * \mid \mathsf{Fst}(e) \mid \mathsf{Snd}(e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \\
 & & & x \mid \lambda(x:T).e \mid e_1\ e_2 \mid (e_1, e_2) \mid \\
 & & & \mathsf{Iter}(e_1, e_2, e_3) \\
\textsc{Context} & \Gamma & ::= & \bullet \mid \Gamma, x : A \\
\textsc{Judgements} & \mathcal{J} & ::= & x : T \in \Gamma \mid \Gamma \vdash e : T \mid e\ \mathit{val}
\end{array}
$$

Figure 4.1: Grammar of L6

4.1.2 Type checking L6

Type Infer/Synthesis

$$\frac{}{\Gamma \vdash \mathsf{Zero} \Rightarrow \mathsf{Nat}}\ \textsc{Zero} \qquad \frac{\Gamma \vdash n \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Suc}(n) \Rightarrow \mathsf{Nat}}\ \textsc{Suc}$$

$$\frac{}{\Gamma \vdash \mathsf{true} \Rightarrow \mathsf{Bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathsf{false} \Rightarrow \mathsf{Bool}}\ \textsc{False}$$

$$\frac{}{\Gamma \vdash * \Rightarrow *}\ \textsc{Unit}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e_2 \Rightarrow T \qquad \Gamma \vdash e_3 \Rightarrow T' \qquad T = T'}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow T}\ \textsc{EIf}$$

$$\frac{\Gamma \vdash e \Rightarrow T \times T'}{\Gamma \vdash \mathsf{Fst}(e) \Rightarrow T}\ \textsc{Fst} \qquad \frac{\Gamma \vdash e \Rightarrow T \times T'}{\Gamma \vdash \mathsf{Snd}(e) \Rightarrow T'}\ \textsc{Snd}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Var}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda(x : A).e \Rightarrow A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1 e_2 \Rightarrow B}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Leftarrow A \to A \qquad \Gamma \vdash e_3 \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Iter}(e_1, e_2, e_3) \Rightarrow A}\ \textsc{Iter}$$

Type check

$$\frac{}{\Gamma \vdash \mathsf{Zero} \Leftarrow \mathsf{Nat}}\ \textsc{Zero} \qquad \frac{\Gamma \vdash n \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Suc}(n) \Leftarrow \mathsf{Nat}}\ \textsc{Suc}$$

$$\frac{}{\Gamma \vdash \mathsf{true} \Leftarrow \mathsf{Bool}}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \mathsf{false} \Leftarrow \mathsf{Bool}}\ \textsc{False}$$

$$\frac{}{\Gamma \vdash * \Leftarrow *}\ \textsc{Unit}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e_2 \Leftarrow T \qquad \Gamma \vdash e_3 \Leftarrow T' \qquad T = T'}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Leftarrow T}\ \textsc{EIf}$$

$$\frac{\Gamma \vdash e \Rightarrow T \times T'}{\Gamma \vdash \mathsf{Fst}(e) \Leftarrow T}\ \textsc{Fst} \qquad \frac{\Gamma \vdash e \Rightarrow T \times T'}{\Gamma \vdash \mathsf{Snd}(e) \Leftarrow T'}\ \textsc{Snd}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Leftarrow A}\ \textsc{Var}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda(x : A).e \Leftarrow A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1 e_2 \Rightarrow B}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A \qquad \Gamma \vdash e_2 \Leftarrow A \to A \qquad \Gamma \vdash e_3 \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{Iter}(e_1, e_2, e_3) \Leftarrow A}\ \textsc{Iter}$$

4.1.3 Symantics L6

Small-step semantics

Figure 4.2 shows the Small-step semantics in L6.

Big-step semantics

Figure 4.3 shows the Big-step semantics in L6.

4.1.4 Implementation

Types

As mentioned earlier in this chapter, additional types are added to the abstract syntax tree:

$$\frac{e \mapsto e'}{\mathsf{Suc}(e) \mapsto \mathsf{Suc}(e')}\ \textsc{Suc}$$

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\ \textsc{Tuple-Left} \qquad \frac{e_2 \mapsto e_2'}{(e_1, e_2) \mapsto (e_1, e_2')}\ \textsc{Tuple-Righ}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3}\ \textsc{EIf}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_2}\ \textsc{EIf-True} \qquad \frac{}{\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_3}\ \textsc{EIf-False}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{EApp} \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{EApp} \qquad \frac{}{(\lambda x.e_1)e_2 \mapsto [e_2/x]e_1}\ \textsc{EApp}$$

$$\frac{}{\mathsf{Fst}((e_1, e_2)) \mapsto e_1}\ \textsc{EFst} \qquad \frac{}{\mathsf{Snd}((e_1, e_2)) \mapsto e_2}\ \textsc{ESnd}$$

$$\frac{e \mapsto e'}{\mathsf{Fst}(e) \mapsto \mathsf{Fst}(e')}\ \textsc{EFst} \qquad \frac{e \mapsto e'}{\mathsf{Snd}(e) \mapsto \mathsf{Snd}(e')}\ \textsc{ESnd}$$

$$\frac{}{\mathsf{Iter}(e_1, e_2, \mathsf{Zero}) \mapsto e_1}\ \textsc{EIter} \qquad \frac{}{\mathsf{Iter}(e_1, e_2, \mathsf{Suc}(e_3)) \mapsto e_2\ \mathsf{Iter}(e_1, e_2, e_3)}\ \textsc{EIter}$$

$$\frac{e_3 \mapsto e_3'}{\mathsf{Iter}(e_1, e_2, e_3) \mapsto \mathsf{Iter}(e_1, e_2, e_3')}\ \textsc{EIter}$$

Figure 4.2: Small-step semantics of L6

Context

Environment should be defined in L6 since variables are available now, a list of variables can be looked up in the context when constructing lambda calculation.

Look up function

A look-up function is needed in the process when a variable needs to compare to the existed variables in the context: It returns a Maybe type, if the variable get the match to the current variable in the environment, then returns the expression. If it is not a match, then exclude the current variable and look up in the rest of the environment again, a recursive search is constructed. If finally no variable matches, then a Nothing should be returned.
An extension in context is also available, simply using snoc here, which likes cons but saving items in a reverse order:

$$\frac{}{\mathsf{Zero} \Downarrow \mathsf{Zero}}\ \textsc{Zero} \qquad \frac{e \Downarrow e' \qquad e'\ val}{\mathsf{Suc}(e) \Downarrow \mathsf{Suc}(e')}\ \textsc{Suc}$$

$$\frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ \textsc{True} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ \textsc{false}$$

$$\frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow e_2' \qquad e_2'\ val}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow e_2'}\ \textsc{EIf-True} \qquad \frac{e_1 \Downarrow \mathsf{false} \qquad e_3 \Downarrow e_3' \qquad e_3'\ val}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow e_3'}\ \textsc{EIf-False}$$

$$\frac{e_3 \Downarrow \mathsf{Zero} \qquad e_1 \Downarrow e_1' \qquad e_1'\ val}{\mathsf{Iter}(e_1, e_2, e_3) \Downarrow e_1'}\ \textsc{EIter-Zero}$$

$$\frac{e_3 \Downarrow \mathsf{Suc}(e_3') \qquad e_2\ \mathsf{Iter}(e_1, e_2, e_3') \Downarrow e_4 \qquad e_4\ val}{\mathsf{Iter}(e_1, e_2, e_3) \Downarrow e_4}\ \textsc{EIter}$$

$$\frac{}{* \Downarrow *}\ \textsc{EUnit} \qquad \frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2' \qquad e_1'\ val \qquad e_2'\ val}{(e_1, e_2) \Downarrow (e_1', e_2')}\ \textsc{ETuple}$$

$$\frac{e \Downarrow (e_1, e_2) \qquad e_1\ val}{\mathsf{Fst}(e) \Downarrow e_1}\ \textsc{Fst} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_2\ val}{\mathsf{Snd}(e) \Downarrow e_2}\ \textsc{Snd}$$

$$\frac{}{\lambda(e_1 : A).e_2 \Downarrow \lambda(e_1 : A).e_2}\ \textsc{Elam} \qquad \frac{e_1 \Downarrow \lambda(x : A).e}{e_1\ e_2 \Downarrow [e_2/x]e}\ \textsc{EApp}$$

Figure 4.3: Big-step semantics of L6

```
data Ty
  = TNat
  | TBool
  | TUnit
  | TProd Ty Ty
  | TFun Ty Ty
```

