RUHR-UNIVERSITÄT BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Type Systems in Automatic Amortized Resource Analysis

Luca Witt

Bachelor's Thesis – September 22, 2023
Chair for Quantum Information.

Supervisor:	Prof. Michael Walter
Daily Supervisor:	Sai Anurudh Reddy Peduri

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

The performance of computer programs is of paramount interest to software engineers. There are multiple lenses through which performance may be evaluated: Worst-case analysis, average-case analysis and amortized analysis. In the case of amortized analysis, however, analysis was previously done manually - making it tedious and error-prone. This thesis presents Automatic Amortized Resource Analysis (AARA), a method that permits *automatic* elicitation of a program's resource consumption through augmented type inference.

AARA embellishes type rules with linear constraints. As part of type inference, those constraints are collected and solved using an LP-solver.

This thesis presents concepts of AARA by incrementally building up a programming language. In chapter 2 we start off by introducing resource demands, which capture the resource consumption of a program. We then introduce a sequencing operation that allows composing resource demands, to build up larger programs, and a relation that facilitates comparing resource demands.

In chapter 3 we introduce the *tick* instruction, which is an explicit handle for consuming/freeing resources. In chapter 4, we extend this language by introducing variables. In chapter 5, we introduce lists as a data type. This requires introducing potentials in order to elicit lower bounds on the resource consumption that is variable under the length of the list.

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

Contents

Todo list

Symbols

Resources

$\mathcal{R}$	Set of resources (Definition 2.1)
$\mathcal{D}$	Resource Demands (Definition 2.2)
$\triangleright$	Sequencing Operator (Definition 2.6)
$\succcurlyeq$	Relaxation Relation (Definition 2.9)

Typing

$\mathbb{T}$	Set of Types (Definition 3.3, Definition 4.2, Definition 5.2)
Unit	Unit Type
$\bullet$	The Unit Value
Bool	Type Bool
Int	Type Int
$A_{p_1}^{p_0}$	Resource-Aware Type (Definition 3.5)
$A \to B_{p_1}^{p_0}$	Resource-Aware Function
Γ	Context (Definition 4.8)
$\mathbf{dom}(\Gamma)$	Domain of a Context (Definition 4.9)
$\Gamma_1 \cap \Gamma_2 = \emptyset$	Disjoint Contexts (Definition 4.10)
$\Gamma_1; \Gamma_2$	Union of Contexts
$\Gamma; x : A$	Augmenting a Context
$\mathbf{List}^q(A)$	List Type with Potential (Definition 5.5)
$\mathbf{List}_n(A)$	Set of Lists with length n (Definition 5.3)
$\lvert l \rvert$	Length of List (Definition 5.7)

Φ	Potential Function (Definition 5.6)
$\Gamma \vdash e : A$	Typing Judgement (Definition 4.11)

Evaluation

E	Environment (Definition 4.4)
$E(x)$	Value of x
$\mathbf{dom}(E)$	Domain of E (Section 4.1.1)
$x \in E$	Existence of x in E (Section 4.1.1)
$E[x \mapsto y]$	Map x to y in E (Definition 4.6)
$E \vDash e \overset{p_0 \; p_1}{\rightsquigarrow} v$	Evaluation judgement (Definition 4.7)
Var	Set of Variable Names (Chapter 4)
Vals	Set of Values (Definition 4.3)

PL Syntax

$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	Let expression (Definition 5.1)
$\mathbf{tick}\ k$	Tick (Definition 5.1)
x :: xs	List Constructor (Definition 5.1)
nil	Nil Constructor (Definition 5.1)
$\mathbf{match}\ l\ \mathbf{with}\ \mid \mathbf{nil} \to e_1 \mid x :: xs \to e_2$	Match on Lists (Definition 5.1)
$f\ x$	Function Application (Definition 5.1)
$\mathbf{letfun}\ f = x \to e_f\ \mathbf{fin}\ e_2$	Function Declaration (Definition 5.1)

1 Introduction

Analyzing the performance of algorithms features multiple approaches: worst-case, average-case and amortized analysis. Each of these serves a nuanced purpose, and helps in understanding the overall performance of algorithms better. Amortized analysis, on which automatic amortized resource analysis (AARA) builds, describes the performance of algorithms by analyzing a sequence of instructions. Additionally, instructions can be *amortized* by the state of a data structure.

Performing amortized analysis by hand, however, is time-consuming and error-prone. We first need to define a suitable potential function; this will allow the state of the data structure to amortize more expensive operations. Afterward we need to calculate the amortized cost of multiple instructions and average their results. AARA aims to automate this process, programmers write code and *additionally* provide minor annotations. The resulting program can be analyzed for performance automatically by a type inference algorithm, allowing programmers to allocate more time to working on software instead of analyzing it.

There are three key components to automatizing this procedure. First, we introduce a new expression **tick** k. This expression can be used by programmers to embed a virtual cost into their programs. The **tick** expression can be used to consume or free virtual resources. Next, we define type rules for our programming language. These type rules are *resource-aware* - they have a cost assigned to them, and propagate the cost of previous expressions properly. Finally, the program is analyzed using type inference. This step collects all information about resource consumption in the form of *linear constraints*, which are collectively solved to provide an upper-bound on the resource consumption.

Let us informally work through the AARA procedure, in order to foster a high-level understanding that is useful for following chapters. First, let us look at example code to see how annotating a program with **tick** instructions looks like.

```
def add1 l = match l with
   | nil -> nil
   | cons(x, xs) -> let _ = tick 1 in
                    let x' = x + 1 in
                    let xs' = add1 xs in
                    cons(x', xs')
```

In the code above, we define a function *add1*, which given a list of integers returns the list of integers with every element incremented. For example, the function maps $(1, 2, 3, 4)$ to $(2, 3, 4, 5)$. Furthermore, there is a **tick** expression in line three. This tick instruction resembles a cost of one resource. Thus, incrementing a list costs one resource *per element* in the list.

With this fundamental understanding of the function and its associated cost, let us understand how this cost is embedded into type rules in the form of constraints, and, how to arrive at a resource bound from a set of constraints.

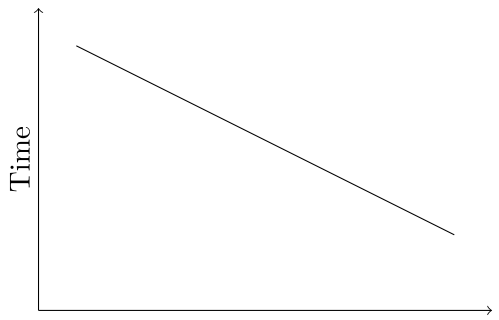
Types are to programs what sets are to mathematics. In a mathematical setting, an element may be a member of a set. Analogously, a program and its values can have a *type*. Strings and floats are examples of types that most programmers know. In the same way that mathematical functions map between sets, functions, like $add1$ above, map between types. In this case, the function $add1$ maps from $\text{List}(\text{Int})$ to $\text{List}(\text{Int})$. We denote this by writing $add1 :: \text{List}(\text{Int}) \to \text{List}(\text{Int})$.

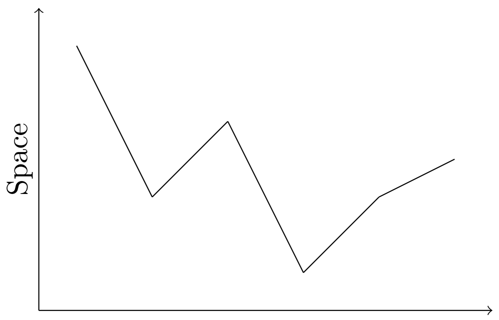
Type rules allow us to reason about the type of transformed values. For example, if we know that 1 and 2 are integers then $1 + 2$ is also an integer. This kind of reasoning is encoded into type rules in the form of premises and conclusions. The premises have to be fulfilled in order for the conclusion to hold. For the addition of integers, we have "if x and y are integers *then* $x + y$ is also an integer". For values and types we do not use the symbol $\in$ from set theory. Instead, we write $x : \text{Int}$ to denote that x has type integer. Type rules are written in the following form:

$$(\text{Addition})\frac{x : \text{Int} \quad y : \text{Int}}{x + y : \text{Int}}$$

Back to our example function $add1$. What could a type rule for the tick expression from the third line look like? Let us first understand what value the tick expression returns, as this will help us understand what type we should assign to **tick** 1. While addition of integers returns an integer, invoking **tick** 1 returns nothing. Readers with experience programming in java might recognize the type *void* to typically be used in such settings. We give the type void a different name, we call it *unit*. It is therefore justified to claim that **tick** $1 : \text{Unit}$.

In order to embed the resource consumption of **tick** 1, we need to first formulate precisely what resources are and how we choose to model them. The most common resources, with respect to which programs are analyzed, are time and space. While we classify both as resources, they differ vastly in their characteristics. While memory can be allocated and subsequently freed, time can only be consumed - it is not possible to generate or free time. If we visualize the evolution of time and space, this may look like the two diagrams below; the left-hand side displays time and the right-hand side displays space.





Embedding the entire evolution of an expression into the type rule is to elaborate - we need to reduce our representation. This leads us to model *resource demands* as the resources needed to start evaluating the program *and* the resources remaining afterward. We use tuples as a compact way to denote this. The tuple $(4, 2)$ then expresses that a program requires 4 resources to start evaluating and, once finished, leaves 2 resources remaining.

Thus, **tick** 1 from our example above can be described using $(1, 0)$, because **tick** 1 consumes 1 resource and leaves non resource free afterward. We can now justify claiming **tick** $1 : \text{Unit}$ with cost $(1, 0)$. The cost can be embedded into the type, to provide a more compact and readable notation, like so: **tick** $1 : \text{Unit}^1_0$. We get the following type rule for **tick** 1:

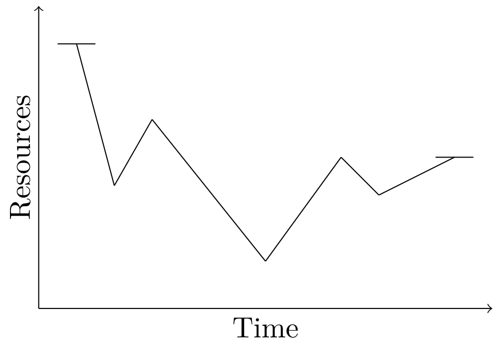
$$\text{(Tick 1)}\,\frac{}{\textbf{tick}\ 1 : \text{Unit}^1_0}$$

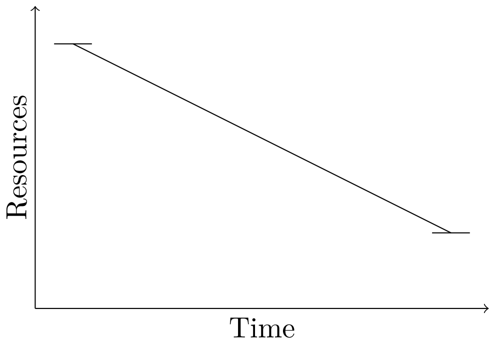
There are two limitations, however. First, using this approach we would need to provide a type rule for every $k \in \mathbb{Z}$. Secondly, we can see that there are many more tuples (p_0, p_1) that allow **tick** 1 to be executed. For example, $(2, 1)$ is also valid, because the program can have two resources allocated to it instead of one.

