Linear λ-calculus, extensions and relation to substructural logics

Ana Jorge Almeida

Supervisor: Mário Florido
Co-supervisor: Sandra Alves

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

July, 2023

Abstract

There are three structural properties satisfied by the simply typed λ-calculus that can be used to constrain interfaces that provide access to system resources. These properties are *exchange*, *weakening* and *contraction*, each determining the amount of times and order in which variables can be written down in the context. Linear type systems use only the *exchange* rule, ensuring that every variable in the context is used exactly once.

In this project, we implemented a type checker and operational semantics for a language that uses both linear and unrestricted data structures, and we also implemented a parser, which produces the corresponding abstract syntax tree.

1 Introduction

Abstract types can be used to restrict access to data structures and to limit the use of newly defined operations, but they are not sufficient to limit the ordering and number of uses of functions in an interface [2]. Substructural type systems augment standard type abstraction mechanisms with the ability to control the number and order of uses of a data structure or operation [2]. They are particularly useful for constraining interfaces that provide access to system resources such as files, locks and memory [2] and they also provide sound static mechanisms for keeping track of state changes and preventing operations on objects in an invalid state [2].

The objective of this project is to implement a linear type system with extensions that accommodates both unrestricted and linear data structures, by associating an explicit qualifier to each data structure, and controls their usage, deallocating a linear data structure immediately after being used [2].

This report is organised in 3 chapters. **Development**, chapter 2, where we provide an overview of all relevant concepts covered, and **Conclusion**, chapter 3, where we summarise what was done and discuss the obstacles we encountered.

2 Development

A standard simply-typed λ-calculus rule is defined as $\Gamma \vdash M : \tau$, where Γ is a type environment, which is a sequence of variables and their types, M is a λ-term and τ its corresponding type [1]. This means "The closed term M has type τ under the Γ set of assumptions" [1].

2.1 Structural properties

The simply typed λ-calculus satisfies three basic structural properties: *exchange*, which indicates that the order in which we write down variables in the context is irrelevant, *weakening*, which indicates that adding extra, unneeded assumptions to the context does not prevent a term from type checking, and *contraction*, which states that if we can type check a term using two identical assumptions then we can check the same term using a single assumption [2]. These properties are defined below.

Lemma (Exchange)**.** *If* $\Gamma_1, x_1 : T_1, x_2 : T_2, \Gamma_2 \vdash M : T$ *then*
$\Gamma_1, x_2 : T_2, x_1 : T_1, \Gamma_2 \vdash M : T$

Lemma (Weakening)**.** *If* $\Gamma_1, \Gamma_2 \vdash M : T$ *then* $\Gamma_1, x : T_1, \Gamma_2 \vdash M : T$

Lemma (Contraction)**.** *If* $\Gamma_1, x_2 : T_1, x_3 : T_1, \Gamma_2 \vdash M : T_2$ *then*
$\Gamma_1, x_1 : T_1, \Gamma_2 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]M : T_2$

Linear type systems ensure that every variable is used exactly once [2], following only the *exchange* rule and discarding both *weakening* and *contraction* [2].

2.2 Linear type system

In order to ensure that objects are used exactly once, we need to know that the data we deallocate is never used in the future [2]. Therefore, the system we have implemented uses type qualifiers q that annotate the introduction forms of all data structures [2], which can be linear or unrestricted. The linear qualifier, *lin*, indicates that the data structure in question will be used exactly once in the program [2], meaning that it will have to be eliminated immediately after its use. On the other hand, the unrestricted qualifier, *un*, indicates that the data structure behaves as in the standard simply-typed λ-calculus [2], meaning it can be used as many times as desired [2].

We also need to define a relation $q \sqsubseteq q'$, which is reflexive, transitive and $lin \sqsubseteq un$ [2]. This states that linear data structures can hold objects with linear or unrestricted type, but unrestricted data structures can only hold objects with unrestricted type [2]. It is also imported to note that base cases must ensure that no linear variable is discarded without being used [2], since that would undoubtedly compromise the properties of the linear type system.

Here, our type checker is defined for both *unrestricted* and *linear variables*, *boolean values*, *if statements*, *pairs*, a *split* operation, *abstractions* and *applications*.

2.2.1 Extensions

We extend the syntax for the linear λ-calculus with the standard introduction and elimination forms for sums and recursive types [2].

Values with sum type are introduced by left and right injections [2] with a qualifier and an associated type without a qualifier. There is also a case expression that will execute its first branch if its primary argument is a left injection and its second branch if its primary argument is a right injection [2].

Recursive types are introduced with roll expressions [2], with a recursive type, and unroll expressions. Recursive data types have no data of their own and therefore, do not need a separate qualifier to control their allocation behaviour [2], so these expressions take on the qualifier of the underlying term [2]. In order to write computations that process recursive types, we add recursive function declarations to our language as well [2]. We cannot allow the closure to contain linear variables since free variables in a recursive function closure will be used on each recursive invocation of the function [2], therefore, all recursive functions are unrestricted data structures.

2.2.2 Operational Semantics

To make the memory management of the language clear, we evaluate terms in an abstract machine with an explicit store [2], which is a sequence of variable-value pairs [2], and any variable appears at most once on the left-hand side of a pair [2]. A value is a pair of a qualifier together with some data [2], which in our standard linear type system are *booleans*, simplified *pairs* and λ-*terms*.

In our extended language, dealing with sum types and recursive types, we also add left and right injections and roll expressions to our value list [1].

3 Conclusion

In this project, we studied and implemented a linear type system with extensions to deal with sum types and recursive types. The implementation can be seen here, `https://github.com/anathegrey/linear-lambda-calculus`.

We encountered some difficulties while studying these extensions because the paper we used [2] lacked consistency in its presented type checker and did not provide any information on these extensions' operational semantics, which made us have to use other resources [1] and adapt its contents to this particular type system, and so we were not able to study the other presented extensions, such as polymorphism and arrays, under the available time frame.

References

[1] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[2] David Walker. Substructural type systems. *Advanced topics in types and programming languages*, pages 3–44, 2005.