Figure 4.4: Type defination in L6

```
data Ctx
  = Emp
  | Snoc Ctx (Name, Ty)
```

Figure 4.5: Expression definition L6

```
lookupCtx :: Name -> Ctx -> Maybe Ty
```

Figure 4.6: Defination of lookupCtx

```
extendCtx :: Name -> Ty -> Ctx -> Ctx
extendCtx x ty ctx = Snoc ctx (x, ty)
```

Figure 4.7: Definition of extended context

Expression

Expressions are always the core part of the language, they consist all the basic expressions user need. In L6, EAdd and EMul are dropped from the abstract syntax tree because they can be represented by the recursive expression. They are later shown in implementation part.
The deriving part needs Generic and NFData because later on a benchmark performance will need these expressions evaluated, so instcance should be created here.

```
data Exp
  = EZero
  | ESucc Exp
  | ETrue
  | EFalse
  | EIf Exp Exp Exp
  | EUnit
  | ETuple Exp Exp
  | EFst Exp
  | ESnd Exp
  | EVar Name --variables
  | ELam Name Ty Exp --abstraction
  | EApp Exp Exp --application
  | EIter Exp Exp Exp
```

Figure 4.8: Expression in L6

Reduction rules

Evaluate the Lambda term by processing several steps of substitution. There are three rules in Lambda calculus, alpha equivalence, beta-reduction and Eta-reduction, these three work together to make evaluation process working properly.

```
subst :: Name -> Exp -> Exp -> Exp
```

Figure 4.9: General type of Reduction rules in L6

Substitution rules is a trick to make expressions much tidier and simpler and easier to compute, which is an identity transformation and remains the value of the original expression.

4.1.5 Function Testing

An implementation in L6 is conducted by several basic and higher order functions, every one of them can be defined recursively, which Iter maximum its usage in the calculation. They are also implemented in L6 to test the ability of our language in constructing primitive recursion.

IsZero

$$\mathsf{isZero} :: \mathsf{Nat} \rightarrow \mathsf{Bool}$$

$$\mathsf{isZero} = \lambda(n : \mathsf{Nat}).\mathsf{Iter}(\mathsf{true}, \lambda(b : \mathsf{Bool}).\mathsf{false}, n)$$

IsZero is simply check if the input of the Nat is Zero or not. A traditional Haskell function is used to verify a natural number, if it is a Zero, then after evaluate, it returns true, if not, returns false.

Pred

$$\mathsf{predExp} :: \mathsf{Nat} \rightarrow \mathsf{Nat}$$

$$\mathsf{predExp} = \lambda(n : \mathsf{Nat}).\mathsf{Fst}(\mathsf{Iter}((\mathsf{Zero}, \mathsf{Zero}), \lambda(t : (\mathsf{Nat} \times \mathsf{Nat})).(\mathsf{Snd}(t), \mathsf{Suc}(\mathsf{Snd}(t))), n))$$

Pred is simply decreasing numbers by 1, which is the reverse of the Succ, it uses primitive recursion, which use tuples to select recursive needed number.
If the input is n , then it helps applies the second parameters n times to the initial tuple, which is $(0, 0)$, after one time, the result will be $(0, 1)$, second time the result will be $(1, 2)$, third time the result will be $(2, 3)$, so on so forth, at the end the result will be $(n - 1, n)$, and we select the first item of the tuple to get the final result. Please notice that we are working on natural numbers, so if the input is 0, the result will still be 0.

Add

$$\mathsf{addExp} :: \mathsf{Nat} \rightarrow (\mathsf{Nat} \rightarrow \mathsf{Nat})$$

$$\mathsf{addExp} = \lambda(n : \mathsf{Nat}).\lambda(m : \mathsf{Nat}).\mathsf{Iter}(m, \lambda(t : \mathsf{Nat}).\mathsf{Suc}(t), n)$$

Add is designed to take two Nat inputs and calculate the sum of them. It has the type of

$$\mathsf{Nat} \rightarrow (\mathsf{Nat} \rightarrow \mathsf{Nat})$$

in our design, which follows the Haskell function right combination rules. The idea of addition is a repeated Suc, for example, if we input m and n, it applies Suc to the input n for m times.
Note that doubleExp has been implemented in our project, too. It works as the same way as addExp does, it applies Suc to the input n for n times.

Fib

$$\mathsf{fibExp} :: \mathsf{Nat} \to \mathsf{Nat}$$

$$\mathsf{fibTmp} = \lambda(n : \mathsf{Nat}).\mathsf{Iter}((\mathsf{Zero}, \mathsf{Suc}(\mathsf{Zero})), \lambda(t : (\mathsf{Nat} \times \mathsf{Nat})).(\mathsf{Snd}(t), \mathsf{addExp}\ \mathsf{Fst}(t)\ \mathsf{Snd}(t)), n)$$

$$\mathsf{fibExp} = \lambda(m : \mathsf{Nat}).\mathsf{Fst}(\mathsf{fibTmp}\ m)$$

Fib is the construction on Fibonacci numbers, the recursive calculation is easily becoming really time-consuming, so during the test we only generate numbers lower than 10. Here the addExp is used to calculate, the idea is to maintain a list of tuples : (0,1) (1,1) (1,2) (2,3) (3,5) ... and the first column of the tuple list is retrieved after construction which is the Fibonacci number we want.

Mul

$$\mathsf{mulExp} :: \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$$

$$\mathsf{mulExp} = \lambda(n : \mathsf{Nat}).\mathsf{Iter}(\mathsf{Zero}, \mathsf{addExp}\ m, n)$$

Mul is the multiplication, which is constructed based on addExp, the idea of multiplication is to transform it to addition, for example m*n will be added m n-1 times and finally add m itself.

Expo

$$\mathsf{expoExp} :: \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$$

$$\mathsf{expoExp} = \lambda(n : \mathsf{Nat}).\lambda(m : \mathsf{Nat}).\mathsf{Iter}(\mathsf{Suc}(\mathsf{Zero}), \mathsf{mulExp}\ n, m)$$

Expo is the function of exponentiation, the idea is similar to Mul, exponentiation is constructed by multiple several times, for example, m**n is multiple m n times which will get the result of the exponential.

Tet

$$\mathsf{tetExp} :: \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$$

$$\mathsf{tetExp} = \lambda(a : \mathsf{Nat}).\lambda(n : \mathsf{Nat}).\mathsf{Iter}(\mathsf{Suc}(\mathsf{Zero}), \mathsf{expoExp}\ a, n)$$

Tetration is a similar to exponential function, it is based on iterated, or repeated, exponentiation. It will increase rapidly, which the test cases can not be too big, or it will take a long time.

Factorial

$$\mathsf{facExp} :: \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$$

$$\mathsf{facExp2} = \lambda(n : \mathsf{Nat}).\mathsf{Iter}((\mathsf{Zero}, \mathsf{Suc}(\mathsf{Zero})), \lambda(t : (\mathsf{Nat} \times \mathsf{Nat})).(\mathsf{Suc}(\mathsf{Fst}(t)), \mathsf{mulExp}\ \mathsf{Suc}(\mathsf{Fst}(t))\ \mathsf{Snd}(t)), n)$$

$$\mathsf{facExp} = \lambda(m : \mathsf{Nat}).\mathsf{Snd}(\mathsf{facExp2}\ m)$$

Factorial function is easier than fibnacii function. It has only one recursive call, for example, input n, then during the iteration and decrease of n, a sequence of $n \times (n-1) \times (n-2) \cdots$ multiplication is constructed and finally applied to 1.

Ackermann

Ackermann function was published by Wilhelm Ackermann in 1928 [Ack28]. The original Ackermann function defines with three paramters, however, the most common one that easy to understand is defined as follows:

$$\mathsf{A}(0, n) = n + 1$$

$$\mathsf{A}(m + 1, 0) = \mathsf{A}(m, 1)$$

$$\mathsf{A}(m + 1, n + 1) = \mathsf{A}(m, \mathsf{A}(m + 1, n))$$

We constructed it in system T in the following way:

$$\mathsf{ackerExp} :: \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$$

$$\mathsf{compExp} = \lambda(t : ((\mathsf{Nat} \to \mathsf{Nat}) \times (\mathsf{Nat} \to \mathsf{Nat}))).\lambda(x : \mathsf{Nat}).\mathsf{Fst}(t)\ \mathsf{Snd}(t)\ x$$

$$\mathsf{itExp} = \lambda(f : (\mathsf{Nat} \to \mathsf{Nat})).\lambda(n : \mathsf{Nat}).\mathsf{Iter}(\lambda(x : \mathsf{Nat}).x, \lambda(g : (\mathsf{Nat} \to \mathsf{Nat})).\mathsf{compExp}\ (f, g), n)$$

$$\mathsf{sExp} = \lambda(m : \mathsf{Nat}).\mathsf{Suc}(m)$$

$$\mathsf{rExp} = \lambda(f : (\mathsf{Nat} \to \mathsf{Nat})).\lambda(m : \mathsf{Nat}).\mathsf{itExp}\ f\ m\ f\ \mathsf{Suc}(\mathsf{Zero})$$

$$\mathsf{ackerExp} = \lambda(n : \mathsf{Nat}).\mathsf{Iter}(\mathsf{sExp}, \mathsf{rExp}, n)$$

Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive. It grows faster than an exponential function, or even a multiple exponential function [Wei96].