2 Resources

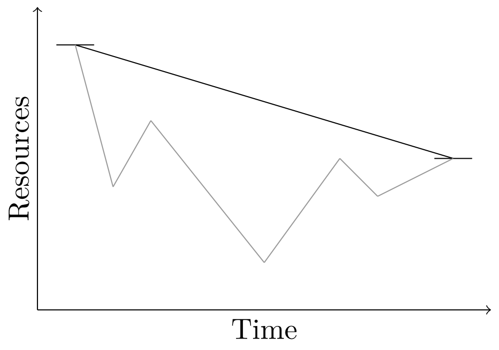
Throughout this thesis, we introduce programming language constructs, in order to elicit resource consumption of programs. It is, therefore, important to mention that resources can be an arbitrary cost metric, including but not limited to: memory consumption, runtime, energy consumption. While the details of the respective type of resource are nuanced, the resource consumption generally comprises two parts: Resources consumed to start evaluating, and possibly, resources freed after evaluation.

Tracing the heap-space usage of a program may look like the diagram on the left-hand side. It both consumes and frees resources in intermediary steps. For time consumption, things are different. We cannot reimburse time, but only consume it - as illustrated in the right-hand side example.

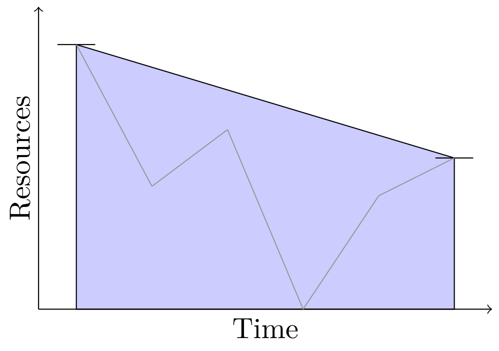




We are most interested in the value at the beginning and at the end, which leads us to simplify the resource consumption further. Drawing a straight line between resources at the start and at the end.



It is important to note that distinct programs may have the same start and end resources, while they differ vastly in their temporal development. In order to reflect this in a resource diagram, we draw a polygon which contains *every* valid development for the specific beginning and end resource amount.



We use this visual representation to foster intuition throughout this chapter - illustrating examples and concepts.

This overarching structure of resource demands leads us to a refined version of resource consumption. Furthermore, we will define a rich structure on the resource demand of programs; Most notably, we introduce sequencing of resource demands, which gives rise to a monoidal structure, and relaxation, which defines a *partial order* - allowing us to compare resource demands, in a way that is consistent with our intuition of cost.

2.1 Resource Demand

We start by defining resources mathematically, which will lead to defining resource demand of programs as a tuple of resources. Afterward, we introduce multiplication of these tuples and disambiguate reasoning about resource demand, by naming key figures of resource demands.

Definition 2.1 (Resources)**.** We identify resources with the natural numbers, where the value refers to an amount of resources.

$$\mathcal{R} = \mathbb{N}_0$$

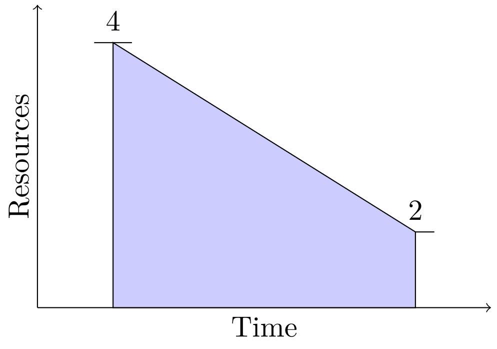
Definition 2.2 (Resource demand)**.** We denote by $\mathcal{D}$ the set of resource demands, which is a pair of resources.

$$\mathcal{D} = \mathcal{R} \times \mathcal{R}$$

When reasoning about a resource demand $(p_0, p_1) \in \mathcal{D}$, p_0 are the *initial* resources, and p_1 are the *residual* resources.

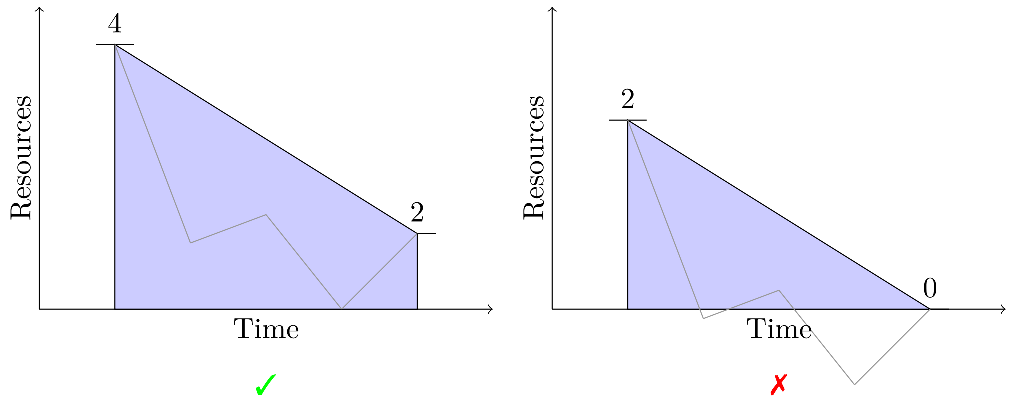
Example 2.3. $(4, 2) \in \mathcal{D}$

Representing this resource demand using resource diagrams yields:



For this example, one might wonder if we can simplify $(4, 2)$ to $(2, 0)$, by simply shifting the polygon downward. This is not possible, as a program with demand $(4, 2)$ may not have sufficient resources if it is initiated with only 2 initial resources.

Consider the diagrams below. On the left-hand side, the development of the program is contained in the associated polygon. However, on the right-hand side, this is not the case. The amount of available resources is even negative at some point - this is not valid.



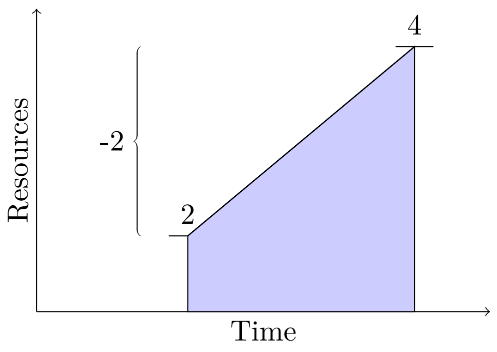
On the other hand, one may ask if a program in $(2, 0)$ can also be represented by $(4, 2)$. The answer is yes! Further clarification as to why this is the case is subject of Section 2.2.

The notion of *consumption* is nuanced. On the one hand, the example above may use all four resources at some point, but frees two resources after its execution. For our use, we view the demand of programs as a black box, only concerned with initial and residual resources. As a result, the *consumption* of a resource demand is precisely the difference between initial and residual resources.

Definition 2.4 (Resource Consumption)**.** Given $(p_0, p_1) \in \mathcal{D}$, we define the *resource consumption* as the difference between initial and residual resources, $p_0 - p_1$.

While a resource demand is a pair of *non-negative* integers, its resource consumption can be negative - this corresponds to a program that *frees* resources instead of consuming them. We can think of the resource consumption as the directed (!) height of the associated resource diagram. Consider the following example.

Example 2.5. For $(2, 4) \in \mathcal{R}$, the resource consumption equals $2 - 4 = -2$. An associated programs *frees* 2 resources!

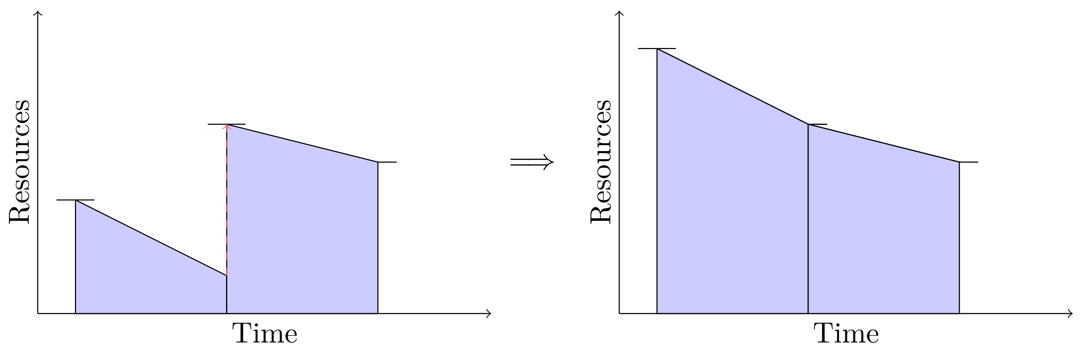


As most programs are made up of multiple subsequent evaluations, we need a mechanism for composing evaluations; More specifically, we need to define the resource cost of subsequent evaluations. For this, there are two cases: The first operation has enough residual resources for the second operation to be executed, or the second operation requires more initial resources than residual resources of the first operation provides. In the latter case we need to increase the initial resources of the composed evaluation.

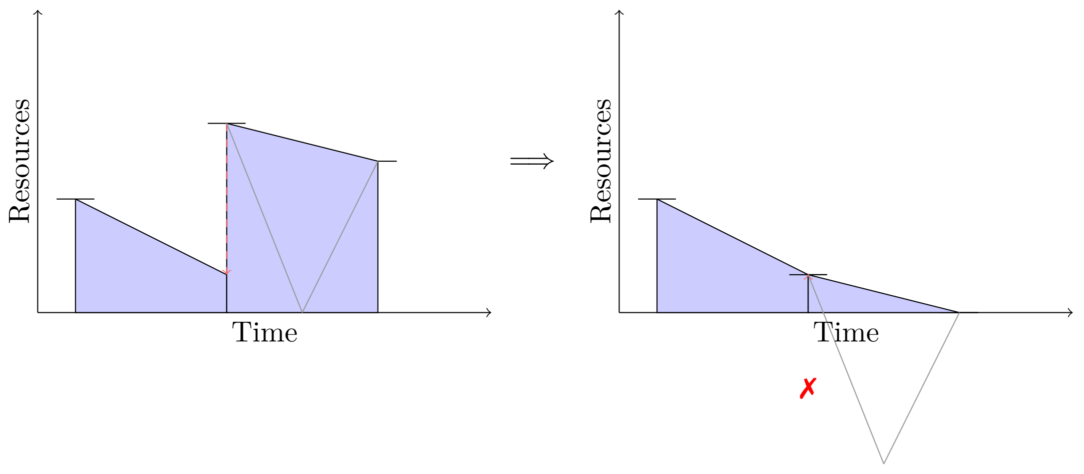
This yields two cases for the resulting resource demand:

$$\begin{cases} (p_0 + q_0 - p_1, q_1) & \text{if } q_0 \geq p_1 \\ (p_0, q_1 + p_1 - q_0) & \text{if } q_0 < p_1 \end{cases}$$

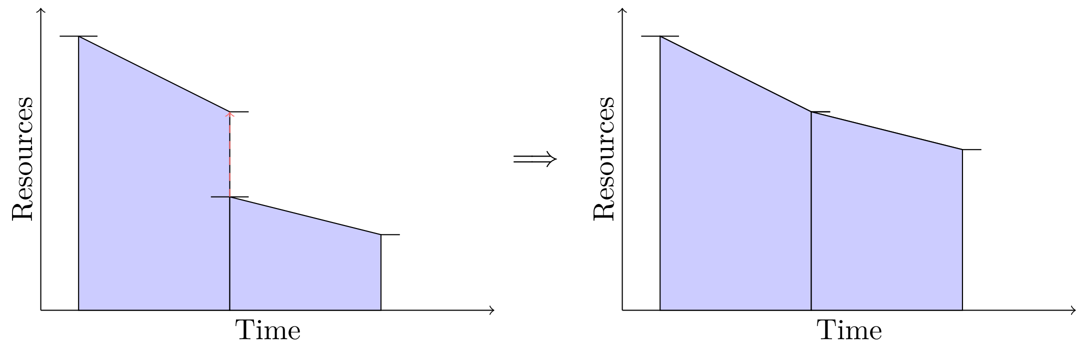
For the first case, we lift the first resource demand so its residual resources match the initial resources of subsequent resource demand - allowing us to compose both resource demands.



