Article title: A Type Theory of Leelus Type System in C++ Programming Language.

Authors: Frank Appiah[1]

Affiliations: Department of Informatics[1]

Orcid ids: 0000-0001-6732-9503[1]

Contact e-mail: appiahnsiahfrank@gmail.com

License information: This work has been published open access under Creative Commons Attribution License <http://creativecommons.org/licenses/by/4.0/>, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Conditions, terms of use and publishing policy can be found at <https://www.scienceopen.com/>.

Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.

DOI: 10.14293/S2199-1006.1.SOR-.PPZQLT1.v1

Preprint first posted online: 08 August 2021

Keywords: Type theory , Type system , Assignment , Judgement , Environment , Leelus

A Type Theory of Leelus Type System in C++ Programming Language.

Frank Appiah ¹

Abstract. This research is on type theory, which describes type, term and value in a type system programmed in C++. Type theory is closely related to, and in some cases overlaps with, computational type systems, which are a programming language feature used to reduce bugs.

Keywords. *term, type, type theory, value, definitions, type systems, environment, assignments, denotation, judgements.*

1 Introduction

Between 1902 and 1908 Bertrand Russell proposed various "theories of type" in response to his discovery that Gottlob Frege's version of naive set theory was afflicted with Russell's paradox. By 1908 Russell arrived at a "ramified" theory of types together with an "axiom of reducibility" both of which featured prominently in Whitehead and Russell's Principia Mathematica published between 1910 and 1913. They attempted to resolve Russell's paradox by first creating a hierarchy of types, then assigning each concrete mathematical (and possibly other) entity to a type. Entities of a given type are built exclusively from entities of those types that are lower in their hierarchy, thus preventing an entity from being assigned to itself.

In the 1920s, Leon Chwistek and Frank P. Ramsey proposed an unramified type theory, now known as the "theory of simple types" or simple type theory, which collapsed the hierarchy of the types in the earlier ramified theory and as such did not require the axiom of reducibility.

¹ Frank Appiah is with King's College London, Strand, London, England, UK. Department of Informatics.

The common usage of "type theory" is when those types are used with a term rewrite system. The most famous early example is Alonzo Church's simply typed lambda calculus. Church's theory of types[1] helped the formal system avoid the Kleene–Rosser paradox that afflicted the original untyped lambda calculus. Church demonstrated that it could serve as a foundation of mathematics and it was referred to as a higher-order logic.

Some other type theories include Per Martin-Löf's intuitionistic type theory, which has been the foundation used in some areas of constructive mathematics. Thierry Coquand's calculus of constructions and its derivatives are the foundation used by Coq, Lean, and others. The field is an area of active research, as demonstrated by homotopy type theory.

In mathematics, logic, and computer science, a type system is a formal system in which every term has a "type" which defines its meaning and the operations that may be performed on it. Type theory is the academic study of type systems.

Typing usually takes place in some context or environment denoted by the symbol, Γ . Often, an environment is a list of pairs $e:T$. This pair is sometimes called an assignment. The context completes the above opposition. Together they form a judgement denoted $\Gamma \vdash e:T$. Type theories have explicit computation and are encoded in rules for rewriting terms. These are called conversion rules or, if the rule only works in one direction, a reduction rule. In programming languages, a type system is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a computer program, such as variables, expressions, functions or modules.[1] These types formalize and enforce the otherwise implicit categories the programmer uses for algebraic data types, data structures, or other components (e.g. "string", "array of float", "function returning boolean"). The main purpose of a type system is to reduce possibilities for bugs in computer programs[2] by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen statically (at compile time), dynamically (at run time), or as a combination of both. Type systems have other purposes as well, such as expressing business rules, enabling certain compiler optimizations, allowing for multiple dispatch, providing a form of documentation, etc.[10,11]

Leelus[6] is a typed language theory for interpretable artificial intelligence method for multi-agent perspective. This is based on the Object-Oriented C++[1,2,3,4,5] language. In describing, readers will be looking at type system constructs as type, term and value. There are only three variables in the language argumentation. The argumentatives includes action,

location and temporal with a variant type still including three arguments- rank, interest 1 and interest 2.