$\mathsf{A}(m, n)$is the most complicated one among all the functions we used to test L6. It grows really fast and consumes a lot of memory, we should be careful when we do the test on this particular function. Ackermann function is very special, it is constructed by lexicographic induction on the pair of arguments. Either m decreases or n decreases iteratively [Har16]. Our implementation is slow due to plenty of recursion in $\mathsf{A}(m, n)$, we need to optimize the recursion by loops which is done by GHC optimizer.

It is a higher-order primitive recursive function, and definable in system T, we have to realize that every $\mathsf{A}(m, n)$ can be written as n iterations of $\mathsf{A}(m - 1, -)$ finally applies to $\mathsf{A}(m - 1, 1)$. This triggers an idea of using a function composition, which means in recursion, we could recursively apply $\mathsf{A}(m - 1, -)$ until the last $\mathsf{A}(m - 1, 1)$, compExp compose the f and g, g represent the $f(1)$ in our design. Since composition is based on two function, the first parameter of the recursion should be an identity function.

In our design, compExp is the composition of two functions, itExp is used to control how many iterations $\mathsf{A}(m-1,-)$ applies. sExp is the Suc function and rExp is using itExp and passed the final $f(1)$ to it.

Chapter 5

Evaluation

5.1 Tests

Property tests

What property test do is generating random expressions under the rules developer writes and check if the random generated examples satisfied a specific property the program should fulfil. Here tests are using Testy test framework and every test goes parallel, in L6 test are applied to Big-step and Small-step separately. In each one of them, there are four different property tests, which conclude

- Progress: Type-inferable expressions always reduce to a value.
- Type-preservation: Type-inferable expressions reduce to a value of the same type.
- Progress: Well-typed expressions always reduce to a value.
- Type-preservation: Well-typed expressions reduce to a value of the same type.

all tests have been tested in an acceptable number of cases and time. Sometimes calculation goes huge and memory consuming, so limit the number and time of test is important. Tasty is good enough to provide such features for user to make the control. In addition, arbitrary expression

```
QC.withMaxSuccess 1_000_000 -- test 1 million times
QC.within 5_000_000 -- timeout after 5s
```

Figure 5.1: General way to control Property Tests in L6

can be defined separately in Tasty and it is very helpful on constructing well-typed expression.

Unit tests

Unit tests are flexible as it is from L1 to L5, any specific tests you want to run should be put into a unit test, sometimes unit test is the quickest way to check if the specific expression is success or not.

5.2 Benchmark

Benchmark is used to mainly test and compare the efficiency of the example programs written in traditional Haskell language and our L6 language. There are two other tests added to the benchmark, one is function eval, opt and opt2 in L1 which are optimizers constructed in different ways, they all worked well but with different efficiency. Another one is Add in L5 and L6, in L5, addition is primitive, and in L6 recrusivly constructed add is used.
Also O0 and O2 have been applied to see the difference, O0 is literally turn off all the optimization while O2 on the other side apply as much non-dangerous optimization as possible to quickly generate the code even during compile it takes longer time.
Figure 5.2 is the general result of the function performance in O2, Mean means mean excuation time of all the loops, MeanLB means Mean lower bound, MeanUB means Mean upper bound, Stddev means standard deviation, StddevLB means standard deviation lower bound, StddevUB means standard upper bound.
From figure 5.2 and appendix B we could see that in L1 even we tried optimizer, it still consumes much more time than original Haskell one. Small-step evaluation takes less time than Big-step evaluation isZero, predExp, fibExp, but take longer time than Big-step in addExp, doubleExp, expoExp, facExp, mulExp, tetExp, ackerExp.

Please note that the same function can be compared horizontally, but different functions compare cannot compare horizontally because due to computation time limit reason, we use different cases between different functions.

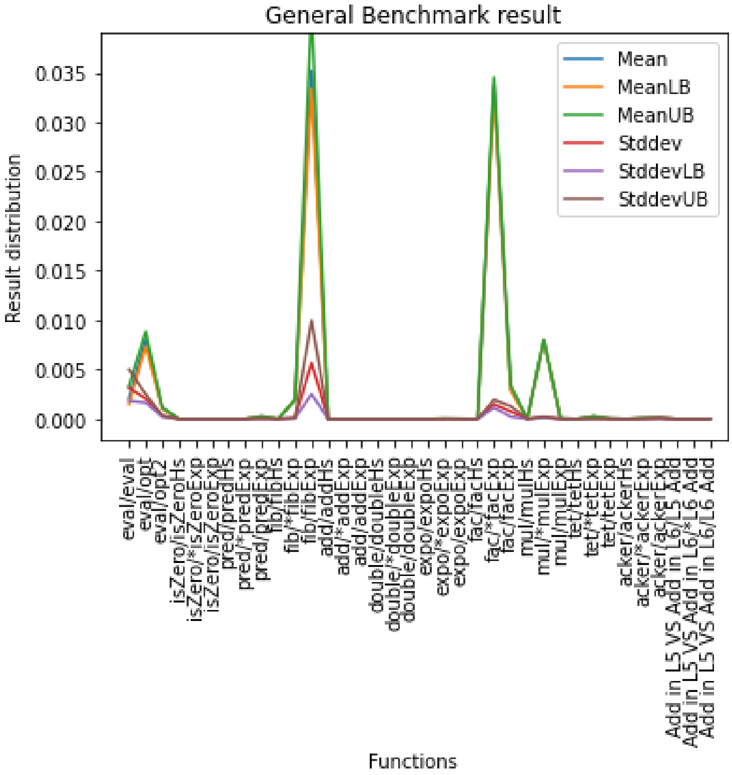


Figure 5.2: Funtion general benchmark result

Chapter 6

Conclusion

In our implementation, system T is successfully realized in L6 with multiple types. The recursive lambada calculus is the main point to construct function, it is easier and tidier for developers to write. Multiple functions have been tested by tasty and got successful result.

However, the language we built still need further work to do to improve the efficiency. Due to the result mentioned in benchmark, O2 optimization of Haskell has already been put into use, but time consumption of Small-step and Big-step stays high. The reason Big-step and Small-step have difference time-consuming performance in different function still needs to discover. In addition, intrinsic version of L6 still need to be done in lambda expressions part, the lookup judgement has been lifted to the In type, so we can only construct bound variables. To extract the variable name from an instance of In ctx b, we will need to demote the b to a Bind and pattern match on it.

Right now, there are some version conflicts between our language environment and Liquidhaskell requirements, but there are potential chance to work with Liquidhaskell in our language to do verification in algorithm in the future.

Bibliography

[Ack28] W. Ackermann. “Zum Hilbertschen Aufbau der reellen Zahlen.” In: *Math. Ann.* 99 (1928), pp. 118–133.

[Göd58] Von Kurt Gödel. “ÜBER EINE BISHER NOCH NICHT BENÜTZTE ERWEITERUNG DES FINITEN STANDPUNKTES.” In: *Dialectica* 12.3-4 (1958), pp. 280–287. DOI: https://doi.org/10.1111/j.1746-8361.1958.tb01464.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1746-8361.1958.tb01464.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1746-8361.1958.tb01464.x.

[Gog75] Joseph A Goguen. “Semantics of computation.” In: *Category theory applied to computation and control*. Springer, 1975, pp. 151–163.

[Mil78] Robin Milner. “A theory of type polymorphism in programming.” In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[Hal+92] Cordelia Hall et al. “The Glasgow Haskell Compiler: A Retrospective.” In: Jan. 1992, pp. 62–71. ISBN: 978-3-540-19820-8. DOI: 10.1007/978-1-4471-3215-8_6.

[Aab96] A.A. Aaby. *Introduction to Programming Languages*. 1996. URL: https://books.google.co.uk/books?id=WT6fMwEACAAJ.

