RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Insert title here

Insert your name here

Master's Thesis – June 9, 2023
Chair for Network and Data Security.

Supervisor: Name of or first examiner
Advisor: Name(s) of your other advisers

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

How can I improve my scientific writing skills?

- https://www.zfw.rub.de/sz/
- http://www.cs.joensuu.fi/pages/whamalai/sciwri/sciwri.pdf
- https://www.student.unsw.edu.au/writing
- https://www.sydney.edu.au/content/dam/students/documents/learning-resources/learning-centre/writing-a-thesis-proposal.pdf

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Date	Insert your name here

Erklärung

Ich erkläre mich damit einverstanden, dass meine Abschlussarbeit am Lehrstuhl Netz- und Datensicherheit dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen. Weiterhin erkläre ich mich damit einverstanden, dass die Abschlussarbeit auf die Webseite des Lehrstuhls veröffentlicht werden darf.

Date

Insert your name here

Contents

1 Preliminaries

1.1 Amortized Analysis: Potential Method

Analysing the time and space complexity of algorithms is itself a vast field. For AARA, amortized analysis using the potential method is used. To perform amortized analysis, we define a potential function Φ, which assigns to every possible state of a data structure a *non-negative* integer. Using the potential assigned to a specific state, more expensive operations can be amortized by preceding, cheaper operations.

To illustrate the advantage of amortized analysis over worst-case analysis, we illustrate both using sequences of inserts over a *DynamicArray* as an example. When initialising a DynamicArray, we provide the size needed. Subsequent inserts to the DynamicArray will be performed instantaneously if memory is free. Whenever an insert to the array would exceed the memory allocated to it, the array doubles in size. This operation is costly, because we have to allocate the memory and move all previous data into the new memory location. Looking at the worst-case runtime, inserting into a dynamic array has a cost of $\mathcal{O}(n)$. There are two important nuances: (1) Not every insert operation is equally costly and (2) The expensive inserts are rarer.

In order to perform amortized analysis using the potential method, we first need to define a potential function Φ. The amortized cost is subsequently given as the sum of the actual cost of the operation and the difference in potential before and after the operation. Formally, we write $C_{actual}(o)$ to denote the actual cost of some operation o as well as S_{before} and S_{after} for the state of the DynamicArray before and after performing operation o. This yields the following formula for the amortized cost of an operation o:

$$C_{amortized}(o) = C_{actual}(o) + (\Phi(S_{after}) - \Phi(S_{before}))$$

For an arbitrary DynamicArray D of size N, of which n memory cells have been used, we define the potential function $\Phi(D) = 2n - N$. Note that $n \leq N$ and $2n \geq N$, because the DynamicArray is always at least half full due to the resizing strategy explained above. As alluded to earlier, a potential function needs to be non-negative for every possible state passed to it. We can immediately conclude that the above function satisfies that constraint, due to $2n \geq N$ being an invariant. We now examine how different types of insert operations affect the potential function and subsequently the amortized cost. Suppose we insert into a DynamicArray, such that no doubling in size is necessary. The actual cost of the operation is constant. Because no resizing of the DynamicArray is induced, we simply increment n. This yields the following potentials:

$$\Phi(S_{before}) = 2n - N$$

$$\Phi(S_{after}) = 2(n + 1) - N$$

Label/annotate equations?

Hence, $\Phi(S_{after}) - \Phi(S_{before}) = 2$. This yields an amortized cost of $C_{amortized}(o) = C_{actual}(o) + 2$, where we know that $C_{actual}(o)$ is constant. As a result, the amortized cost is again constant.

Let us now assume that we insert one element into a DynamicArray, inducing a doubling in size. This results in the following potentials:

$$\Phi(S_{before}) = 2n - N$$

$$\Phi(S_{after}) = 2(n + 1) - 2N$$

Note that $n + 1 = N$, because the array needed to double in size. The potential therefore simplifies to $\Phi(S_{after}) = 0$. This concludes that our potential function is indeed well-formed, because it will not yield negative values for any valid state. We know that the actual cost of resizing the array is $\mathcal{O}(n)$, pluging this into the formula for amortized cost: $C_{amortized}(o) = \mathcal{O}(n) + (0 - \mathcal{O}(n))$. Yielding constant time again, because the difference in potential allowed us to 'pay' for the cost incurred by reallocating the array.

