RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Type Systems in Automatic Amortized Resource Analysis

Luca Witt

Bachelor's Thesis – June 21, 2023
Chair for Quantum Information.

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	Luca Witt

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	Luca Witt

Contents

1 Preliminaries

1.1 Amortized Analysis: Potential Method

Analysing the time and space complexity of algorithms is itself a vast field. There are three methods for computing amortized resourec bounds: Aggregate method, accounting method and the potential method. For AARA, potential method is used, as one can define particularly intuitive and ergonomic potential functions, which is done for linear potentials in **??** and for polynomial potentials in **??**. To perform amortized analysis, we define a potential function Φ, which assigns to every possible state of a data structure a *non-negative* integer. The *amortized* cost of an operation accounts for the actual cost of the operation, as well as the difference in potential as a consequence of the operation. Using the potential assigned to a specific state, more expensive operations can be amortized by preceding, cheaper operations. As we will see, this provides a less pessimistic bound than worst-case analysis while matching empirical bounds tightly .

reference data from thesis

As a firm grasp on the potential method is essential to understanding AARA, we contrast amortized analysis and worst-case analysis, by 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 the 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 $C_{\text{actual}}(o)$ to denote the actual cost of some operation o as well as S_{before} and S_{after} for the state of the DynamicArray (or any arbitrary data structure) before and after performing operation o. This yields the following formula for the amortized cost of an operation o:

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

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:

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

Label/annotate equations?

Hence, $\Phi(S_{\text{after}}) - \Phi(S_{\text{before}}) = 2$. This yields an amortized cost of $C_{\text{amortized}}(o) = C_{\text{actual}}(o) + 2$, 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:

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

Note that $n + 1 = N$, because the array needed to double in size. The potential therefore simplifies to $\Phi(S_{\text{after}}) = 0$. 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: $C_{\text{amortized}}(o) = \mathcal{O}(n) + (0 - \mathcal{O}(n))$. 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:

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

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. In **??**, we introduce a set of potential functions that are especially ergonomic for AARA with linear potential. This forms the basis for extending AARA to polynomial potentials in **??**.

Reference to chapter for potential properties.

1.2 Big-Step operational semantics

Operational Semantics provide a framework for reasoning about properties of parts of a formalized language, such as safety or correctness. In the case of programming languages, operational semantics define how a program is interpreted as a sequence of computations. In order to reason about the behavior of a sequence of computational steps, we need to choose an appropriate type of operational semantics. There are two types of operational semantics: small-step and big-step operational semantics, both differing in the level of detail and abstraction level they offer. Small-step operational semantics provide a detailed account of atomic evaluation steps, focusing on the transitions from one step in the program to another. Big-step operational semantics on the other hand focus on relation the input to the output of expressions, providing a higher level of abstraction. Most papers on AARA use big-step operational semantics, such as . One reason for favoring big-step operational semantics is their higher level of abstraction, as we are most interested in the relation between input and output.

Reference papers

Before we are able to define big-step operational semantics, we need to introduce a couple of concepts, namely: stack, heap and evaluation judgement. All three are necessary to provide resource bounds for the evaluation of expressions.

Since expressions can contain variable names, we need a mechanism of mapping variable names to the values associated with them - this is what the stack does. A stack, denoted by V, is a mapping from variable identifiers to values: $V : VID \rightarrow Val$. As such, different stacks may map the same variable name to different values, which could drastically alter the resulting potential. To avoid ambiguity, we specify a stack V explicitly when necessary. When tracking resource consumption in the form of memory, we need to know precisely how much memory is used and freed after an evaluation step. For this we need a heap, denoted by H. Any values that are kept in memory are stored on the stack. Thus, a stack H is a mapping from locations to values: $H : loc \rightarrow val$.

Having defined stacks and heaps, we can now define evaluation judgements with resource-annotations. An evaluation judgment has the following form:

$$V, H, e \Downarrow v, H' \mid (p, p')$$

With the following meaning: Given a stack V and a heap H, the expression e evaluates to the value v and the new stack H'. In order to evaluate e we need p resources beforehand, and are left with p' resources after evaluation. This yields a resource consumption $\delta = p - p'$, which can be a negative integer if resources become available after the evaluation of the expression.

Type rules for operational semantics?