A type definition is made with the declaration:[6]

typedef datatype var_name;

In Leelus type language, each data type is declared as a character pointer to represent an array of characters. The type definitions are as follows in Leelus:

typedef char* action;

typedef char* temporal;

typedef char* interest;

typedef char* location;

typedef char* rank;

The action, location, rank and two interests are of character type dimensioned in array. It is programmed as a pointer definition. I will now described programmed actions with four values in type system:

```
action a="buy", a1="sell", a2="sold", a3="trade";
```

The type system is in C++ declaration. I will now use type theory to study the system. We will be looking at type, term and value in tabular form. I will also describe the typing environment and judgement denotation.

Type System	Type	Term	Value
action a="buy" ;	action	a	buy
action a1="sell" ;	action	a1	sell
action a2="sold";	action	a2	sold
action a3="trade";	action	a3	trade

The first programmed action declared as **a** is an equalisation of "buy" action. This action is both a business and economics term. A buy action is used to describe a process of buying a good or service or both (good and service). The second programmed action is declared as **a1** and it is an equalisation of "sell" action. Sell action is used to describe a process of selling

a good or service or both. The third programmed action value is sold and it is declared as **a2**. It is used to indicate whether a good or service is sold or not.

Typing **action** type system gives an environment listed as a pair:

- *a: action*
- *a1: action*
- *a2: action*
- *a3: action*

These listed pairs are assignments in the action type system:

- *a: buy*
- *a1: sell*
- *a2: sold*
- *a3: trade*

The denotation of judgements are as listed:

- $\Gamma \vdash a: \text{buy}$
- $\Gamma \vdash a1: \text{sell}$
- $\Gamma \vdash a2: \text{sold}$
- $\Gamma \vdash a3: \text{trade}$

The next set of programmed interests are described in terms of their value in the program.

The code for declaration is as follows :

```
interest ia="sell_pc", ia1="buy_pc";
```

Type System	Type	Term	Value
interest ia="sell_pc";	interest	ia	sell_pc
interest ia="buy_pc";	interest	ia1	biy_pc

Two values of interest are used in the program namely : sell_pc or buy_pc.

This means that an agent has value if the entity sells pc or either buys pc. Buy is the anti-sell and vice versa.

On the business side, interest is a return value on investment denoted by a principal value period in some year(s) at a certain percentage limited to 100%. Again in such ways, what is

the interest of an agent in business? This is a means to evaluate the interest of an action to another.

Typing **interest** type system gives an environment listed as a pair:

- *ia: interest*
- *ia1: interest*

These listed pairs are assignments in the **interest** type system:

- *ia: sell_pc*
- *ia1: buy_pc*

The denotation of judgements are as listed:

- |- *ia: sell_pc*
- |- *ia1: buy_pc*

Businesses cannot stay relevant at all times. It is important to rank them to store the value of their rank in such a process. The value of rank for an agent to stay relevant to an organisation interested in delivering always or sometimes quality of service is required. Quality of service is required in doing business at the highest level and as such it is catered for by the rank value. The declaration of rank is coded as follows :

```
rank r="rank_9", rr="rank_7";
```

Type System	Type	Term	Value
rank r="rank_9";	rank	r	rank_9
rank rr="rank_7";	rank	rr	rank_7

A single rank is declared in this program. It is declared as r. The rank is a numeric value but here the number is used after an underscore. A business which scores 9 is better off than another which scores 12. Hence, a lower rank value is better off in terms of doing quality of service than a higher rank entity.

Typing **rank** type system gives an environment listed as a pair:

- *r: rank*
- *rr: rank*

These listed pairs are assignments in the **rank** type system:

- $r: rank_9$
- $rr: rank_7$

The denotation of judgements are as listed:

- $\Gamma|- r: rank_9$
- $\Gamma|- rr: rsnk_7$

We are looking at two locations in this coded program. It is programmed that declared loc is valued at Tottenham for an agent to do business. This is the first location value.