[Wei96] Eric W. Weisstein. *Ackermann Function*. https://archive.lib.msu.edu/crcmath/math/math/a/a044.htm. Accessed: 2021-12-11. 1996.

[Rey00] John C. Reynolds. “The Meaning of Types From Intrinsic to Extrinsic Semantics.” In: *BRICS Report Series* 7.32 (June 2000). DOI: 10.7146/brics.v7i32.20167. URL: https://tidsskrift.dk/brics/article/view/20167.

[Jon03] Joel Jones. “Abstract Syntax Tree Implementation Idioms.” In: *Pattern Languages of Program Design* (2003). Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html. URL: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[Pfe04] Frank Pfenning. *Lecture Notes on Bidirectional Type Checking*. Oct. 2004.

[Har05] Robert Harper. “Mechanizing the Meta-Theory of Programming Languages.” In: *SIGPLAN Not.* 40.9 (Sept. 2005), p. 240. ISSN: 0362-1340. DOI: 10.1145/1090189.1086396. URL: https://doi.org/10.1145/1090189.1086396.

[NN07] H.R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer London, 2007. ISBN: 9781846286926. URL: https://books.google.co.uk/books?id=oPi0yERDUeYC.

[Zil15] Noam Zilberstein. *CIS 194: Introduction to Haskell (Spring 2015):GADTs in Action.* Apr. 2015.

[Har16] Robert Harper. *Practical foundations for programming languages, second edition.* Jan. 2016, pp. 1–476. DOI: 10.1017/CBO9781316576892.

[Owe+16] Scott Owens et al. "Functional Big-Step Semantics." In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Berlin, Heidelberg: Springer-Verlag, 2016, pp. 589–615. ISBN: 9783662494974. DOI: 10.1007/978-3-662-49498-1_23. URL: https://doi.org/10.1007/978-3-662-49498-1_23.

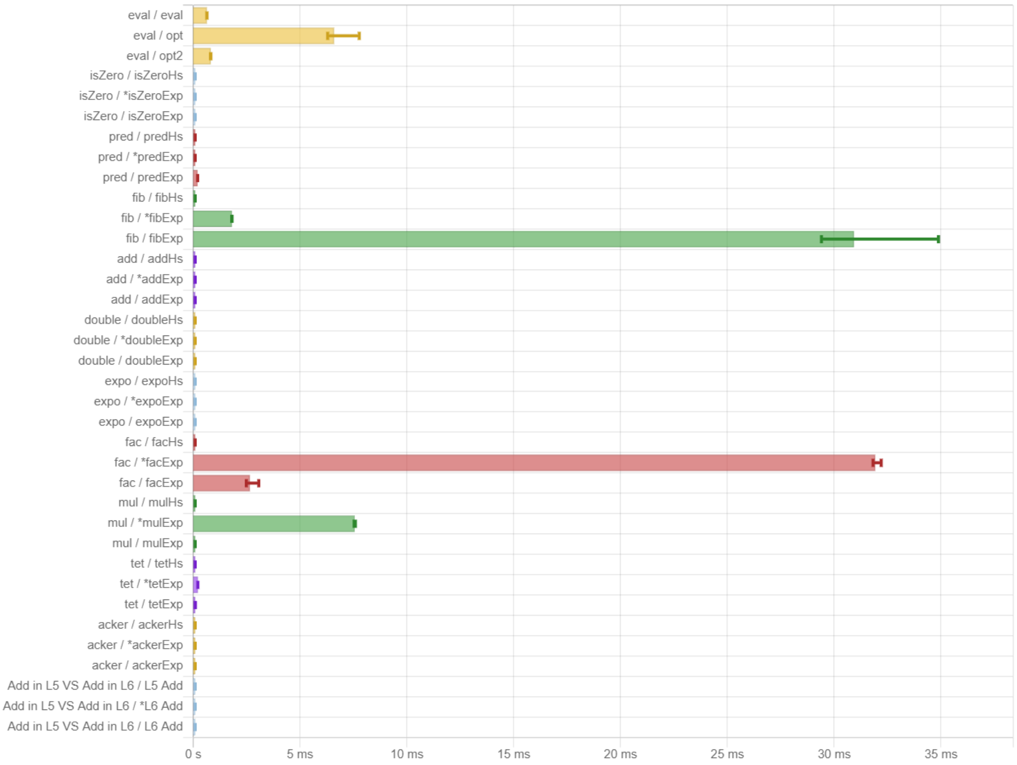
[Chr19] David Thrane Christiansen. *Checking Dependent Types with Normalization by Evaluation: A Tutorial (Haskell Version).* May 2019.

Appendix A

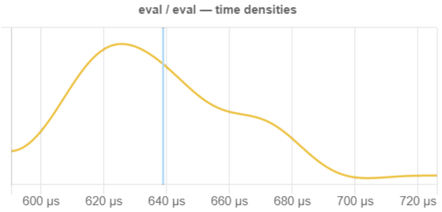
Benchmark result

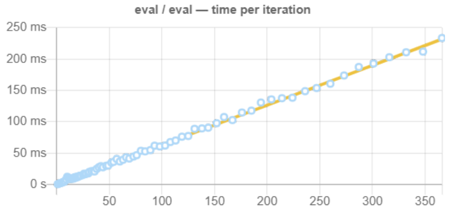
criterion performance measurements

overview



eval / eval

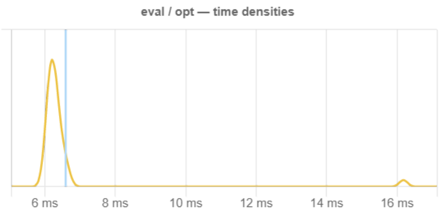


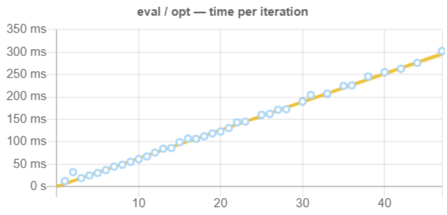


	lower bound	**estimate**	upper bound
OLS regression	624 μs	633 μs	641 μs
R² goodness-of-fit	0.997	0.998	0.999
Mean execution time	633 μs	639 μs	647 μs
Standard deviation	20.4 μs	24.6 μs	32.9 μs

Outlying measurements have a moderate (30.3%) effect on estimated standard deviation.

eval / opt

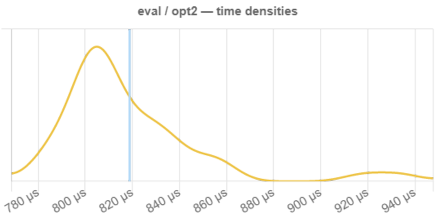


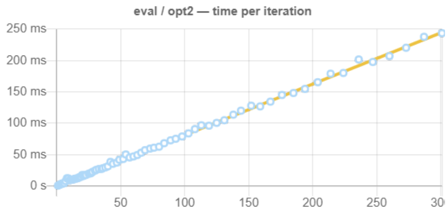


	lower bound	**estimate**	upper bound
OLS regression	6.13 ms	6.30 ms	6.42 ms
R² goodness-of-fit	0.994	0.997	0.999
Mean execution time	6.29 ms	6.59 ms	7.77 ms
Standard deviation	147 μs	1.69 ms	3.55 ms

Outlying measurements have a severe (91.4%) effect on estimated standard deviation.

eval / opt2

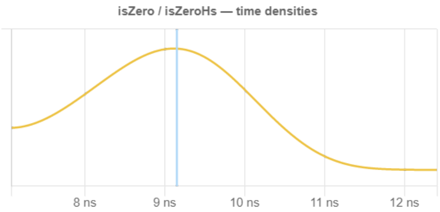


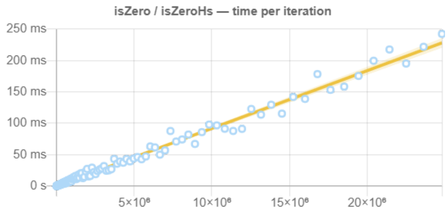


	lower bound	**estimate**	upper bound
OLS regression	804 μs	813 μs	823 μs
R² goodness-of-fit	0.998	0.999	0.999
Mean execution time	812 μs	819 μs	831 μs
Standard deviation	18.8 μs	30.4 μs	46.6 μs

Outlying measurements have a moderate (28.0%) effect on estimated standard deviation.