Intuitively, one may want to lower the second resource demand to match. This, however, is not a valid composition. Consider the same resource demands, but with an example progression outlined in the second resource demand. If we were to lower the second resource demand, we would have negative resources at some point in our evaluation, similar to **??** - this would not be sound.



For the second case, we lift the second resource demand to match. We cannot lower the first resource demand, due to reasons congruent with the example above. Fundamentally, composing resource demands forces us to lift either resource demand to match, as lowering any resource demand may reduce the set of programs represented by the resulting resource demand.



We can, however, simplify the multiplication of resource demands by merging the two cases. Consider that $p_0 - p_1 + \max(p_1, q_0)$ precisely matches the initial resources for both cases. Similarly, $q_1 - q_0 + \max(p_1, q_0)$ precisely matches the residual resources for both cases. This yields the following definition:

Definition 2.6 (Composition of resource demands)**.** Let (p_0, p_1) and (q_0, q_1) be two Resource demands, as introduced above. We then define:

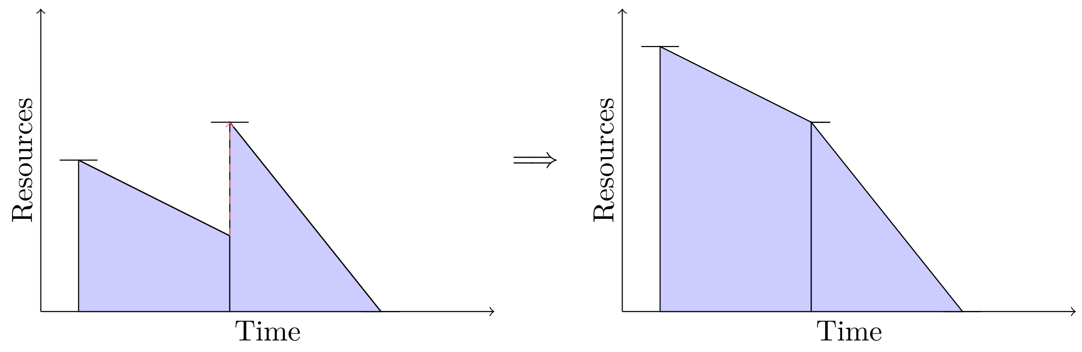
$$(p_0, p_1) \triangleright (q_0, q_1) = (p_0 - p_1 + \max(p_1, q_0), q_1 - q_0 + \max(p_1, q_0))$$

Elaborate/remark: This isnt linear, how can it be solve by LP?

As the value $\max(p_1, q_0)$ is key for composing resource demands, we call it the *disparity* of two resource demands.

Better name

Example 2.7. For $(4, 2), (5, 0) \in \mathcal{R}$, we have $(4, 2) \triangleright (5, 0) = (7, 0)$.



Note that compopsition of resource demands is *not* commutative. As a counterexample, consider $(2, 0) \triangleright (0, 2)$ and $(0, 2) \triangleright (2, 0)$. The first evaluates to $(2, 2)$, whereas the second evaluates to $(0, 0)$.

Lemma 2.8. Resource demands, with unit $(0, 0)$ form a *monoid* under sequencing.

Proof

Proof. □

2.2 Comparing Resource Pairs

Whenever we define a programming language, we provide evaluation semantics and type rules, which we ultimately want to link by proving their *soundness*. Soundness of a programming language with respect to evaluation semantics and type rules comprises two parts. First, we need to prove that the type we assign to an expression is consistent - evaluating the expression will produce a value of that type. Secondly, we need to show that the inferred resource demand is always *at least* the concrete resource demand of evaluating an expression. In subsequent chapters, the main challenge of proving soundness will be in proving consistency of resource demands.

In order to do this, we need to be able to compare resource demands. This ultimately motivates the definition of *relaxation*, which induces a partial order on the set of resource demands, streamlining future soundness proofs. Let us, therefore, begin by motivating how to compare resource demands in a way that aligns with our goal of AARA.

Intuitively, if a resource demand a is a *relaxation* of another resource demand b, we expect every program that is executable with respect to b to also be executable with respect to a. Therefore, a may have more *initial resources* than b. This makes sense, as a program would then have more resource available throughout its execution.

Furthermore, we want to compare the *consumption* of both resource demands. In this sense, if a is a relaxation of b then a should consume *at least* as many resources as b.

Definition 2.9 (Relaxation)**.** Let $(p_0, p_1), (q_0, q_1) \in \mathcal{D}$. We define the relation $\succcurlyeq$ as follows:

$$(p_0, p_1) \succcurlyeq (q_0, q_1) :\iff p_0 \geq q_0 \land p_0 - p_1 \geq q_0 - q_1$$

We then say that (p_0, p_1) is *a relaxation of* (q_0, q_1).

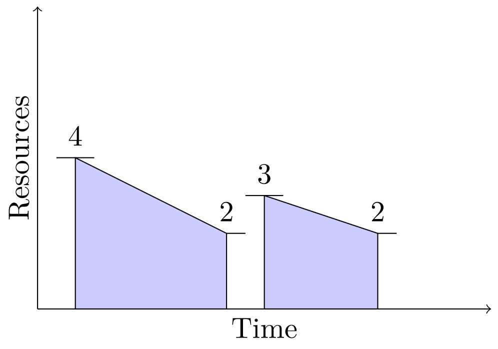
Another way to compare resource demands also comes to mind: Instead of the resource consumption to be at least as large, we could enforce that $p_1 \leq q_1$ - the relaxed resource demand needs *at most* as many residual resources. This would provide a less precise comparison of cost than our definition of relaxation.

To illustrate why, consider the two resource demands $(4, 2), (3, 1) \in \mathcal{D}$. Any program that can be executed under $(3, 1)$ should also be executable under $(4, 2)$; Intuitively, we provide the program one additional resource. Indeed, $(4, 2) \succcurlyeq (3, 1)$. For the second possible definition of relaxation, we see that $2 \not\leq 1$. This highlights the key difference between those definitions - relaxation allows lifting a resource demand, which the second notion prohibits.

Equipped with our definition of relaxation, we illustrate relaxation using resource diagrams. The relaxed resource demand has to start at least the same height *and* be at least as long as the resource demand compared to.

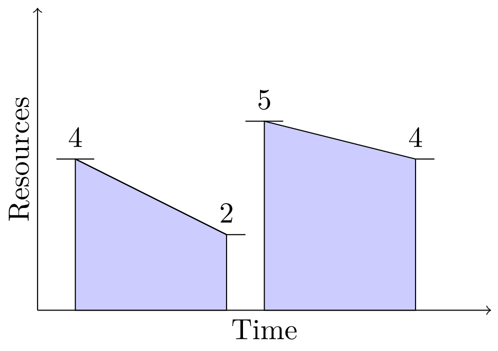
Example 2.10. $(4, 2) \succcurlyeq (3, 2)$

Here both constraints are satisfied, $4 \geq 3$ and $4 - 2 \geq 3 - 2$. We see: (1) The first resource demand has a higher starting point, indicating higher initial resources. (2) The first resource demand is vertically longer, which indicates larger resource consumption.



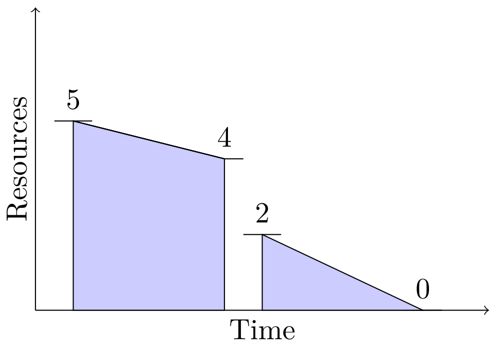
Example 2.11. $(4, 2) \not\succcurlyeq (5, 4)$

As $4 \not\geq 5$ we have less initial resources. Thus, $(4, 2)$ *cannot* be a relaxation. In the diagram below we see that the first resource demand begins at a lower point.



Example 2.12. $(5, 4) \not\succcurlyeq (2, 0)$

For this example it holds that $5-4 \not\geq 2-0$, which violates a constraint for relaxation - the first resource demand consumes *less* resources. In the diagram below, this correlates to the first resource demand being vertically smaller than the second one.



Most notably, relaxation defines a partial ordering on the set of resource demands. Let us recall the definition of *partial orders*:

Definition 2.13 (Partial Order)**.** Let M be a set, and $\leq$ be a relation on M. A partial order satisfies the following properties for any $a, b, c \in M$:

I Reflexivity: $a \leq a$

II Anti-symmetry: $a \leq b$ and $b \leq a$, then $a = b$

III Transitivity: $a \leq b$ and $b \leq c$, then $a \leq c$

Before we can show that resource demands form a partial order under relaxation, we need to define equality for resource demands. Intuitively two resource demands are equal, if the respective initial cost is equal and the resource consumption is equal.

Lemma 2.14. The relaxation relation $\succcurlyeq$ defines a partial ordering on the set of resource demands.

Proof. In the following, let $p, q, r \in \mathcal{D}$.

Reflexivity: It trivially holds that $p_0 \geq p_0$ and $p_0 - p_1 \geq p_0 - p_1$, which shows that $p \succcurlyeq p$.