2 Linear Amortized Analysis

In this chapter, we introduce AARA for linear potentials, 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 AARA for polynomial potentials, which augments the case of linear potentials by introducing a specific set of polynomial functions as potentials.

2.1 Type System

In order to bound resource consumption, we need to define a type system that a notion of resources. More precisely, we introduce a type system featuring resource-annotated types. This is done by supplementing types with a potential $q \in \mathbb{Q}$. The interpretation of the annotated potential $q \in \mathbb{Q}$ depends on the specific type. 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 $A \overset{p/p'}{\text{III}\!\!\rightarrow} B$. The meaning of the potentials p and p' is different compared to lists. The type $A \overset{p/p'}{\text{III}\!\!\rightarrow} B$ 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:

$$A, B = Unit | A \times B | A + B | L^q(A) | A \overset{p/p'}{\text{III}\!\!\rightarrow} B$$

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. Suprisingly, neither pairs nor sum types have a resource-annotation attached to them. .

Explain why

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 $addL : A \overset{p/p'}{\text{III}\!\!\rightarrow} B$. 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 $addL : (int, L^q(A)) \overset{p/p'}{\text{III}\!\!\rightarrow} B)$. Because *addL* returns a list, we can further update the type B, yielding $addL : (int, L^q(A)) \overset{p/p'}{\text{III}\!\!\rightarrow} L^{q'}(A)$. 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*:

$$\text{addL} : (int, L^2(A)) \overset{0/0}{\text{III}\!\!\rightarrow} L^0(A)$$

```
1 fun addL i l = match l with | nil -> nil
2    | x::xs -> (x + i)::(addL i xs)
3 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

We now define the potential function for the type system in 2.1:

$$
\begin{aligned}
\Phi(a : A) &= 0 && A = Unit \\
\Phi(left(a) : A + B) &= \Phi(a : A) \\
\Phi(right(b) : A + B) &= \Phi(b : B) \\
\Phi((a, b) : A \times B) &= \Phi(a : A) + \Phi(b : B) \\
\Phi(l : L^q(A)) &= q \cdot n + \sum_{i=1}^{n} \Phi(a_i : A) && l = [a_1, \ldots, a_n]
\end{aligned}
$$

As our type system 2.1 permits nesting lists, we could have an element l of type $L^q(L^r(A))$. The potential function we defined also permits those types of values. We get:

$$\Phi(l : L^q(L^r(A))) = q \cdot n + \sum_{i}^{n} \Phi(l_i : L^r(A)) = q \cdot n + n \cdot (r \cdot m + \Phi(a_i : A))$$

which is a multivariate linear potential, as n and m are the lengths of the respective lists. This illustrates that our definition of linear potential functions allows for multivariate functions.

2.2.1 Subtyping and Sharing Relation

The overarching goal is to automize the elucidation of amortized resource bounds. More specifically, we want to have the tightest bounds possible. In this section, we define the subtyping and sharing relation, which will enable us to elicit the most general resource-annotated type and account for multiple occurences of a resource-annotated variable.

The *Subtyping relation* specifies the conditions under which one (resource-annotated) type is a subtype of another (resource-annotated) type. Intuitively, a resource-annotated type A is a subtype of a resource-annotated type B, if they have the same semantic values, i.e. $[\![A]\!] = [\![B]\!]$ and the potential of the subtype is greater than the potential of the supertype. Defining subtyping in this way results in the most general resource-annotated typing corresponding to the tightest possible potential.

We define the subtyping relation as the smallest relation that satisfies the following conditions:

$$
\begin{array}{rl}
A <: A & \text{for an arbitrary type A} \\
A \times B <: C \times D & \text{iff.} A <: C \ \& B <: D \\
A + B <: C + D & \text{iff.} A <: C \ \& B <: D \\
L^q(A) <: L^r(B) & \text{iff.} A <: B \ \& q \geq r
\end{array}
$$

The *Sharing* relation allows us to have a consistent potential, even if a variable if referenced multiple times. Consider the function ... below as an illustrative example.

add an example, illustrating sharing

Intuitively, the sharing relation ensures that whenever there are multiple references to a variable, the potential of the variable that is being referenced is split up into the references. This ensures that our potential function behaves as expected in those cases...

We define the sharing relation as the smallest relation, such that the following conditions are satisfied:

$$
\begin{array}{rl}
A \curlyvee (A, A) & \text{for an arbitrary type A} \\
A \times B \curlyvee ((A_1 \times B_1), (A_2 \times B_2)) & \text{iff.} A \curlyvee (A_1, A_2) \ \& B \curlyvee (B_1, B_2) \\
A + B \curlyvee ((A_1 + B_1), (A_2 + B_2)) & \text{iff.} A \curlyvee (A_1, A_2) \ \& B \curlyvee (B_1, B_2) \\
L^q(A) \curlyvee (L^r(A_1), L^m(A_2)) & \text{iff.} A \curlyvee (A_1, A_2) \ \& q = r + m
\end{array}
$$

Make the ampersands prettier

For the subtyping and sharing relation we have a corrolary that relates them to the potential function we defined earlier.

Lemma 1. *Let A and B be types, such that $A <: B$. It then holds for every $a \in A$ and $b \in B$ that: $\Phi(a : A) \geq \Phi(b : B)$.*

Lemma 2. *Let A, A_1, A_2 be types, such that $A \curlyvee A_1, A_2$. It then holds for every $a \in A, a_1 \in A_1, a_2 \in A_2$ that: $\Phi(a : A) = \Phi(a_1 : A_1) + \Phi(a_2 : A_2)$*

Prettify the Lemmata

2.2.2 Type rules

3 Univariate Polynomial Potential

Having introduced linear potentials, this chapter will augment the constructs introduced in 2 to allow univariate polynomial bounds. To achieve this, we change the potential from a rational number to a vector in $\mathbb{Q}^k$. In conjunction, we define a set of polynomial functions that use binomial coefficients as their coefficients; entries in the vector then correspond to values of the binomial coefficient of the monomial.

Rough introduction on additivie shift

3.1 Polynomial Potential Function

To accomodate polynomial potentials, we need to change the type system from 2, by changing the potential annotation of the generic list type. Instead of $L^q(A)$ with $q \in \mathbb{Q}$, we write $L^{\vec{q}}(A)$ where $\vec{q} \in \mathbb{Q}^k$. This yields the following type system for univariate polynomial potentials:

$$A, B = \mathbb{1} | A \times B | A + B | L^{\vec{q}}(A) | A \overset{p/p'}{\Longrightarrow} B$$