isZero / isZeroHs

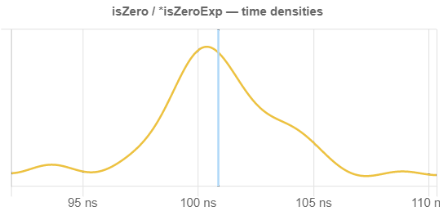


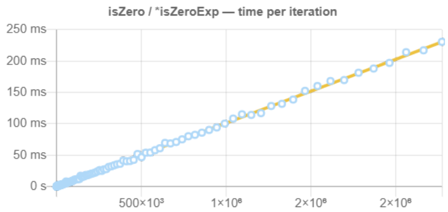


	lower bound	**estimate**	upper bound
OLS regression	8.87 ns	9.19 ns	9.49 ns
R² goodness-of-fit	0.986	0.990	0.994
Mean execution time	8.84 ns	9.15 ns	9.49 ns
Standard deviation	808 ps	1.02 ns	1.39 ns

Outlying measurements have a severe (93.8%) effect on estimated standard deviation.

isZero / *isZeroExp

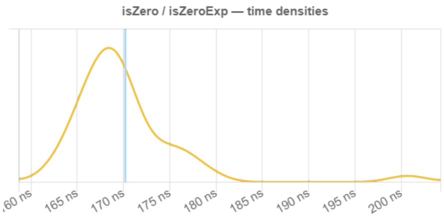


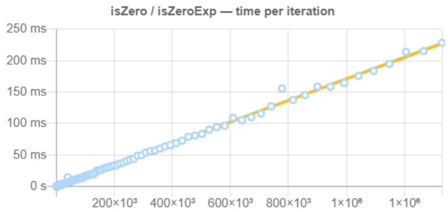


	lower bound	**estimate**	upper bound
OLS regression	101 ns	101 ns	102 ns
R^2 goodness-of-fit	0.999	0.999	1.00
Mean execution time	99.9 ns	101 ns	102 ns
Standard deviation	2.57 ns	3.29 ns	4.36 ns

Outlying measurements have a severe (50.0%) effect on estimated standard deviation.

isZero / isZeroExp

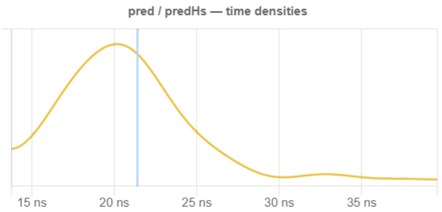


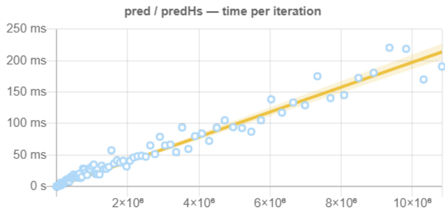


	lower bound	**estimate**	upper bound
OLS regression	169 ns	171 ns	174 ns
R^2 goodness-of-fit	0.996	0.998	1.00
Mean execution time	169 ns	170 ns	173 ns
Standard deviation	3.65 ns	6.08 ns	11.6 ns

Outlying measurements have a severe (53.9%) effect on estimated standard deviation.

pred / predHs

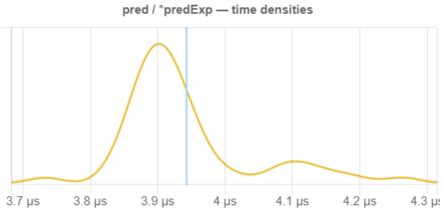


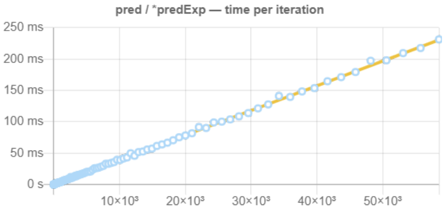


	lower bound	**estimate**	upper bound
OLS regression	18.6 ns	19.7 ns	21.0 ns
R^2 goodness-of-fit	0.969	0.976	0.984
Mean execution time	20.3 ns	21.4 ns	22.9 ns
Standard deviation	3.42 ns	4.51 ns	6.18 ns

Outlying measurements have a severe (98.2%) effect on estimated standard deviation.

pred / *predExp

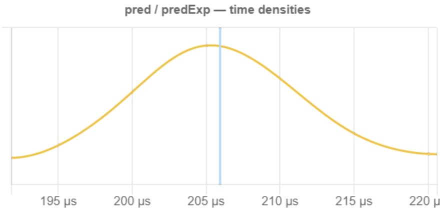


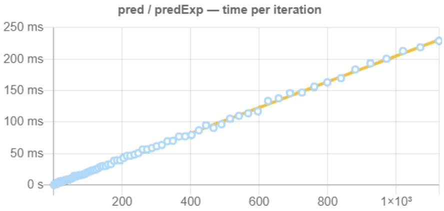


	lower bound	**estimate**	upper bound
OLS regression	3.91 μs	3.93 μs	3.97 μs
R² goodness-of-fit	0.999	1.00	1.00
Mean execution time	3.92 μs	3.94 μs	3.99 μs
Standard deviation	79.0 ns	102 ns	136 ns

Outlying measurements have a moderate (30.9%) effect on estimated standard deviation.

pred / predExp

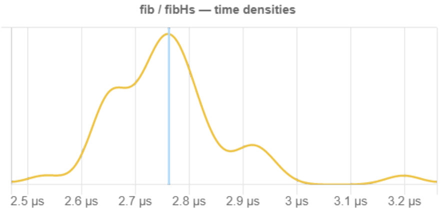


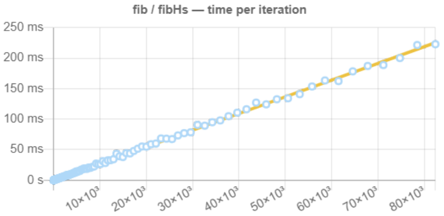


	lower bound	**estimate**	upper bound
OLS regression	203 μs	205 μs	207 μs
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	204 μs	206 μs	208 μs
Standard deviation	4.72 μs	5.65 μs	7.23 μs

Outlying measurements have a moderate (22.4%) effect on estimated standard deviation.

fib / fibHs

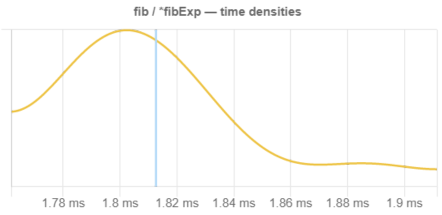


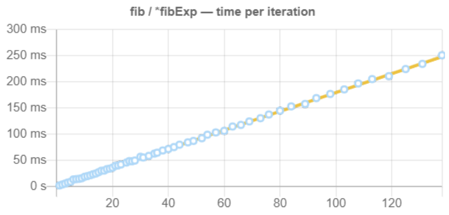


	lower bound	**estimate**	upper bound
OLS regression	2.71 μs	2.73 μs	2.77 μs
R² goodness-of-fit	0.999	0.999	0.999
Mean execution time	2.73 μs	2.76 μs	2.80 μs
Standard deviation	84.3 ns	112 ns	177 ns

Outlying measurements have a severe (53.7%) effect on estimated standard deviation.

fib / *fibExp

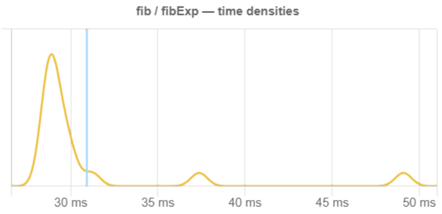

fib / *fibExp — time densities

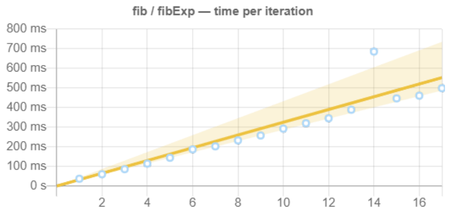

fib / *fibExp — time per iteration

	lower bound	**estimate**	upper bound
OLS regression	1.79 ms	1.80 ms	1.81 ms
R² goodness-of-fit	1.00	1.00	1.00
Mean execution time	1.80 ms	1.81 ms	1.82 ms
Standard deviation	22.5 μs	30.3 μs	38.4 μs

Outlying measurements have a slight (6.35%) effect on estimated standard deviation.