Generalizing the new won insight, allows us to claim the following: *Any sequence of n insert operations takes $\mathcal{O}(n)$ amortized time.* This follows, as the sum of n constant time operations is $\mathcal{O}(n)$.

The formula for amortized cost 1.1 allows us to provide an upper bound on the actual cost of an operation as well. Since the potential function is required to be non-negative, we get that $\Phi(S) \geq 0$ for some state S. Using this inequality, we can infer the following bound simply by using the new inequality:

$$C_{actual}(o) \leq C_{amortized}(o)$$

Because we inferred that any sequence of n insert operations has $\mathcal{O}(n)$ amortized cost, this inequality shows that any sequence of insert operations also has at most $\mathcal{O}(n)$ actual cost.

AARA will deploy the potential method in order to provide bounds. To this end, we will define properties of the potential function in **??**.

Reference to chapter for potential properties.

1.2 Type System

A type system groups values in a computer program into different *types* and defines the set of operations that are valid on each type. Most programmers were already exposed to types, some of them being: string, integers, floats. Subsequently, the operations defined on those types may differ; where we might be able to multiply two integers, it is not immediately clear what multiplying two strings means. Multiplying two integers yields again an integer, this behavior is encoded into type rules which comprise a set of premises, assertions that must hold prior, and a set of conclusions, which hold if all the premises are fullfilled. As such, we may encode the fact that multiplying two integers yields again an integer into a type rule. Leveraging the type system and type rules will allow automatic resource analysis as presented in **??**.

2 Linear Amortized Analysis

Linear Amortized Analysis concerns itself with linear potentials only, that is potential functions that are linear with respect to the input parameter. As such, $f(n) = 3 \cdot n + 2$ is considered a linear potential and $f(n) = 3 \cdot n^2 + 2$ is not. In chapter **??** we introduce automatic amortized analysis for polynomial potentials, which builds on the case for linear potentials.

2.1 Type System

In order to enable bounding resource consumption, we need to define a type system that permits this. More precisely, we introduce a type system featuring resource-annotated types. This is done by supplementing types with a potential $q \in \mathbb{Q}$. One resource-annotated type is that of a generic list $L^q(A)$. That is, a list comprising elements of type A, where *every element* of the list has the assigned potential q. We will see that this induces a potential of the form $f(n) = q \cdot n$, where n is the size of the list.

Another resource-annotated type are functions, written $A \xrightarrow{p/p'} B$. The meaning of the potentials p and p' is different compared to lists. The type $A \xrightarrow{p/p'} B$ can be interpreted as a function from type A to type B, for which we need p *additional* resources in order to start the evaluation and are left with p' resources after evaluation. The resources p are *additional*, as the type A may be resource-annotated itself.

The type system used is given as an EBNF in Figure 2.1 below:

$$A, B = Unit | A \times B | A + B | L^q(A) | A \xrightarrow{p/p'} B$$

Figure 2.1: Resource-Annotated Type System

Maybe split the type system into primitive two different grammars?

Besides the aforementioned types, pairs, denoted by $A \times B$, and sum types, denoted by $A + B$ are available types. Practical examples for sum types with which the reader might be familiar are, among many: union in C++ and enums in rust.

Before introducing type rules, let us build an intuition for resource-bound types, by working through a rudimentary example. Given the function *addL* in figure 2.2 below, we want to calculate an upper bound on heap-space usage. For this, we conclude that a list storing values of a primitive type A must allocate two memory cells. One for the value itself, and one for the pointer to the next element in the list. Furthermore, storing a list of type *nil* demands no memory. This choice is mainly for convenience, as it only alters the resulting amount of memory cells by a constant term.

Equipped with this assumption, we can immediately conclude: Given a list l of length n, the function *addL* requires $2n$ memory cells. Hence, we get $l : L^2(A)$. *addL* requires no additional resources, besides those supplemented by the list l.