Transitivity: Let $p \succcurlyeq q$, $q \succcurlyeq r$. We want to show that $p \succcurlyeq r$. Below we list all assumptions that follow, where the left-hand side follows from $p \succcurlyeq q$ and the right-hand side follows from $q \succcurlyeq r$.

$$p_0 \geq q_0 \qquad\qquad q_0 \geq r_0 \tag{2.1}$$
$$p_0 - p_1 \geq q_0 - q_1 \qquad\qquad q_0 - q_1 \geq r_0 - r_1 \tag{2.2}$$

Using both inequalities in Eq. (2.1) and transitivity of $\geq$ for natural numbers, it follows that $p_0 \geq r_0$. Similarly, we combine Eq. (2.2) and the transitivity of $\geq$ for the natural numbers to get $p_0 - p_1 \geq r_0 - r_1$. Both inequalities show that $p \succcurlyeq r$.

Anti-symmetry: Let $p \succcurlyeq q$ and $q \succcurlyeq p$. Again, we list all inequalities that follow from $p \succcurlyeq q$ and $q \succcurlyeq p$, respectively.

$$p_0 \geq q_0 \qquad\qquad q_0 \geq p_0 \tag{2.3}$$
$$p_0 - p_1 \geq q_0 - q_1 \qquad\qquad q_0 - q_1 \geq p_0 - p_1 \tag{2.4}$$

Using Eq. (2.3) we get $p_0 = q_0$ and using Eq. (2.4) we get $p_0 - p_1 = q_0 - q_1$. Thus, it follows that $p = q$.

□

It is important to note that, for any two resource pairs, one *need not* be a relaxation of the other. Therefore, relaxation does not define a total order - as the following example shows.

Example 2.15. Let $(4, 2), (5, 4) \in \mathcal{D}$. Then, $(4, 2) \not\succcurlyeq (5, 4)$ *and* $(5, 4) \not\succcurlyeq (4, 2)$.

In later chapters (Chapter 5), the introduction of potentials increases the complexity of the proof of soundness (Theorem 5.11). To streamline the proof, we introduce a lemma for easier manipulation of relaxations of resource demands.

For an arbitrary resource demand $(p_0, p_1) \in \mathcal{D}$ and $n, m \in \mathcal{R}$, one may ask if $(p_0 + n, p_1 + m) \succcurlyeq (p_0, p_1)$. As it turns out, this is true if and only if $n - m \geq 0$. This agrees with our intuition: Enforcing that $n - m \geq 0$ ensures that the resource consumption does not decrease - keeping relaxation constraints sound.

Lemma 2.16. Let $(p_0, p_1) \in \mathcal{D}$ and $n, m \in \mathcal{R}$. Then,

$$n - m \geq 0 \iff (p_0 + n, p_1 + m) \succcurlyeq (p_0, p_1)$$

Proof. We start by proving $n - m \geq 0 \implies (p_0 + n, p_1 + m) \succcurlyeq (p_0, p_1)$.

For the initial resources, $p_0 + n \geq p_0$ holds, as $n \in \mathcal{R}$. For the resource consumption, we get:

$$\begin{aligned} p_0 - n - (p_1 + m) &= p_0 - p_1 + (n - m) \\ &\geq p_0 - p_1 \end{aligned}$$

Where we use the assumption $n - m \geq 0$ from line one to two.

We continue with $(p_0 + n, p_1 + m) \succcurlyeq (p_0, p_1) \implies n - m \geq 0$. By assumption, we get that $p_0 + n - (p_1 + m) \geq p_0 - p_1$. Thus, $p_0 - p_1 + (n - m) \geq p_0 - p_1$, which shows that $n - m \geq 0$. $\square$

3 LetTick Language

In this chapter, we introduce our first programming language. This language will form a core, so that subsequent chapters extend this language incrementally. To this end, we define the type system, along with the associated evaluation semantics and type rules. The chapter then concludes with a proof of soundness, showing that the relation between evaluation semantics and type rules is consistent - the resource bounds assigned to types indeed provide a lower bound on the concrete resource consumption of evaluating an expression.

In order to elicit resource bounds for programs, we need two things: (1) A construct in the programming language that allows altering the resource demand and (2) Types that reflect the resource demand. The first problem is solved by introducing the instruction **tick** k, where k is an integer, indicating the amount of resources that are either consumed of freed. The second problem is solved by embellishing the return type with a resource demand.

3.1 Language Syntax

We begin by defining the programming language and the associated type system. Initially, the syntax for let expressions may seem obfuscated. We choose this syntax with later chapters in mind, to make extension as easy as possible. Generally, the literal after **let** is used to bind the result of e_1. For this chapter, we do not yet bind any literals - for now, we use _ as a placeholder.

Definition 3.1 (LetTick syntax).

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

Expressions inside the LetTick language are sequences of tick instructions. As an example, consider the programs below. In **??** we provide the associated type derivation and derive resource bounds for this program.

Example 3.2.

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

Intuitively, we have three distinct steps. The first one consumes 3 resources, the second one frees up 2 resources and the last one consumes 5 resources.

Definition 3.3 (LetTick types)**.**

$$\mathbb{T} := \text{Unit}$$

Note that the unit type has exactly one, unique value that we denote by $\bullet$. A specific constructor for $\bullet$ is not required, we can already generate this value. By writing **tick** 0, no resources are consumed and $\bullet$ is returned. Thus, it serves as a constructor for $\bullet$.

3.2 Evaluation Semantics

When evaluating an expression from Definition 3.1, we are interested in (1) The value that the expression evaluates to and (2) The resource demand required for evaluation. Both are defined by the evaluation rules below.

Before we can define evaluation rules, we need to introduce new notation. We denote by $\vDash e \overset{p_0\ p_1}{\rightsquigarrow} v$ the fact that the expression e evaluates to the value v, while requiring p_0 initial and p_1 residual resources. The definition of the *evaluation judgement* is then given by the evaluation rules.

First, we define a rule for **tick** k instructions. Such an instruction either frees resources or consumes them, if k is negative or positive, respectively. We model these cases as two evaluation rules.

$$(E : tick_+)\frac{k \geq 0}{\vDash \textbf{tick}\ k \overset{k\ 0}{\rightsquigarrow} \bullet} \qquad (E : tick_-)\frac{k < 0}{\vDash \textbf{tick}\ k \overset{0\ |k|}{\rightsquigarrow} \bullet}$$

Next, we need a rule that allows us to combine two expression into a let expression. The resulting resource demand is the composition of the resource demand of each sub-expression. For a definition of the $\triangleright$ operator, please refer to Definition 2.6.

$$(\text{E:let})\frac{\vDash e_1 \overset{p_0\ p_1}{\rightsquigarrow} \bullet \quad , \quad \vDash e_2 \overset{q_0\ q_1}{\rightsquigarrow} \bullet}{\vDash \textbf{let}\ \ = e_1\ \textbf{in}\ e_1 \overset{r_0\ r_1}{\rightsquigarrow} \bullet} \qquad \text{where } (r_0, r_1) = (p_0, p_1) \triangleright (q_0, q_1)$$

Definition 3.4 (Evaluation Judgement)**.** Let e be an expression of our programming language, let v be the value that is obtained from evaluating e. Furthermore, let $(p_0, p_1) \in \mathcal{D}$. Where p_0 are the initial resources needed to evaluate e, and p_1 resources are freed afterward.

$$\vDash e \overset{p_0 \; p_1}{\rightsquigarrow} v$$

In the next section, we introduce type rules that are tightly linked to the evaluation semantics defined now.

3.3 Type System

The evaluation judgement in the previous section defined our *ground truth*; That is, using the rules defined there, we can exactly calculate the resource-consumption of an arbitrary expression from Definition 3.1. The type rules introduced in this chapter will allow us to assign a type to an expression, where the resource-annotations of that type provide an **upper-bound** on the resource-consumption. As a result, type inference can be used to determine an upper bound on the resource consumption, without executing a program.

3.3.1 Resource-aware Type System

We start by introducing *resource-aware types*, which comprise two parts: the type assigned to an expression, together with a resource-pair representing the inferred resource consumption. To this end, we provide an ergonomic notation for resource-aware types:

Definition 3.5 (Resource-aware types)**.** Let $A \in \mathbb{T}$ and $(p_0, p_1) \in \mathcal{D}$. We define resource-aware types as tuples $(A, (p_0, p_1)) \in \mathbb{T} \times \mathcal{D}$. Most of the time we write $A^{p_0}_{p_1}$ for $(A, (p_0, p_1))$.

For this chapter, the only type that A can adopt is Unit. Therefore, resource-aware types have the form $\text{Unit}^{p_0}_{p_1}$ *for this chapter.*

3.3.2 Inference Rules

For let expressions, we compose resource demands using the sequencing operator $\triangleright$ introduced in Definition 2.6. Note that this rule matches its associated evaluation rule.

$$\text{(T:let)}\frac{\vdash\ e_1 : \text{Unit}^{p_0}_{p_1} \qquad \vdash\ e_2 : \text{Unit}^{q_0}_{q_1}}{\vdash\ \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 : \text{Unit}^{r_0}_{r_1}} \qquad (r_0, r_1) = (p_0, p_1) \triangleright (q_0, q_1)$$

The expression **tick** k consumes (or frees) k resources. We can type this expression with any resource demand (q_0, q_1) where $q_0 - q_1 \geq k$. Compared to the evaluation rules in the previous section, this rule is more generic - the resource demand (p_0, p_1) is not unique.

Let us elaborate on why this makes sense.

$$\text{(T:tick)}\frac{q_0 \geq k + q_1}{\vdash\ \textbf{tick}\ k : \text{Unit}^{q_0}_{q_1}}$$

Let us now perform an exemplary type derivation using the type rules we just introduced, to get a better understanding of how the definitions orchestrate together. For this, consider the program from Example 3.2. The resulting type derivation is displayed below.

$$\frac{\dfrac{q_0 \geq 3 + q_1}{\vdash\ \textbf{tick}\ 3 : \text{Unit}^{q_0}_{q_1}} \qquad \dfrac{\dfrac{t_0 \geq -2 + t_1}{\vdash\ \textbf{tick}\ -2 : \text{Unit}^{t_0}_{t_1}} \qquad \dfrac{s_0 \geq 5 + s_1}{\vdash\ \textbf{tick}\ 5 : \text{Unit}^{s_0}_{s_1}}}{\vdash\ \textbf{let}\ \_ = \textbf{tick}\ -2\ \textbf{in tick}\ 5 : \text{Unit}^{r_0}_{r_1}}}{\vdash\ \textbf{let}\ \_ = \textbf{tick}\ 3\ \textbf{in let}\ \_ = \textbf{tick}\ -2\ \textbf{in tick}\ 5 : \text{Unit}^{p_0}_{p_1}}$$

We are now left with the task of resolving the collected, numeric constraints. These are the collected constraints:

$$q_0 \geq 3 + q_1 \tag{3.1}$$
$$t_0 \geq t_1 - 2 \tag{3.2}$$
$$s_0 \geq 5 + s_1 \tag{3.3}$$
$$(r_0, r_1) = (t_0, t_1) \triangleright (s_0, s_1) \tag{3.4}$$
$$(p_0, p_1) = (q_0, q_1) \triangleright (r_0, r_1) \tag{3.5}$$