fib / fibExp

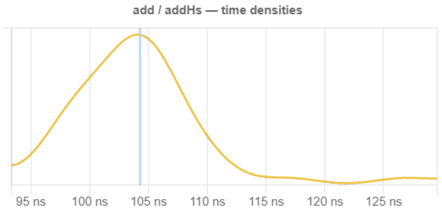

fib / fibExp — time densities

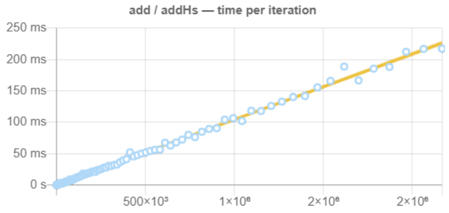

fib / fibExp — time per iteration

	lower bound	**estimate**	upper bound
OLS regression	28.7 ms	32.5 ms	43.2 ms
R² goodness-of-fit	0.768	0.865	0.999
Mean execution time	29.4 ms	30.9 ms	34.9 ms
Standard deviation	798 μs	5.09 ms	9.00 ms

Outlying measurements have a severe (63.3%) effect on estimated standard deviation.

add / addHs

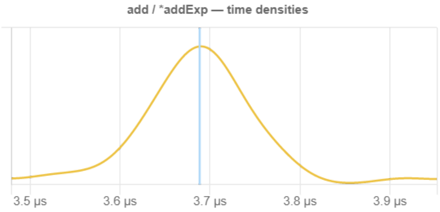

add / addHs — time densities

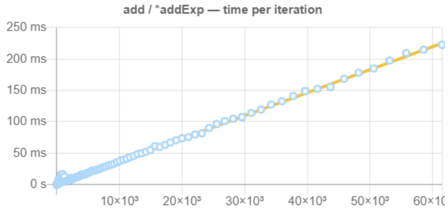

add / addHs — time per iteration

	lower bound	**estimate**	upper bound
OLS regression	103 ns	104 ns	107 ns
R² goodness-of-fit	0.996	0.998	0.999
Mean execution time	103 ns	104 ns	106 ns
Standard deviation	3.71 ns	5.39 ns	7.96 ns

Outlying measurements have a severe (71.9%) effect on estimated standard deviation.

add / *addExp

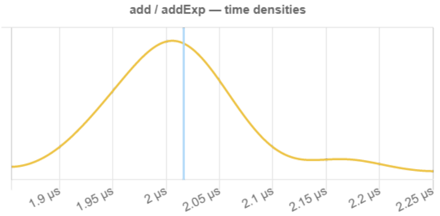


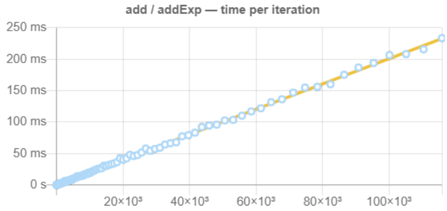


	lower bound	**estimate**	upper bound
OLS regression	3.64 μs	3.66 μs	3.69 μs
R² goodness-of-fit	0.997	0.998	0.999
Mean execution time	3.67 μs	3.69 μs	3.71 μs
Standard deviation	48.4 ns	64.1 ns	95.4 ns

Outlying measurements have a moderate (17.0%) effect on estimated standard deviation.

add / addExp

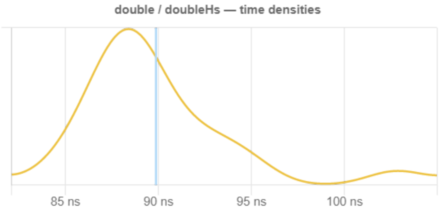


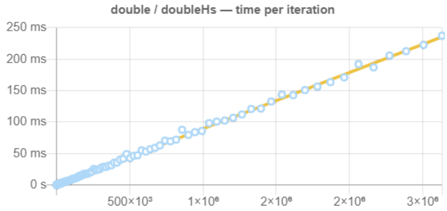


	lower bound	**estimate**	upper bound
OLS regression	1.99 μs	2.01 μs	2.03 μs
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	2.00 μs	2.02 μs	2.04 μs
Standard deviation	54.4 ns	68.8 ns	90.6 ns

Outlying measurements have a moderate (45.7%) effect on estimated standard deviation.

double / doubleHs

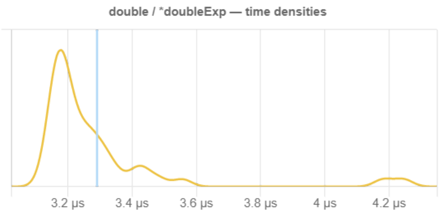


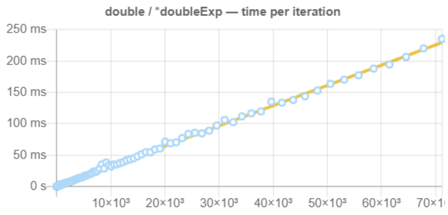


	lower bound	**estimate**	upper bound
OLS regression	88.5 ns	89.4 ns	90.3 ns
R² goodness-of-fit	0.998	0.999	1.00
Mean execution time	88.8 ns	89.9 ns	91.2 ns
Standard deviation	2.59 ns	3.91 ns	5.66 ns

Outlying measurements have a severe (64.9%) effect on estimated standard deviation.

double / *doubleExp

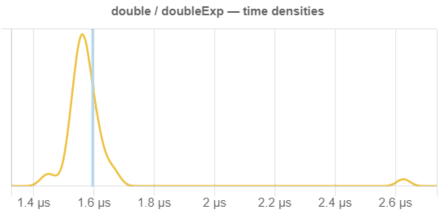


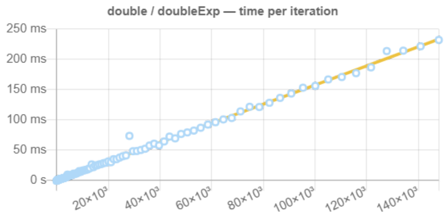


	lower bound	**estimate**	upper bound
OLS regression	3.20 μs	3.22 μs	3.26 μs
R² goodness-of-fit	0.998	0.999	0.999
Mean execution time	3.25 μs	3.29 μs	3.41 μs
Standard deviation	98.6 ns	224 ns	365 ns

Outlying measurements have a severe (76.5%) effect on estimated standard deviation.

double / doubleExp

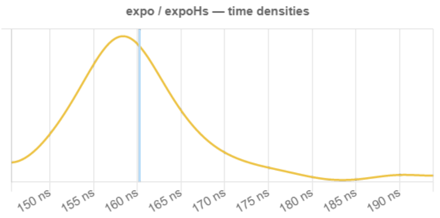


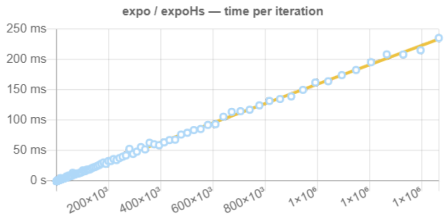


	lower bound	**estimate**	upper bound
OLS regression	1.56 μs	1.58 μs	1.60 μs
R² goodness-of-fit	0.993	0.997	1.00
Mean execution time	1.57 μs	1.59 μs	1.70 μs
Standard deviation	41.2 ns	168 ns	351 ns

Outlying measurements have a severe (89.5%) effect on estimated standard deviation.

expo / expoHs

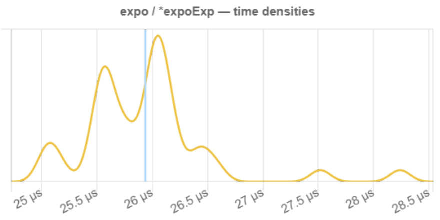


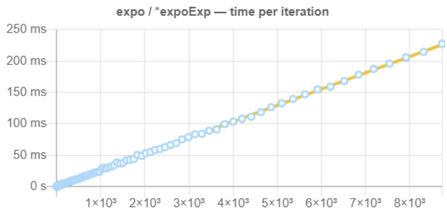


	lower bound	**estimate**	upper bound
OLS regression	158 ns	159 ns	161 ns
R² goodness-of-fit	0.998	0.999	0.999
Mean execution time	158 ns	160 ns	163 ns
Standard deviation	5.27 ns	7.42 ns	11.3 ns

Outlying measurements have a severe (66.5%) effect on estimated standard deviation.

expo / *expoExp

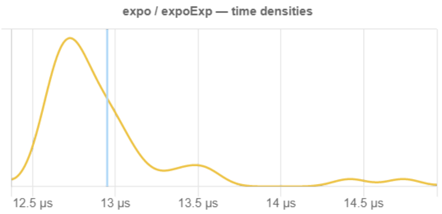


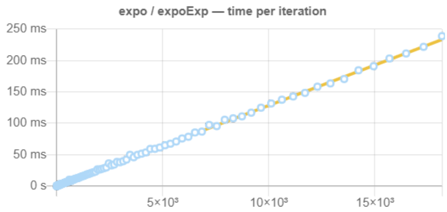


	lower bound	**estimate**	upper bound
OLS regression	25.8 μs	25.9 μs	26.0 μs
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	25.8 μs	25.9 μs	26.1 μs
Standard deviation	406 ns	579 ns	984 ns