Figure 3.1: Type System using Polynomial Potential

Next, we define how to interpret the potential $\vec{q}$. For this, we restrict polynomial potential functions to a specific form; that is, functions with binomial coefficients as their coefficients. We then identify entries in $\vec{q}$ as indexing the polynomial. We define the potential of $L^{\vec{q}}(A)$ as :

$$\Phi(l : L^{\vec{q}}(A)) = \sum_{i=1}^{k} q_i \cdot \binom{n}{i} + \sum_{j=1}^{n} \Phi(a_i : A)$$

Where n is the length of the list, a_i are elements of the list for $1 \leq i \leq n$, and $\vec{q} = (q_1, \ldots, q_k)$.

Let us look at a concrete example, before continuing. Let us suppose, we have a list l of the type $L^{\vec{q}}(\mathbb{1})$, for $\vec{q} = (1, 2)$. Calculating the potential yields:

$$\Phi(l : L^{\vec{q}}(\mathbb{1})) = \sum_{i=1}^{2} q_i \cdot \binom{n}{i} + \sum_{j=1}^{n} \Phi(a_i : \mathbb{1})$$

As elements of type $\mathbb{1}$ have potential zero assigned to them, the equation above simplifies to:

$$\Phi(l : L^{\vec{q}}(\mathbb{1})) = \sum_{i=1}^{2} q_i \cdot \binom{n}{i}$$

We can now insert the values q_1 and q_2, to get:

$$\begin{aligned}\Phi(l : L^{\vec{q}}(\mathbb{1})) &= 1 \cdot \binom{n}{1} + 2 \cdot \binom{n}{2} \\ &= n + 2 \cdot \frac{n \cdot (n-1)}{2} \\ &= n + n^2 - n \\ &= n^2\end{aligned}$$

3.1.1 Subtyping and Sharing Relation

The subtyping and sharing relation introduced in **??** only need to be augmented with respect to lists. As we now define a potential to be an element $\vec{q}, \vec{r} \in \mathbb{Q}^k$, we need to specify what the following inequality means: $\vec{q} \geq \vec{r}$. With this definition in place, we are again able to capture the most general resource-annotated typing for a generic list. We define the following:

$$\vec{q} \geq \vec{r} \iff \forall i \in I : q_i \geq r_i$$

Prettify/more concise notation?

Using this definition, we will be able to prove a similar result for polynomial potentials; linking the magnitude of potentials with the subtyping relation. Before doing that, let us define the subtyping relation for polynomial resource-annotated types. We define the subtyping relation as the smallest relation satisfying the following conditions:

$$\begin{aligned} A &<: A && \text{for an arbitrary type A} \\ A \times B &<: C \times D && \text{iff.} A <: C \ \& B <: D \\ A + B &<: C + D && \text{iff.} A <: C \ \& B <: D \\ L^{\vec{q}}(A) &<: L^{\vec{r}}(B) && \text{iff.} A <: B \ \& \vec{q} \geq \vec{r} \end{aligned}$$

Lemma 3. *Let A, B be resource-annotated types. If $A <: B$, then $\Phi(a : A) \geq \Phi(a : B)$ for all $a \in [\![A]\!]$.*

Proof. In order to prove that the inequality holds for an arbitrary type, we simply prove it for every inductive case.

$(A <: A)$: This case is trivial, as $\Phi(a : A) = \Phi(a : A)$ trivially holds.

$(A \times B <: C \times D)$:

$$\begin{aligned} \Phi((c,d) : C \times D) &= \Phi(c : C) + \Phi(d : D) && \text{(by definition)} \\ &\leq \Phi(c : A) + \Phi(d : B) && (A <: C \text{ and } B <: D) \\ &= \Phi((c,d) : A \times B) && \text{(by definition)} \end{aligned}$$

$(A + B <: C + D)$:

$$\begin{aligned} \Phi(\text{left}(a) : A + B) &= \Phi(a : A) && \text{(by definition)} \\ &\geq \Phi(a : C) && (A <: C) \\ &= \Phi(\text{left}(a) : C + D) && \text{(by defintion)} \end{aligned}$$

$$\begin{aligned} \Phi(\text{right}(b) : A + B) &= \Phi(b : B) && \text{(by definition)} \\ &\geq \Phi(b : D) && (B <: D) \\ &= \Phi(\text{right}(b) : C + D) && \text{(by definition)} \end{aligned}$$

$(L^{\vec{q}}(A) <: L^{\vec{r}}(B))$:

$$\begin{aligned} \Phi(l : L^{\vec{q}}(B)) &= \phi(n, \vec{q}) + \sum_{i=1}^{n} \Phi(l_i : B) && \text{(by definition)} \\ &\geq \phi(n, \vec{r}) + \sum_{i=1}^{n} \Phi(l_i : B) && (\vec{q} \geq \vec{r}) \\ &\geq \phi(n, \vec{r}) + \sum_{i=1}^{n} \Phi(l_i : A) && (A <: B) \\ &= \Phi(l : L^{\vec{r}(A))} \end{aligned}$$

□

Prettier way of typesetting the proof?

4 Rudimentary Type System

In this chapter, we define the type system, along with the associated operational semantics and type rules, for a simple system, that only counts the amount of ticks in a program. The chapter concludes with a soundness proof, showing that operational semantics and type rules line up. In order to elicit resource bounds for programs, we need two things: (1) A construct in the programming language that allows altering the calculated resource consumption and (2) types that reflect resource consumption. The first problem is solved by introducing tick q to the programming language. Where q is a rational number, indicating the amount of resources that are either consumed of freed. The second problem is solved by embellishing the unit type with two resource annotations, one for the resources required a priori, and another for the resources freed afterward.