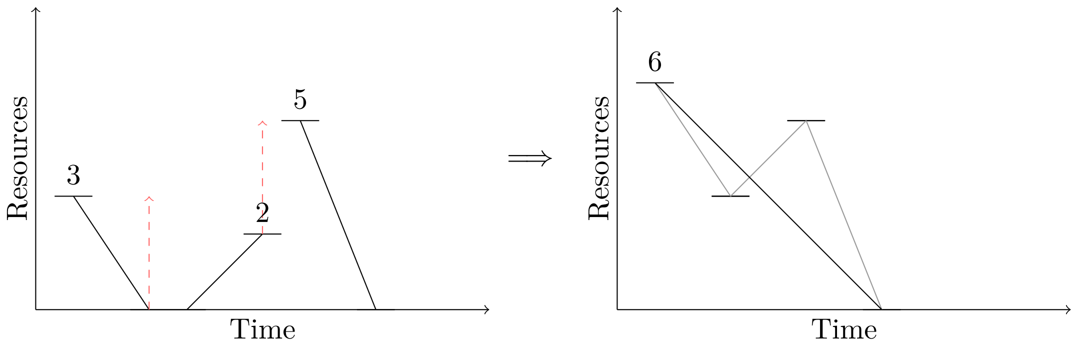
Eq. (3.1), Eq. (3.2) and Eq. (3.3) stem from applications of (T:Tick). Eq. (3.4) and Eq. (3.5) stem from applications of (T:Let). To derive a resulting resource demand, we must find a solution to the set of constraints above. For this specific instance, a valid solution would be:

$$\begin{aligned} q_0 &= 3 & q_1 &= 0 \\ t_0 &= 0 & t_1 &= 2 \\ s_0 &= 5 & s_1 &= 0 \end{aligned}$$

Thus, we can calculate (p_0, p_1) using Eq. (3.4) and Eq. (3.5). We get:

$$\begin{aligned} (r_0, r_1) &= (0, 2) \triangleright (5, 0) = (3, 0) \\ (p_0, p_1) &= (3, 0) \triangleright (3, 0) = (6, 0) \end{aligned}$$

As a visual aid, consider the associated resource diagram:



This completes our type derivation, as inserting the calculated resource-annotations yields our desired type.

$$\textbf{let } \_ = \textbf{tick } 3 \textbf{ in let } \_ = \textbf{tick } -2 \textbf{ in tick } 5 \ : \ \text{Unit}_0^6$$

Having a solid grasp of how the type rules orchestrate together, we now prove soundness for the LetTick programming language.

3.4 Soundness

Ultimately, we want our type rules to be consistent with the operational semantics defined previously. This will allow us to reason about the type of a program, while ensuring that the resource-annotations of the resulting type line up with the concrete resource consumption of the expression.

Let us first sketch the structure of our soundness proof:

If $\vdash e : \text{Unit}_{p_1}^{p_0}$ and $\vDash e \overset{q_0\ q_1}{\rightsquigarrow} \bullet$, then the following hold true: The result $\bullet$ of the expression e has a type that is consistent with the one assigned to e. The initial resources of the type are not exceeded in the concrete evaluation of the expression e. The resource consumption assigned to the type is not exceeded by the evaluation, either. We can state the same more precisely, by using the relaxation relation defined in Definition 2.9.

Theorem 3.6 (Soundness of typing for LetTick language)**.** Let e be an expression in our programming language, and let $(q_0, q_1), (p_0, p_1) \in \mathcal{R}$.

$$\text{If } \vdash e : \text{Unit}_{p_1}^{p_0} \text{ and } \vDash e \overset{q_0\ q_1}{\rightsquigarrow} \bullet, \text{ then } (p_0, p_1) \succcurlyeq (q_0, q_1).$$

Proof. For this proof, we use structural induction on the type derivation of the expression. This means, we prove soundness for every application of an inference rule, like so: We assume that the expression e has a form that allows the application of the current inference rule. Then, we show that the resource demand inferred by the inference rules is in fact a relaxation of the resource demand generated by the evaluation rules. In some cases, we make further assumptions regarding sub-expressions, for example.

(T:tick): If we can apply the rule (T:Let), the expression must be of the form $e :=$ **tick** q for some q. As q can either be positive or negative, which affects the application of evaluation rules, we make a case distinction.

Let $q \geq 0$. Then the following hold by assumption:

$$q \geq 0 \tag{3.6}$$
$$q_0 = q \tag{3.7}$$
$$q_1 = 0 \tag{3.8}$$
$$p_0 \geq q + p_1 \tag{3.9}$$

Where Eq. (3.7) and Eq. (3.8) follow from applying the associated evaluation rule (E:Tick+), and Eq. (3.9) follows from premise of the (T:Tick) rule.

Since $p_1 \geq 0$ by definition, using Eq. (3.9), $p_0 \geq q$ holds. Applying this inequality to Eq. (3.7), we get that $p_0 \geq q_0$. Which proves the first part. Next, we show

$p_0 - p_1 \geq q_0 - q_1$. Combining Eq. (3.9) with Eq. (3.7), we get $p_0 \geq q_0 + p_1$. Subtracting p_1 from both sides, we get $p_0 - p_1 \geq q_0$. Since $q_1 \geq 0$ by definition as well, it especially holds that $p_0 - p_1 \geq q_0 - q_1$. Which concludes the proof for the case of $q \geq 0$.

For the second case, we assume that $q < 0$. Then, the following hold:

$$q < 0 \tag{3.10}$$
$$q_1 = q \tag{3.11}$$
$$q_0 = 0 \tag{3.12}$$

Furthermore, Eq. (3.9) also holds true. By definition $p_0 \geq 0$ holds. Using 3.12, we get $p_0 \geq q_0$.

Subtracting p_1 from both sides in 3.9, we get $p_0 - p_1 \geq q$. Applying Eq. (3.11), we get $p_0 - p_1 \geq q_1$. Because $q_1 \geq 0$ by definition, it also holds that $p_0 - p_1 \geq 0 - q_1$. Using 3.12, we get $p_0 - p_1 \geq q_0 - q_1$, concluding our proof for tick.

(T:Let): If (T:Let) was applied, the expression has the form $e := \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2$. Our induction hypothesis includes judgements, and soundness, for the sub-expressions e_1, e_2.

Soundness for e_1 and e_2 gives the following assumptions:

$$\begin{array}{ll}
\vdash\ e_1 : \mathrm{Unit}^{p_0}_{p_1} & (p_0, p_1) \succcurlyeq (m_0, m_1) \\
\vDash e_1 \overset{m_0\ m_1}{\rightsquigarrow} v_1 & (q_0, q_1) \succcurlyeq (n_0, n_1) \\
\vdash\ e_2 : \mathrm{Unit}^{q_0}_{q_1} & \vDash e_2 \overset{n_0\ n_1}{\rightsquigarrow} v_2
\end{array}$$

The premise of (T:Let) further yields the following, additional, assumptions:

$$\begin{array}{ll}
(s_0, s_1) = (m_0, m_1) \triangleright (n_0, n_1) & (r_0, r_1) = (p_0, p_1) \triangleright (q_0, q_1) \\
\vdash\ \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 : \mathrm{Unit}^{r_0}_{r_1} & \vDash \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 \overset{s_0\ s_1}{\rightsquigarrow} v_2
\end{array}$$

To prove soundness for the let expression, we need to prove $(r_0, r_1) \succcurlyeq (s_0, s_1)$. First, we prove soundness for the resulting resource consumption:

$$\begin{aligned}
s_0 - s_1 &= m_0 - m_1 + \max(m_1, s_0) - (n_1 - n_0 + \max(m_1, s_0)) && (3.13)\\
&= m_0 - m_1 + n_0 - n_1 + \max(m_1, s_0) - \max(m_1, s_0) && (3.14)\\
&= m_0 - m_1 + n_0 - n_1 && (3.15)\\
&\leq p_0 - p_1 + q_0 - q_1 && (3.16)\\
&= p_0 - p_1 + \max(p_1, q_0) + q_0 - q_1 - \max(p_1, q_0) && (3.17)\\
&= \underbrace{p_0 - p_1 + \max(p_1, q_0)}_{=r_0} - \underbrace{(q_1 - q_0 + \max(p_1, q_0))}_{=r_1} && (3.18)
\end{aligned}$$

Where Eq. (3.16) uses $(p_0, p_1) \succcurlyeq (m_0, m_1), (q_0, q_1) \succcurlyeq (n_0, n_1)$. We have proven $r_0 - r_1 \geq s_0 - s_1$. Next, we prove $r_0 \geq s_0$ by means of case distinction.

To prove soundness for the initial resources, we make a case distinction. We distinguish between $n_0 \geq m_1$ and $n_0 < m_1$.

Let $n_0 \geq m_1$.

$$\begin{aligned}
s_0 &= m_0 - m_1 + \max(m_1, n_0) && (3.19)\\
&= m_0 - m_1 + n_0 && (3.20)\\
&\leq p_0 - p_1 + n_0 && (3.21)\\
&\leq p_0 - p_1 + q_0 && (3.22)\\
&\leq p_0 - p_1 + \max(p_1, q_0) && (3.23)\\
&= r_0 && (3.24)
\end{aligned}$$

Where Eq. (3.21) and Eq. (3.22) are justified by $(p_0, p_1) \succcurlyeq (m_0, m_1)$ and $(q_0, q_1) \succcurlyeq (n_0, n_1)$, respectively. Next, we prove that $r_0 - r_1 \geq s_0 - s_1$.

Let $n_0 < m_1$.

$$\begin{aligned}
s_0 &= m_0 - m_1 + \max(m_1, n_0) && (3.25)\\
&= m_0 - m_1 + m_1 && (3.26)\\
&= m_0 && (3.27)\\
&\leq p_0 && (3.28)\\
&\leq p_0 - p_1 + \max(p_1, q_0) = r_0 && (3.29)
\end{aligned}$$

Where Eq. (3.28) uses $(p_0, p_1) \succcurlyeq (m_0, m_1)$, and Eq. (3.29) uses the fact that $\max(p_1, q_0) - p_1 \geq 0$ for any p_1, q_0.

This concludes our proof of soundness. $\square$

4 VarTick Language

Before continuing, let us examine the limitations of the LetTick syntax from the previous chapter. We are not able to declare variables, control flow, or constants. All of those features will be implemented in this chapter. In order to declare and use variables in our programming language, we need to provide a definition of an environment, which comprises the concrete values of variables. As we will see, introducing variables will demand cautious reformulation of some evaluation rules to account for environments.

Similarly to environments, we will introduce contexts for the type rules. While an environment maps variables to concrete values, a context will map variables to types. This will also require carefully embellishing previous type rules, to provide rules that are rigorous with respect to contexts. We define the syntax of the augmented programming language below.

Before we can define the syntax, we have to define the set of variable names (Var). For our language, valid variable names are those that comprise only letters and numbers, without whitespace.