Outlying measurements have a moderate (20.5%) effect on estimated standard deviation.

expo / expoExp

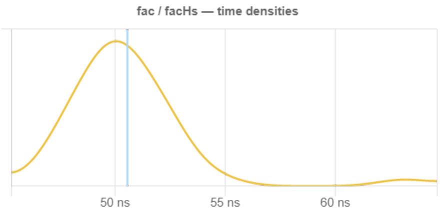


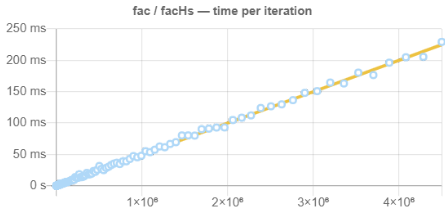


	lower bound	**estimate**	upper bound
OLS regression	12.8 μs	12.9 μs	12.9 μs
R^2 goodness-of-fit	0.999	1.00	1.00
Mean execution time	12.9 μs	13.0 μs	13.1 μs
Standard deviation	258 ns	442 ns	722 ns

Outlying measurements have a moderate (40.6%) effect on estimated standard deviation.

fac / facHs

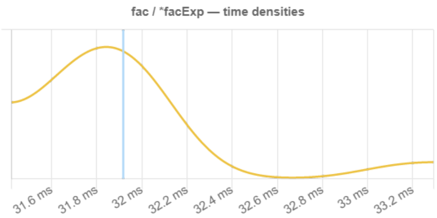


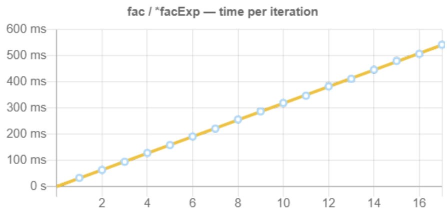


	lower bound	**estimate**	upper bound
OLS regression	49.2 ns	49.9 ns	50.5 ns
R^2 goodness-of-fit	0.998	0.999	0.999
Mean execution time	50.0 ns	50.6 ns	51.5 ns
Standard deviation	1.65 ns	2.60 ns	4.36 ns

Outlying measurements have a severe (73.1%) effect on estimated standard deviation.

fac / *facExp

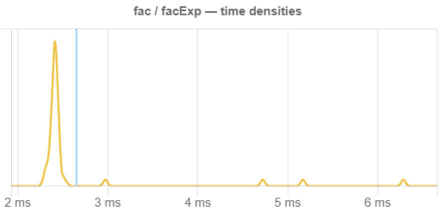

fac / *facExp — time densities

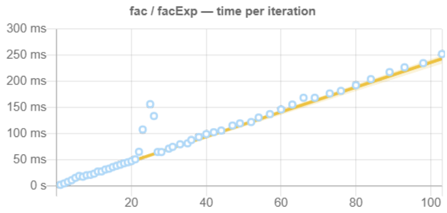

fac / *facExp — time per iteration

	lower bound	estimate	upper bound
OLS regression	31.7 ms	31.8 ms	32.0 ms
R² goodness-of-fit	1.00	1.00	1.00
Mean execution time	31.8 ms	31.9 ms	32.2 ms
Standard deviation	157 μs	355 μs	652 μs

Outlying measurements have a slight (5.54%) effect on estimated standard deviation.

fac / facExp

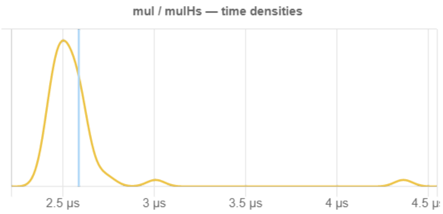

fac / facExp — time densities

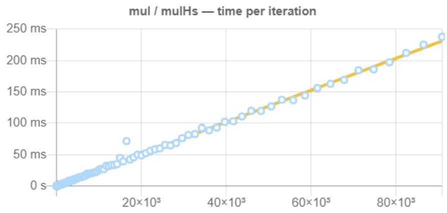

fac / facExp — time per iteration

	lower bound	estimate	upper bound
OLS regression	2.27 ms	2.36 ms	2.43 ms
R² goodness-of-fit	0.846	0.935	0.998
Mean execution time	2.48 ms	2.65 ms	3.07 ms
Standard deviation	426 μs	800 μs	1.34 ms

Outlying measurements have a severe (95.9%) effect on estimated standard deviation.

mul / mulHs

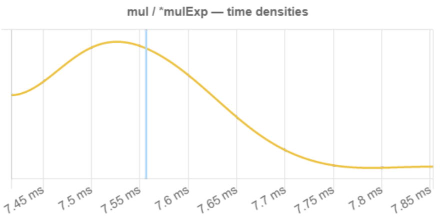

mul / mulHs — time densities

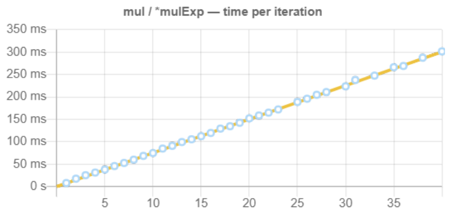

mul / mulHs — time per iteration

	lower bound	estimate	upper bound
OLS regression	2.52 μs	2.55 μs	2.57 μs
R² goodness-of-fit	0.992	0.997	0.999
Mean execution time	2.53 μs	2.59 μs	2.77 μs
Standard deviation	84.2 ns	294 ns	598 ns

Outlying measurements have a severe (90.6%) effect on estimated standard deviation.

mul / *mulExp

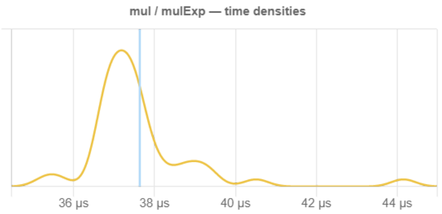


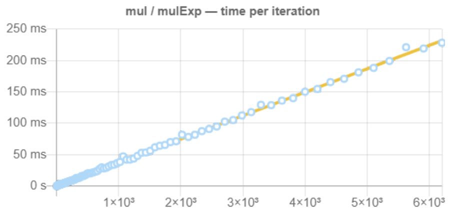


	lower bound	**estimate**	upper bound
OLS regression	7.46 ms	7.51 ms	7.57 ms
R² goodness-of-fit	1.00	1.00	1.00
Mean execution time	7.53 ms	7.56 ms	7.59 ms
Standard deviation	58.4 μs	79.7 μs	118 μs

Outlying measurements have a slight (2.85%) effect on estimated standard deviation.

mul / mulExp

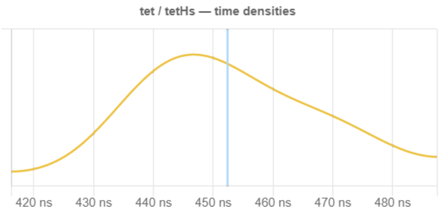


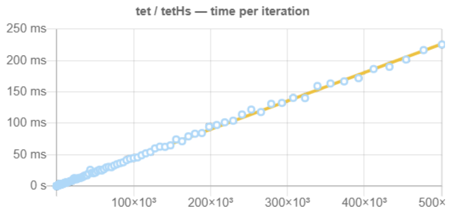


	lower bound	**estimate**	upper bound
OLS regression	37.0 μs	37.4 μs	37.8 μs
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	37.3 μs	37.6 μs	38.3 μs
Standard deviation	878 ns	1.39 μs	2.29 μs

Outlying measurements have a moderate (40.8%) effect on estimated standard deviation.

tet / tetHs

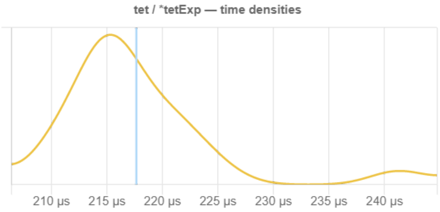


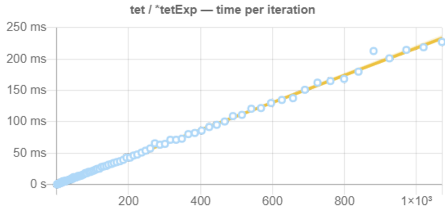


	lower bound	**estimate**	upper bound
OLS regression	448 ns	452 ns	457 ns
R² goodness-of-fit	0.999	0.999	0.999
Mean execution time	448 ns	452 ns	457 ns
Standard deviation	11.9 ns	13.8 ns	16.9 ns