Let us now incrementally build up a type for the function *addL*. Because it is a function type, we can start with $addL : A \xrightarrow{p/p'} B$. The input of *addL* is of type $(int, L^q(A))$, a pair comprising an integer and a list. Updating our initial typing, we get $addL : (int, L^q(A)) \xrightarrow{p/p'} B)$. Because *addL* returns a list, we can further update the type B, yielding $addL : (int, L^q(A)) \xrightarrow{p/p'} L^{q'}(A)$. Lastly, we need to infer the resource bounds p, p', q, q'. We already inferred that $q = 2$ and that $p = p' = 0$. Thus, the only resource annotation missing is q'. Since all the necessary resources are provided by the input list, the resource bound does not increase with respect to the output list.

Thus, we arrive at the following type for *addL*:

$$addL : (int, L^2(A)) \xrightarrow{0/0} L^0(A)$$

```
fun addL i l = match l with | nil -> nil
    | x::xs -> (x + i)::(addL i xs)
end
```

Figure 2.2: AddL function

In order to automize the above procedure, the rules for inference need to be rigorously defined by means of *type rules*. Furthermore, we need to select a set of potential functions that are specifically handy for automatic analysis. This is the aim of

2.2 The Potential Function

Before defining the potential function, we need to introduce a couple of definitions in order to permit a rigorous definition. Let A be a (resource-annotated) type, denote by $[\![A]\!]$ the set of *semantic values* of type A. That is, all the concrete values that belong to type A. $[\![L^q(int)]\!]$, therefore, describes the set of lists of integers, and $[1; 2; 3] \in [\![L^q(int)]\!]$.

When arguing about the potential of a variable, we need to consider the *heap* and the *stack*, denoted by H and V respectively. This is because the type of a variables, as well as its potential, can only be inferred if we can map the variable to a concrete value - which is precisely what the stack does. The correct notation would therefore be $\Phi(V(x) : A)$, which is less ergonomic than writing $\Phi(x : A)$. For convenience, we use the second notation and assume that a stack V is given implicitly. In order to track resource-consumption, we need to supply a heap H. Similarly, we assume the head as implicit and use our ergonomic notation, instead of $\Phi_H(x : A)$. We denote the set of types with linear potential by $\mathcal{A}_{lin}$.

Throughout this thesis we assume that any primitive types, that is types without a resource-annotation, have no effect on the resource consumption. As such their potential is zero.

Elaborate on why this is okay and how to move to non zero potentials

2.3 Judgements

In this section we will introduce all needed definitions from type theory, that will allow us to encode a notion of resource usage in type judgements. Let *Bochum* be a string of characters. In type theory, writing *Bochum* : T is called a *typing judgement*, stating that *Bochum* is of type T. In this example T could be the type *String*. However, *Bochum* : *int* would not be valid, for obvious reasons.

In the above example, we were judging the concrete *value*. In most cases, we dont perform judgements over concrete values, but using the variable names assigned to them; this resembles how software is written, in that we assign variable names to values. Because judgments usually only comprise variable names, we need to have a mechanism that allows associating variable names with values - this is precisely what *contexts* do. A context Γ is a mapping from variable names to values. Putting both together, we can write the following judgement: $\Gamma \vdash e : A$, which can be interpreted as "Given the context Γ, the expression e is of type A". After introducing the Type system in 2.1, we will introduce typing rules. Those will provide a mechanism of deducing new information.

Whenever we make a claim about the result of some computation, there are different levels we have to discriminate.

Do these judgements have specific names???

We call the first type of judgement a *Bounding Judgement*, and encode it as $\Gamma \mid\!\frac{p}{p'}\; e : A$, where Γ is a context, p and p' are non-negative rationals and e is some expression of type A. The judgement can then be read as "Given the context Γ and p resources, we can evaluate the expression e of type A, and we have p' resources remaining".

The second type of judgement is an *Evaluation Judgement*. While a *Bounding Judgment* provides a judgement about the type of an expression along with a resource bound, an *Evaluation Judgement* is an assertion about the concrete result of an expression. The prior permits eliciting resource bounds for *any* expression of a specific type, whereas the latter provides concrete bounds for specific expression.

We denote a *Bounding Judgement* by writing $e \downarrow v$, expressing that the expression e evaluates to the value v. This judgement can also be decorated with resource bounds in the following way: Denote by $e \underset{p/p'}{\downarrow} v$ that the expression e evaluates to the value v, requiring p available resources beforehand and returning p' resources after evaluation.

2.3.1 Type rules

List of Figures

List of Tables