Definition 4.1 (VarTick syntax)**.**

$$\begin{array}{rll}
e := & \textbf{let } x = e_1 \textbf{ in } e_2 & \text{(let)} \\
& \textbf{tick } k & \text{(tick)} \\
& x & \text{(var)} \\
& \textbf{true} \mid \textbf{false} & \text{(bool)} \\
& k & \text{(integers)} \\
& \text{where } x \in \text{Var and } k \in \mathbb{Z} &
\end{array}$$

The newly introduced constructors for integers and booleans are reflected in the type system for the language:

Definition 4.2 (VarTick types)**.**

$$\mathbb{T} := \text{Unit} \mid \text{Bool} \mid \text{Int}$$

In order to give a sound definition of the set of values (Vals), we define the set of values for each type respectively.

$$\begin{aligned}\mathrm{Vals}_{\mathrm{Unit}} &= \{\bullet\}\\ \mathrm{Vals}_{\mathrm{Bool}} &= \{\mathbf{true}, \mathbf{false}\}\\ \mathrm{Vals}_{\mathrm{Int}} &= \mathbb{Z}\end{aligned}$$

The set of *all* possible values is simply the disjoint union of the set of values of each type. Note that $\sqcup$ denotes the disjoint union of sets.

Definition 4.3. $\mathrm{Vals} = \mathrm{Vals}_{\mathrm{Unit}} \sqcup \mathrm{Vals}_{\mathrm{Bool}} \sqcup \mathrm{Vals}_{\mathrm{Int}}$

We also want to express that a variable has a specific type, which is the same as claiming that the variable is in the set Vals_A for the appropriate type A. Thus, we write $v : A$ to denote $v \in \mathrm{Vals}_A$.

4.1 Evaluation Semantics

Expressions now feature variables, necessitating the introduction of *environments* to resolve variables to their associated value. In Section 4.1.1 we define environments, afterward we introduce evaluation rules in Section 4.1.2.

4.1.1 Environments

Environments map variable names to values. Here, Var is the set of variable names, and Vals is the set of values.

Definition 4.4 (Environment)**.** An environment $E : \mathrm{Var} \rightharpoonup \mathrm{Vals}$ is a partial function.

For an environment E, we may be interested in the set of variable names that are defined. This is called the *domain* of E, and we denote it by $\mathbf{dom}(E)$. As some evaluation rules require the existence of certain variables, we use the domain to check variable names for membership. For this, we write $x \in \mathbf{dom}(E)$ for $x \in$ Var.

Definition 4.5 (Domain of environments)**.** Let E be an environment. The *domain* $\mathbf{dom}(E)$ is the set of all variable names that are defined in E.

In most programming settings, we start with an empty environment and incrementally populate it with variables as our program evaluates. This constitutes the need for extending environments - we want to add variables to the environment upon introducing them.

Definition 4.6. Let E be an environment, $x \in \mathrm{Var}$ and $v \in \mathrm{Vals}$ be the value assigned to x. We write $E[x \mapsto v]$ for the environment E, where x is *additionally* mapped onto v.

Having all of these definitions in place, we now update the definition of an *evaluation judgement.* Previously, expression were evaluated as is, as no environment influenced the execution of the expression. Now, we consider evaluating expression against an environment E.

Definition 4.7 (Evaluation Judgement with Environments)**.** Let E be an Environment, e an expression of our programming language (Definition 4.1), v the value obtained from evaluating e, and let $(p_0, p_1) \in \mathcal{R}$. We then write the following for the evaluation judgement with respect to an environment:

$$E \vDash e \overset{p_0\ \ p_1}{\rightsquigarrow} v$$

Remark. Note that the evaluation of an expression e depends on the environment, due to the ambiguity of variable names. Thus, both the return value v and the resource demand (p_0, p_1) can differ for discrete environments.

4.1.2 Evaluation Rules

Having all the necessary definitions at hand, let us now define evaluation rules for our programming language. We start with the rule (E:var), which provides an explicit handle for resolving variable names to their values. In order for the rule to be consistent, we must enforce that the variable we want to resolve is defined in the current environment.

$$\text{(E:var)}\frac{x \in E}{E \vDash x \overset{0\ \ 0}{\rightsquigarrow} E(x)}$$

Next, we define one rule (E:bool) that unifies the evaluation of both boolean constants. Similarly, we define the evaluation of constant integers using the rule (E:Int). In contrast to the previous rule, we do not need to check the environment for existence of any variables, as both booleans and integers are evaluated in a direct manner.

$$\text{(E:bool)}\dfrac{b \in \{\mathbf{true}, \mathbf{false}\}}{E \vDash b \overset{0\ 0}{\rightsquigarrow} b} \qquad \text{(E:int)}\dfrac{n \in \mathbb{Z}}{E \vDash n \overset{0\ 0}{\rightsquigarrow} n}$$

Next, we introduce the evaluation rule for **tick**. Here, the concrete environment has no impact on the evaluation of a tick expression, it is evaluated *as is*.

$$\text{(E:tick+)}\dfrac{k \geq 0}{E \vDash \mathbf{tick}\ k \overset{k\ 0}{\rightsquigarrow} \bullet} \qquad \text{(E:tick-)}\dfrac{k < 0}{E \vDash \mathbf{tick}\ k \overset{0\ |k|}{\rightsquigarrow} \bullet}$$

The last rule missing is (E:let). Here, we need to carefully weave the value that is the result of the first expression into the second expression. We do so by augmenting the environment using the variable name and result of the preceding sub-expression.

$$\text{(E:let)}\dfrac{E \vDash e_1 \overset{p_0\ p_1}{\rightsquigarrow} v_1 \qquad E[x \mapsto v_1] \vDash e_2 \overset{q_0\ q_1}{\rightsquigarrow} v_2}{E \vDash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \overset{r_0\ r_1}{\rightsquigarrow} v_2}$$

$$\text{where } (r_0, r_1) = (p_0, p_1) \triangleright (q_0, q_1)$$

4.2 Type Rules

To type expressions with variables, knowledge of a variable's type is paramount. This necessitates the definition of *contexts* (Definition 4.8), along with ergonomic notations. The introduction of contexts requires extending *typing judgements* (Definition 4.11). With these in place, we define the *inference rules* (Section 4.2.2) that are context-aware.

4.2.1 Contexts

What environments are for evaluation rules, contexts are for inference rules. Contexts allow us to map variables to types, which is necessary since introducing variables. Similar to Environments, a context is a partial function; Conversely to environments, a context maps variable names to a type from Definition 4.2.

Definition 4.8 (Context)**.** We define a context $\Gamma : \mathrm{Var} \rightharpoonup \mathbb{T}$ as a partial function from variable names to types.

Most definitions for contexts are similar to the definitions for environments. For example, we define the *domain* of a context to comprise all variables that are defined.

Definition 4.9 (Domain of Context)**.** Given a context Γ, the domain $\mathbf{dom}(\Gamma)$ is the set of all variables that are defined in Γ.

To assert that a variable x is defined in the current context Γ, we write $x \in \mathbf{dom}(\Gamma)$.

While executing a program, multiple variables with the name x may be created at discrete times. It is important to note that the *most recent* initialization will determine the value of $\Gamma(x)$ - for let expressions, this can lead to inconsistencies. *Disjoint* contexts circumvent this problem, as two disjoint contexts do not share any variables, eliminating the chance for inconsistencies.

Definition 4.10 (Disjoint contexts)**.** Let Γ_1, Γ_2 be contexts.

$$\Gamma_1 \text{ and } \Gamma_2 \text{ are disjoint} :\Longleftrightarrow \mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) = \emptyset$$

At the start of executing a program the context is empty. During execution, variables are initialized; This requires extending the context to account for the newly initialized variable. Given that $x : A$ we add x to Γ by writing $\Gamma\ ;\ x : A$. However, we require that $x \notin \mathbf{dom}(\Gamma)$. In a sense we require that Γ and the context containing only x are *disjoint*.

With contexts defined, we embed them into our new definition of a *typing judgement*.

Definition 4.11 (Typing Judgement with contexts)**.** Let Γ be a context, e an expression of Definition 4.1 with the return type A, and let $(p_0, p_1) \in \mathcal{R}$. We then denote a typing judgement with respect to the context Γ by writing:

$$\Gamma\ \vdash\ e : A^{p_0}_{p_1}$$

Remark. Similarly to Definition 4.7, the resource-aware type assigned to e will vary on the context in which e is evaluated.

4.2.2 Inference Rules

Let us start by introducing the rule (T:var), which allows us to add a variable and its associated type into a context.

$$\text{(T:var)}\frac{}{\Gamma; x : A\ \vdash\ x : A}$$

Next, we introduce two rules, one for boolean constants and one for integer constants. The rule (T:bool) assigns a type Bool to any occurrence of **true** or **false**, and similarly, the rule (T:int) assigns a type Int to any occurrence of integer constants.

$$\text{(T:bool)}\frac{b \in \{\textbf{true}, \textbf{false}\}}{\Gamma \ \vdash \ b : \text{Bool}} \qquad \text{(T:int)}\frac{n \in \mathbb{Z}}{\Gamma \ \vdash \ n : \text{Int}}$$

Defining the type rule for the **tick** is straightforward, we add a context Γ to the rule from the previous chapter Section 3.3.2.

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

Remark. As the sub-expressions of the let expression might have different contexts, we need to account for that as well. To permit a consistent formulation of the type rule, we need to add an assumption: the two contexts need to be *disjoint*. This will allow us to avoid any typing inconsistencies that could arise from mismatched types between the two contexts. For this, a variable x could be present in both contexts Γ_1 and Γ_2, with different types assigned to the variable:

$$\Gamma_1 \ \vdash \ x : A \qquad \Gamma_2 \ \vdash \ x : B$$

This is prevented, by demanding that the contexts are *disjoint*. While this makes the rule less flexible, we avoid cases where the application of (T:let) would yield a possibly ill-typed judgement; If the variable x is present in both contexts, altering the type of that variable may construe the resulting typing of the expression.

This leads to the following formulation of the rule (T:Let). Compared to the previous chapter, we now require the contexts to be disjoint, and propagate the type of e_1 to the context Γ_2 to type e_2. Previously, no variables were acrued and thus no propagation was needed.

$$\text{(T:let)}\frac{\Gamma_1 \ \vdash \ e_1 : A^{p_0}_{p_1} \qquad \Gamma_2; x : A \ \vdash \ e_2 : B^{q_0}_{q_1} \qquad \Gamma_1 \cap \Gamma_2 = \emptyset}{\Gamma_1; \Gamma_2 \ \vdash \ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : B^{p_0}_{q_1}}$$

4.3 Soundness

Having introduced Environments and Contexts, it is necessary to formulate the soundness theorem as to account for environments and contexts. This yields the following statement:

Theorem 4.12 (Soundness of typing for var-tick language)**.** Let E be an environment, Γ be a context. Further, let A be a type, $(p_0, p_1), (q_0, q_1) \in \mathcal{R}$ and an expression e that evaluates to the value v. The Soundness Theorem states:

$$\text{If } E \vDash e \overset{p_0 \ p_1}{\rightsquigarrow} v \text{ and } \Gamma \ \vdash \ e : A^{q_0}_{q_1}, \text{ then } v : A \text{ and } (q_0, q_1) \succcurlyeq (p_0, p_1).$$

Proof. The proof uses the same structure from the previous chapter (Theorem 3.6). We perform structural induction over the type derivation.