Outlying measurements have a moderate (43.4%) effect on estimated standard deviation.

tet / *tetExp

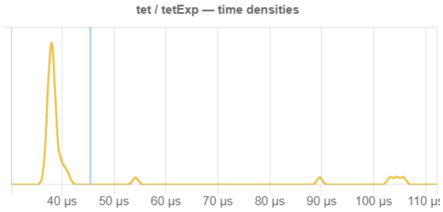


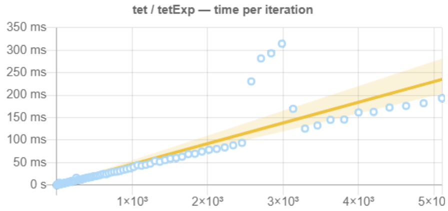


	lower bound	**estimate**	upper bound
OLS regression	214 µs	218 µs	223 µs
R² goodness-of-fit	0.996	0.998	1.00
Mean execution time	216 µs	218 µs	220 µs
Standard deviation	4.46 µs	6.58 µs	10.4 µs

Outlying measurements have a moderate (25.5%) effect on estimated standard deviation.

tet / tetExp

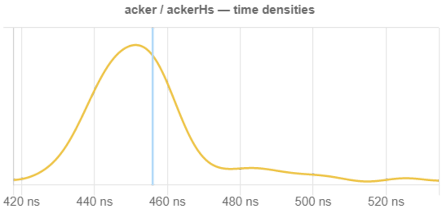


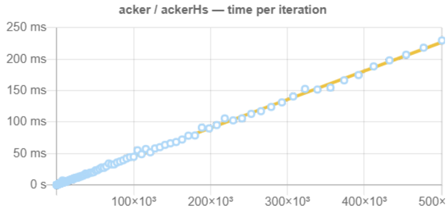


	lower bound	**estimate**	upper bound
OLS regression	39.8 µs	46.0 µs	55.0 µs
R² goodness-of-fit	0.735	0.790	0.943
Mean execution time	40.9 µs	45.5 µs	54.0 µs
Standard deviation	11.1 µs	19.4 µs	26.8 µs

Outlying measurements have a severe (99.2%) effect on estimated standard deviation.

acker / ackerHs

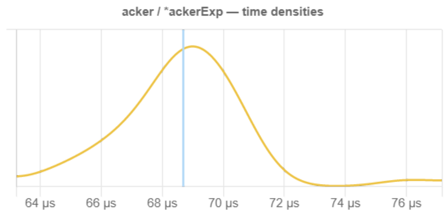


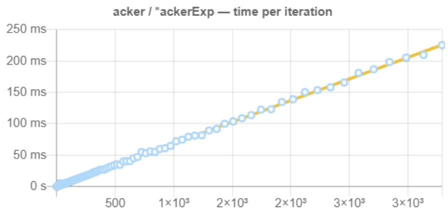


	lower bound	**estimate**	upper bound
OLS regression	449 ns	453 ns	457 ns
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	452 ns	456 ns	462 ns
Standard deviation	13.9 ns	18.7 ns	27.3 ns

Outlying measurements have a severe (58.4%) effect on estimated standard deviation.

acker / *ackerExp

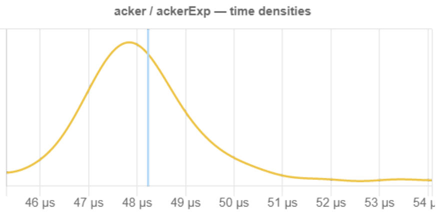


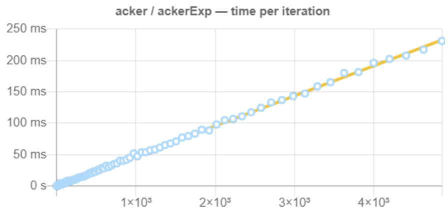


	lower bound	**estimate**	upper bound
OLS regression	68.0 μs	68.6 μs	69.2 μs
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	68.2 μs	68.7 μs	69.4 μs
Standard deviation	1.45 μs	1.96 μs	2.87 μs

Outlying measurements have a moderate (27.0%) effect on estimated standard deviation.

acker / ackerExp

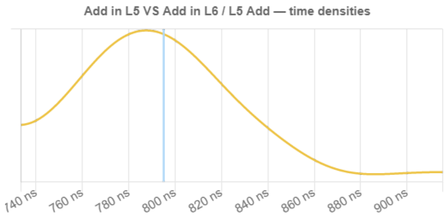


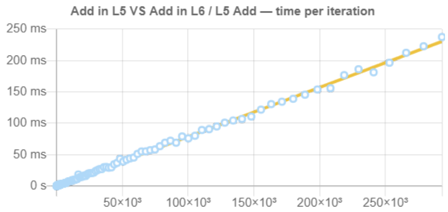


	lower bound	**estimate**	upper bound
OLS regression	47.5 μs	47.9 μs	48.3 μs
R² goodness-of-fit	0.999	0.999	1.00
Mean execution time	47.9 μs	48.2 μs	48.8 μs
Standard deviation	1.04 μs	1.38 μs	2.19 μs

Outlying measurements have a moderate (28.3%) effect on estimated standard deviation.

Add in L5 VS Add in L6 / L5 Add

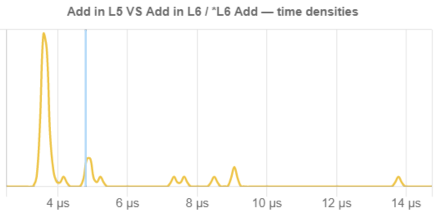


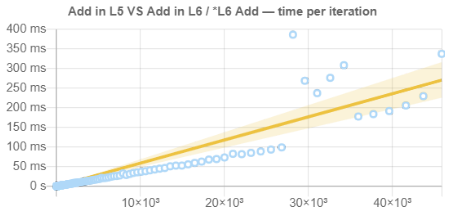


	lower bound	**estimate**	upper bound
OLS regression	776 ns	787 ns	795 ns
R² goodness-of-fit	0.998	0.999	0.999
Mean execution time	786 ns	795 ns	805 ns
Standard deviation	24.6 ns	31.0 ns	43.3 ns

Outlying measurements have a severe (55.0%) effect on estimated standard deviation.

Add in L5 VS Add in L6 / *L6 Add

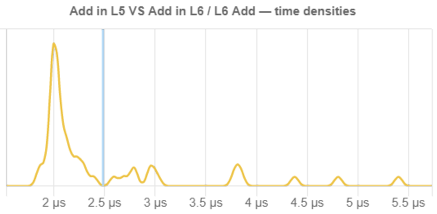

Add in L5 VS Add in L6 / *L6 Add — time densities

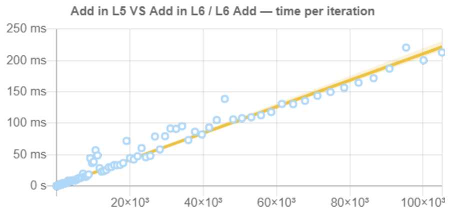

Add in L5 VS Add in L6 / *L6 Add — time per iteration

	lower bound	**estimate**	upper bound
OLS regression	4.92 μs	5.90 μs	6.90 μs
R^2 goodness-of-fit	0.748	0.830	0.915
Mean execution time	4.29 μs	4.80 μs	5.77 μs
Standard deviation	1.47 μs	2.24 μs	3.56 μs

Outlying measurements have a severe (99.4%) effect on estimated standard deviation.

Add in L5 VS Add in L6 / L6 Add


Add in L5 VS Add in L6 / L6 Add — time densities


Add in L5 VS Add in L6 / L6 Add — time per iteration

	lower bound	**estimate**	upper bound
OLS regression	2.05 μs	2.11 μs	2.19 μs
R^2 goodness-of-fit	0.962	0.977	0.988
Mean execution time	2.30 μs	2.48 μs	2.82 μs
Standard deviation	575 ns	811 ns	1.16 μs

Outlying measurements have a severe (98.9%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x*-axis indicates the number of loop iterations, while the *y*-axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression estimate of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line. The transparent area behind it shows the confidence interval for the execution time estimate.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R^2; goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R^2; should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be.

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.