The code of location in stored program is as follows :

```
location loc="tottenham", loc1="London";
```

Type System	Type	Term	Value
location loc="Tottenham" ;	location	loc	Tottenham
location loc="London" ;	location	loc1	London

The second declared term is loc1. It has a value of "London". This again situated action is located in London locality only except loc value which is Tottenham. Agent parties now do businesses in Tottenham and London only. But other local businesses can be located elsewhere. Possibly Accra Biz does business with New York Biz. Again, Kumasi Biz does business with London Biz and so on.

Typing **location** type system gives an environment listed as a pair:

- loc: location
- loc1: location

These listed pairs are assignments in the **location** type system:

- loc: Tottenham
- loc1: London

The denotation of judgements are as listed:

- $\Gamma|- \text{loc: Tottenham}$
- $\Gamma|- \text{loc1: London}$

Doing business cannot be a permanent activity. Holding to do business with an agent in an agency is temporary in some sense. After a certain period of time limits an organisation in doing business with the agent company. That requirement demands a stored temporal value to hold business going until otherwise. On elapse, business stops to other matters until further agreement is signed. Contract by agreement between agent and entity is required to complete in a closed location before opening another business day. The expected commissions necessary on demand by both parties is of importance. The declared variable in temporal dimension is temp and it is coded as:

```
temporal temp="Sat 12/09/2019 12:00pm";
```

Type System	Type	Term	Value
temporal temp="Sat 12/09/2019 12:00pm";	temporal	temp	Sat 12/09/2019 12:00pm

The temp value is Saturday 12/09/2019 12:00pm. The agent can begin to do a buying action or selling action before or after the declared day of doing business. Entity might cease to work if the declared temporal value is elapsed. Again might start to do business with agents after the elapsed. Always or sometimes or never calculi on temporal value might be a requirement on the specification of logic programs.

Typing **temporal** type system gives an environment listed as a pair:

- temp: temporal

These listed pairs are assignments in the **temporal** type system:

- temp: Sat 12/09/2019 12:00pm

The denotation of judgements are as listed:

- $\Gamma|- \text{temp: Sat 12/09/2019 12:00pm}$

2 Conclusion

In this work, I describe a brief history of type theory in studying type system. Leelus type system in c++ programming language is studied in researching the type environment of a Leelus Program. Five type systems are looked at on terminologies of type, term and value. These are : action, location, interest, temporal and rank. Environments, Assignments and denotation of judgements of type systems are also looked at in much detail.

Further Reading

1. Josuttis, N. M. (2012). The C++ standard library: a tutorial and reference.
2. Coplien, J. O. (1991). Advanced C++ programming styles and idioms. Addison-Wesley Longman Publishing Co., Inc..
3. Stroustrup, B. (2000). The C++ programming language. Pearson Education India.
4. Horowitz, E., Sahni, S., & Rajasekaran, S. (1997). Computer algorithms C++: C++ and pseudocode versions. Macmillan.
5. Eckel, B. (2021). Thinking in C++.
6. Frank Appiah (2021). An Object-Oriented Language Based on C++ :Leelus Typed Language. EasyChair Preprint no. 6189
7. Frank Appiah (2021). The Mapping of the Logical Structure, Enact to the Engagement Structure of Enact. EasyChair Preprint no. 6187
8. Frank Appiah (2021). Semantic Computation of the Propositional Model Composites in Enactment Logic. EasyChair Preprint no. 6186
9. Frank Appiah (2021). Leemapper accessed online at <https://github.com/fanhubgt/Leemapper>.
10. Mitchell, John C.; Plotkin, Gordon D. (July 1988). "Abstract Types Have Existential Type" (PDF). ACM Trans. Program. Lang. Syst. 10 (3): 470–502. doi:10.1145/44501.45065.
11. Barendregt, Henk; Dekkers, Wil; Statman, Richard (20 June 2013). Lambda Calculus with Types. Cambridge University Press. p. 66. ISBN 978-0-521-76614-2.