The proof given in Theorem 3.6 still holds for the rules (T:Tick) and (T:Let).

(T:Bool): We know that $p_0 = p_1 = 0 = q_0 = q_1$ by definition of the evaluation and typing rule. As a result $p_0 \geq q_0$ and $p_0 - p_1 \geq q_0 - q_1$, which is identical to $(p_0, p_1) \succcurlyeq (q_0, q_1)$. It furthermore follows that the type of the resulting value b is of type Bool, and such, consistent with the typing derived from the type rule.

(T:Int): Identical to the previous proof, we get that $p_0 = p_1 = 0 = q_0 = q_1$ which shows $(p_0, p_1) \succcurlyeq (q_0, q_1)$, and similarly, the type of n is Int, which is consistent.

□

5 ListTick Language

In this chapter, we introduce the concept of lists as a fundamental data type. This introduction leads to the development of several new definitions, with the most significant among them being potentials. In our previous approach, resource annotations were attached to a type, resulting in a resource cost that followed a "What-you-see-is-what-you-get" principle. However, when dealing with lists, a different approach becomes necessary. Instead of a constant cost for any input list, we assign a specific cost to each individual element within the list. This adaptation allows us to manage resource consumption while accommodating the variable length of lists. The concept of potentials plays a pivotal role in making this adjustment feasible. Beyond our exploration of lists, we delve into function application, which inherently introduces recursion to our programming language.

Definition 5.1 (ListTick syntax)**.**

$$
\begin{aligned}
e := \quad & \textbf{let } x = e_1 \textbf{ in } e_2 && \text{(let)} \\
& \textbf{tick } k && \text{(tick)} \\
& x && \text{(var)} \\
& \textbf{true} \;\mid\; \textbf{false} && \text{(bool)} \\
& k && \text{(integer)} \\
& \mathit{head} :: \mathit{tail} \;\mid\; \textbf{nil} && \text{(list constructor)} \\
& \textbf{match } l \textbf{ with} \mid \textbf{nil} \to e_1 \mid x :: xs \to e_2 && \text{(match list)} \\
& \textbf{letfun } f = x \to e_f \textbf{ fin } e_2 && \text{(function declaration)} \\
& f\ x && \text{(function application)} \\
& \text{where } x \in \text{Var},\ k \in \mathbb{Z},\ \mathit{head} \in A,\ \mathit{tail} \in \text{List}(A)
\end{aligned}
$$

Because lists can contain *any type* of data, assuming it is homogenous, they are an example of a *polymorphic type.* We can think of polymorphic types as functions from types to types, where the input is called a *type variable* and defines the concrete type of list.

Definition 5.2 (ListTick types)**.**

$$A, B := \text{Unit} \mid \text{Bool} \mid \text{Int} \mid \text{List}(A) \mid A \to B$$

In order to define the set of values for list types, we start by defining the set of values for lists of length n, which we denote by $\text{List}_n(A)$.

Definition 5.3 (Lists of Length n)**.** $\text{List}_n(A)$ is the set of all lists of type A with a length of n.

The set of values for $\text{List}(A)$ comprises lists of every possible length n. Thus, it is the union of $\text{List}_n(A)$ for every $n \in \mathbb{N}$.

Definition 5.4 (Vals for List(A))**.**

$$\text{Vals}_{\text{List}(A)} = \bigcup_{n \in \mathbb{N}} \text{List}_n(A)$$

We want to infer resource-bounds for programs. In order for the inference to be computationally tractable, we constrain the use of certain language features. Most notably, we enforce that functions on lists only recur on the tail of the list. As a result, recursive function calls *will terminate*, eventually.

While this reduces the set of programs that we permit, most problems on lists naturally have a form that matches this constraint. For example, folding operations such as sum (Fig. B.2), or functorial operations such as map (Fig. B.3), can be defined in this way.

5.1 Evaluation Semantics

We introduce evaluation rules, which are nuanced when evaluating lists, as the resource demand can be related to the length of the list. This necessitates the introduction of resource demands as *recurrence relations*. We start by introducing recurrence relations and motivate the connection to resource demands.

5.1.1 Recurrence Relations As Resource Demands

Since we introduced lists and recursion into our programming language, this has a profound impact on the relation between input and resource demands. Previously the values of any type in our type system had no *structural implication* for resource demands. As lists can inhabit any (finite) length, we need to find a general approach to relating lists with resource demands.

Let us build an intuition for the interplay of resource demands and lengths of lists, by informally working through an example function. Consider the function $add1$ in Fig. B.1, which increments every entry in the list.

Our goal is to represent the resource demand Q of a list l as a function of the tail of the list. We want to derive an equation of the form $Q(x :: xs) = \alpha + Q(xs)$, assigning a cost to an element x of the list. To ensure that this equation produces a well-defined result, we need to derive a cost $Q(\mathbf{nil})$ as well.

Given a non-empty list l, we execute the body of the second match condition. There, we tick and increment the head of the list. Afterward we call the function recursively on the tail. Thus, we have paid 2 resources for the head of the list, plus the cost of calling the function on the tail, which is not yet determined. This yields an equation of the form

$$Q(x :: xs) = 2 + Q(xs)$$

To derive a complete recurrence relation, and thus a resource demand for arbitrary, finite lists, we need to derive the cost $Q(\mathbf{nil})$. In our example, we simply return nil with no associated cost. Thus, we have $Q(\mathbf{nil}) = 0$. Using both equations we define the cost of a list l:

$$Q(l) = \begin{cases} 2 + Q(xs) & \text{if l = x :: xs} \\ 0 & \text{else} \end{cases}$$

Understanding this example, we define recurrence relations. Notably, we constrain recurrence relations to have *order* 1, as we only allow recursion on the tail of a list.

5.1.2 Evaluation Rules

We start by introducing two rules for the construction of lists, corresponding to the two constructors of our programming language. The construction of empty and non-empty lists has no associated cost. As a result, the cost of operations on lists is purely dictated by the actions performed on the list.

$$\text{(E:Nil)}\frac{}{E \vDash \mathbf{nil} \overset{0\ 0}{\rightsquigarrow} \bullet} \qquad \text{(E:Cons)}\frac{x, xs \in E}{E \vDash x :: xs \overset{0\ 0}{\rightsquigarrow} (E(x), E(xs))}$$

In order to implement functions that work recursively on a list, we need pattern matching. The two cases of the pattern match are defined by the inductive definition of lists: (1) A list is nil, or (2) A list comprises head and tail. We introduce one evaluation rule for each case respectively. The first rule matches an empty list.

$$\text{(E:MatchNil)}\,\frac{E(l) = \bullet \qquad E \vDash e_1 \overset{q_0\ q_1}{\rightsquigarrow} v}{E \vDash \textbf{match } l \textbf{ with } \mid \textbf{nil} \rightarrow e_1 \mid x :: xs \rightarrow e_2 \overset{q_0\ q_1}{\rightsquigarrow} v}$$

Next, we define an evaluation rule for non-empty lists. Here, we need to extract the resource demand as a recurrence relation, which we construct similar to Section 5.1.1.

$$\text{(E:MatchCons)}\,\frac{E(l) = (v_x, v_{xs}) \qquad E[x \mapsto v_x,\ xs \mapsto v_{xs}] \vDash e_2 \overset{p_0\ p_1}{\rightsquigarrow} v}{E \vDash \textbf{match } l \textbf{ with } \mid \textbf{nil} \rightarrow e_1 \mid x :: xs \rightarrow e_2 \overset{p_0\ p_1}{\rightsquigarrow} v}$$

To properly evaluate recursive functions we introduce an evaluation rule for function application. We define e_f as the body of the function f, and the variable name y_f as the argument used in the definition of f.

$$\text{(E:Application)}\,\frac{E(f) = e_f \quad E[y_f \mapsto E(x)] \vDash e_f \overset{q_0\ q_1}{\rightsquigarrow} v}{E \vDash f\ x \overset{q_0\ q_1}{\rightsquigarrow} v}$$

In order to apply a function, it needs to be defined. The rule (E:LetFun) reads the function name, argument name and the body of the function. It then augments the environment, *without executing the function body.* In later expressions the function identifier is available and may be used, incurring the cost for executing the function body then.

$$\text{(E:LetFun)}\,\frac{E[f \mapsto e_f] \vDash e_{\text{next}} \overset{p_0\ p_1}{\rightsquigarrow} v}{E \vDash \textbf{letfun } f = x \rightarrow e_f \textbf{ fin } e_{\text{next}} \overset{p_0\ p_1}{\rightsquigarrow} v}$$

Note that (1) Function declaration itself has no cost assigned to it (2) We use the keyword **fin** to denote the end of the function body. This serves a similar purpose to the keyword **in** from let-expressions, as subsequent expression can be written afterward. A key reason for introducing **fin** like this stems form the need to distinguish sequential expressions inside the function body e_f and the end of function declaration. If both used the keyword **in** this distinction becomes ambiguous.

5.2 Type System

Our goal is to introduce type rules that allow us to assign resource costs to lists. Previously, we assigned a resource-pair (Definition 2.2) to the resulting type; This is not feasible for lists, as the cost of evaluating an expression depends on the length of the list. *Potentials* are the analogue to recurrence relations, allowing us to assign costs to every list element.

5.2.1 Resource-Aware Lists

Resource-aware lists have a *potential* assigned to them, which signifies the cost *per list entry.* The potential attributed to a list solely depends on how the list is used; Thus, potentials are of interest in function signatures to signify cost.

Definition 5.5 (Resource-aware list)**.** Let $p \in \mathbb{N}$, and let A be a type from Definition 5.2. A resource-aware list type is a pair comprising the *base type* and *potential*:

$$\text{List}^p(A) = (\text{List}(A), p)$$

Remark. Resource demands and potentials can coexist; Resource demands capture static cost for executing an expression. Potentials capture dynamic cost that depends on the length of the list. A function $f :: A \to \text{List}^1(B)^2_3$, for example, indicates two facts (1) Every list entry costs 1 resource and (2) Calling the function requires 2 initial and 3 residual resources, irrespective of the concrete list.

Given a concrete list that is typed with some potential p, we want to calculate the exact potential residing in the list. There are two constituents: the type parameter $A \in \mathbb{T}$ and the potential p. The type parameter A defines how much potential a value $v : A$ of the list has, whereas the potential p is the *additional* overhead assigned to every list element - which is specific to the usage of the list.

For any type other than A within the set of types, we assign a potential of 0. In more intricate type systems that include custom types, it is possible to assign non-zero potentials.

Definition 5.6 (Potential Function)**.** For a value v of type A, we define the potential of that value as follows:

$$\Phi(v : A) = \begin{cases} \Phi(x : B) + \Phi(xs : \text{List}^q(B)) + q & A = \text{List}^q(B) \\ 0 & \text{else} \end{cases}$$

To allow for more compact notations of potentials, we introduce $|l|$ to denote the length of a list l.