In the figure below is the programming language that we use:

$$\begin{array}{lll} e := & \text{let } \_ = e_1 \text{ in } e_2 & \text{(let)} \\ & \text{tick } q & \text{(tick)} \end{array}$$

4.1 Operational Semantics

In this section, we define what it means for a program to consume (or free) resources, by defining the operational semantics needed. To this end, let H be a heap e be an expression and v be the value produced by the expression e. We then write $H \vdash e \rightsquigarrow v, H'$ for the fact that given heap H the expression e evaluates to the value v and yields the new heap H'. In order to capture resource-consumption, we augment this notation like this $H \vdash e \rightsquigarrow v, H' \mid (p, p')$, where p and p' are integers and p being the amount of resources needed to start evaluating the expression, and p' being the resources freed after the evaluation of expression e.

Having defined the notion of evaluation, we now introduce rules for the operational semantics provided. First, we need a rule that allows us to combine two expression into a let expression. We also need to, however, provide resource annotations for the resulting let expression. Therefore, we also need to add numeric constraints, resulting in the following rule:

$$(\text{let})\frac{H_1 \vdash e_1 \rightsquigarrow v \; H_1' \mid (p, p') \quad , \quad H_1' \vdash e_2 \rightsquigarrow v_2 \; H_2' \mid (q, q')}{H_1 \vdash \textbf{let } \_ = e_1 \textbf{ in } e_2 \mid (p + q - p', q')}$$

Let us briefly explain how we concluded the resulting resource annotation of $(p + q - p', q')$ for the resulting **let** expression. We want to have enough resources, in order to evaluate e_1 and e_2 sequentially. Therefore, we need *at most* $p + q$ resources. We can provide a more precise lower bound, as we are reimbursed p' resources after the execution of e_1. This leads to the aforementioned resource annotation.

Similarly, we now define a rule for the **tick** instruction. The expression **tick** q consumes precisely q resources; or frees q resources, should the amount be negative. As a result, any resource-annotation where a priori and a posteriori resources differ by at least q, is valid. This leads to the following rule:

$$(\text{tick})\frac{q_0 \geq q + q_1}{H \vdash e \rightsquigarrow \mid (q_0, q_1)}$$

4.2 Type Rules

The only resource-annotated type that we permit is the unit type, denoted by $\mathbb{1}_r^p$, where p provides a lower bound on the resources needed in order to evaluate an expression of that type, and r provides a bound on the amount of resources that are freed after evaluating an expression of that type.

Similarly to the previous section on operational semantics, we define two rules, one for **let** expressions and one for **tick**. Before we can introduce the rule for **let**, however, we need to define how resource-annotations of types propagate. For this, consider the following example:

$$\frac{\Gamma_1 \vdash e_1 : \mathbb{1}_r^p \quad , \quad \Gamma_2 \vdash e_2 : \mathbb{1}_t^s}{\Gamma_1\Gamma_2 \vdash \textbf{let } \_ = e_1 \textbf{ in } e_2 : ?}$$

It is not immediately clear, what type the resulting **let** expression should have. Using a similar argument from the **let** rule in the section on operational semantics, we get the type $\mathbb{1}_t^{p+s-r}$. To make this notation more ergonomic, we define the following operation, easing writers of verbose type annotations.

Definition 4.2.1. *Let $\mathbb{1}_r^p$ and $\mathbb{1}_t^s$ be resource-annotated types. We define $\mathbb{1}_r^p \ltimes \mathbb{1}_t^s = \mathbb{1}_t^{p+s-r}$.*

Is that definition a good idea? Should i embellish it with some more stuff?

Using the newly defined operation on resource-annotated types, we get the following rule:

$$\text{(T:let)}\frac{\Gamma_1 \vdash e_1 : \mathbb{1}_r^p \quad , \quad \Gamma_2 \vdash e_2 : \mathbb{1}_t^s}{\Gamma_1\Gamma_2 \vdash \textbf{let } \_ = e_1 \textbf{ in } e_2 : \ \mathbb{1}_r^p \ltimes \mathbb{1}_t^s}$$

For **tick**, we get the following rule:

$$\text{(T:tick)}\frac{q_0 \geq q + q_1}{\Gamma \vdash \textbf{tick } q : \mathbb{1}_{q_1}^{q_0}}$$

Let us now perform an examplery type derivation for a simple program, using the type rules we just introduced, to get a better understanding of how the definitions orchestrate together. For this, consider the following program:

$$\begin{aligned}&\textbf{let} \_ = \textbf{tick } 3 \textbf{ in}\\ &\textbf{let} \_ = \textbf{tick } -2 \textbf{ in}\\ &\textbf{let} \_ = \textbf{tick } 5 \textbf{ in}\end{aligned}$$

Intuitively, we have three distinct steps, the first one consuming 3 resources, the second one freeing up 2 resources and the last one consuming 5 resources. Let us now derive a typing for the program.

Copy type derivation neatly from handwritten paper

List of Figures

List of Tables