Definition 5.7 (Length of lists)**.** Given a list $l : \text{List}(A)$, we denote by $|l|$ the *length* of the list.

As most types $A \in \mathbb{T}$ have no potential assigned to them, the potential of a list $l : \text{List}^p(A)$ is dictated *solely* by its length and p; Only if the lists' type parameter is of type $\text{List}^q(B)$ do values in the list have inherent potential.

Corollary 5.8 (Potential of lists)**.** Let $l : \text{List}^q(A)$, where $A \in \mathbb{T}$, s.t. $\Phi(v : A) = 0$, then:

$$\Phi(l : \text{List}^q(A)) = |l| \cdot q$$

Proof. Using the definition of the potential function, we get:

$$\begin{aligned}\Phi(l : \text{List}^p(A)) &= \underbrace{\sum_{x \in l} \Phi(x : A)}_{=0} + \sum_{x \in l} q \\ &= |l| \cdot q\end{aligned}$$

$\square$

With potentials defined for $A \in \mathbb{T}$, we naturally extend potentials to contexts. Intuitively, the potential of a context comprises the potentials of every variable in the contexts' domain (see Definition 4.9). This notion will be central to our new definition of soundness in Section 5.3.

Definition 5.9 (Potential of Contexts)**.** Let Γ be a context.

$$\Phi(\Gamma) = \sum_{x \in \mathbf{dom}(\Gamma)} \Phi(x : \Gamma(x))$$

5.2.2 Inference Rules

We start by introducing two rules associated to the two list constructors. An empty list can be constructed freely, with no constraints on its potential. On the other hand, constructing a list from an element and another, compatible, list requires us to propagate the potential. The tail xs of the list is assumed to have potential p, as result, the list $x :: xs$ also has potential p.

$$\text{(T:Nil)}\frac{}{\Gamma \ \vdash \ nil : \text{List}^p(A)} \qquad \text{(T:Cons)}\frac{}{\Gamma; x : A; xs : \text{List}^p(A) \ \vdash \ cons(x, xs) : \text{List}^p(A)^p_0}$$

The potential p of the list l is the initial resource cost, specific to destructuring the list into head and tail. The expression e_1 has a resource demand of (q_0, q_1), whereas the evaluation of e_2 has a resource demand of $(q_0{+}p, q_1)$. Thus, the initial cost for an element of the list is precisely p. Furthermore, (q_0, q_1) represents a static cost that is paid regardless of the concrete list. We attach (q_0, q_1) to the return type. Similar to (T:Let) from previous chapters, we require that $l \notin \Gamma$, as this can otherwise lead to undesired side-effects.

$$\text{(T:MatchList)}\frac{\Gamma \vdash e_1 : B^{q_0}_{q_1} \qquad \Gamma; x : A; xs : \mathrm{List}^p(A) \vdash e_2 : B^{q_0+p}_{q_1}}{\Gamma; l : \mathrm{List}^p(A) \vdash \textbf{match } l \textbf{ with} \mid \textbf{nil} \to e_1 \mid x :: xs \to e_2 : B^{q_0}_{q_1}}$$

For function application, the input type needs to align with the variables that are applied. The rule (T:Application) allows us to type recursive functions. Most importantly, the type of x and the input type of f are then inferred to be the same. For recursive calls on a list, we assume that they recur only on the tail xs. Then, the tail has a potential annotation which is propagated to the function signature, as we assume that the types match.

$$\text{(T:Application)}\frac{\Gamma \vdash f : A \to B^{q_0}_{q_1}}{\Gamma; x : A \vdash fx : B^{q_0}_{q_1}}$$

Remark. We denote resource-aware functions just like any other resource-aware type, by adding a resource demand. In the case of functions, this looks like $A \to B^{p_0}_{p_1}$. Recall from Definition 3.5 that $A \to B^{p_0}_{p_1} = (A \to B, (p_0, p_1))$, where the resource demand represents the cost for evaluating the function.

The final type rule that we introduce is (T:LetFun). Informally, this rule adds the type of the function f to the context under which e_2 is evaluated. Of course, e_2 may use f, but only the resource-aware type of e_2 is of interest to us.

$$\text{(T:LetFun)}\frac{\Gamma \vdash f : A \to B \qquad \Gamma; f : A \to B \vdash e_2 : C^{p_0}_{p_1}}{\Gamma \vdash \textbf{letfun } f = x \to e_1 \textbf{ fin } e_2 : C^{p_0}_{p_1}}$$

5.2.3 Example Type Derivations

We introduce examples of varying complexity, to illustrate the nuances of type derivation for functions and lists and acquaint the reader to the newly introduced inference rules.

Example 5.10. For our first example, consider the function $add1$ that increments every integer in a list. The code can be found in Fig. B.1. We derive a resource-annotated type stepwise, for sake of illustration.

Step I: Application of (T:MatchList) yields two premises. We close the first premise by applying (T:Nil).

$$\text{(T:MatchList)}\frac{\text{(T:Nil)}\dfrac{}{\Gamma \vdash l : \mathrm{List}^p(A)} \qquad \Gamma; x : A; xs : \mathrm{List}^p(A) \vdash e_1 : B^{p_0}_{p_1}}{\Gamma; l : \mathrm{List}^p(A) \vdash \textbf{match } l \textbf{ with} \mid \textbf{nil} \to \textbf{nil} \mid x :: xs \to e_1 : B}$$

For brevity and readability, we denote by e_1 the function body that is executed on a match to cons. Furthermore, we use e_2 & e_3 to denote sub-expressions.

$$e_1 = \textbf{let}\ \_ = \textbf{tick}\ 1\ \textbf{in}\ e_2$$

Step II: Applying (T:Let) splits e_1 up into **tick** 1 and the remaining expression, abbreviated e_2. Applying (T:Tick) closes the left branch of the derivation, allowing us to assign a type to the expression **tick** 1.

$$\text{(T:Let)}\ \dfrac{\text{(T:Tick)}\ \dfrac{}{\Gamma\ \vdash\ \textbf{tick}\ 1 : \mathrm{Unit}_0^1} \qquad \Gamma; x : A; xs : \mathrm{List}^p(A)\ \vdash\ e_2 : B_{q_1}^{q_0}}{\Gamma; x : A; xs : \mathrm{List}^p(A)\ \vdash\ \textbf{let}\ \_ = \textbf{tick}\ 1\ \textbf{in}\ e_2 : B_{p_1}^{p_0}}$$

$$\text{Where } e_2 = \textbf{let}\ x' = x + 1\ \textbf{in}\ e_3$$
$$\text{and } (p_0, p_1) = (1, 0) \triangleright (q_0, q_1) \tag{5.1}$$

Step III: Again, we apply (T:Let), yielding $x + 1$ and the sub-expression e_3. Applying the rule (T:Op), we infer that $x : \mathrm{Int}$. Thus, $xs : \mathrm{List}^p(\mathrm{Int})$ as well.

$$\text{(T:Let)}\ \dfrac{\text{(T:Op)}\ \dfrac{}{\Gamma; x : \mathrm{Int}\ \vdash\ x + 1 : \mathrm{Int}} \qquad \Gamma; x' : \mathrm{Int}; xs : \mathrm{List}^p(\mathrm{Int})\ \vdash\ e_3 : B_{q_1}^{q_0}}{\Gamma; x : A; xs : \mathrm{List}^p(A)\ \vdash\ \textbf{let}\ x' = x + 1\ \textbf{in}\ e_3 : B_{q_1}^{q_0}}$$

$$\text{Where } e_3 = \textbf{let}\ xs' = \mathit{add}1\ xs\ \textbf{in}\ x' :: xs'$$

Step IV: The final application of (T:Let) generates two premises. The first being the recursive function call on xs, and the second being the construction of the final list. We infer that $B = \mathrm{List}^q(\mathrm{Int})$, resulting from the application of (T:Cons), from which the return type is propagated.

Furthermore, $q_0 = 0 = q_1$, as neither sub-expressions have any resource demand.

$$\text{(T:Let)}\ \dfrac{\Gamma; xs : \mathrm{List}^p(\mathrm{Int})\ \vdash\ \mathit{add}1\ xs : \mathrm{List}^q(\mathrm{Int}) \qquad \text{(T:Cons)}\ \dfrac{}{\Gamma; x' : \mathrm{Int}; xs' : \mathrm{List}^q(\mathrm{Int})\ \vdash\ x' :: xs' : \mathrm{List}^q(\mathrm{Int}}}{\Gamma; x' : \mathrm{Int}; xs : \mathrm{List}^p(\mathrm{Int})\ \vdash\ \textbf{let}\ xs' = \mathit{add}1\ xs\ \textbf{in}\ x' :: xs' : \mathrm{List}^q(\mathrm{Int})}$$

$$\text{where } (q_0, q_1) = (0, 0) \tag{5.2}$$

Step V: As a last step, we need to type the recursive function call. The rule (T:Application) allows us to do that. Throughout the previous steps, we derived that the input has type $\text{List}^p(\text{Int})$ and the return type is $\text{List}^q(\text{Int})$.

$$\text{(T:Application)}\frac{\Gamma \ \vdash \ \mathit{add}1 : \text{List}^p(\text{Int}) \to \text{List}^q(\text{Int})}{\Gamma; xs : \text{List}^p(\text{Int}) \ \vdash \ \mathit{add}1 \ xs : \text{List}^q(\text{Int})}$$

Step VI: As a last step, we collect numeric constraints, introduced in Eq. (5.1) and Eq. (5.2). With these constraints, we can elucidate a resource demand for the expression e_1. The resource demand then identifies the potentials p and q of input and output list, respectively. Combining Eq. (5.1) and Eq. (5.2), we get $(p_0, p_1) = (1, 0)$. Thus, the resulting resource-aware type for $\mathit{add}1$ is:

$$\mathit{add}1 :: \text{List}^1(\text{Int}) \to \text{List}(\text{Int})$$

5.3 Soundness

The introduction of potentials has a profound effect on the proof of soundness. Previously, the proof centered around inequalities on resource demands - this does not suffice. The *effective cost* that is modelled by type inference is composed of resource demands and the potential of the context.

This forces us to embellish the inequalities required to hold for soundness. Consider that evaluation and typing judgements have the following form: $E \vDash e \overset{p_0\ p_1}{\rightsquigarrow} v$ and $\Gamma \ \vdash \ e : A^{q_0}_{q_1}$. In order to define sound resource inequalities which account for potentials, we reconsider initial and residual resources. The amortized *initial* cost of an operation is composed of the concrete initial resources *and* the potential of the context in which the operation is performed. Thus the initial resources are given by $p_0 + \Phi(\Gamma)$.

In the case of residual resources, a similar line of thought holds. Here, we additionally account for the potential of the returned value v, which yields $q_1 + \Phi(v : A)$. With both initial and residual resources defined, we express soundness as a relaxation $(q_0 + \Phi(\Gamma), q_1 + \Phi(v : A)) \succcurlyeq (p_0, p_1)$, yielding two inequalities $q_0 + \Phi(\Gamma) \geq p_0$ and $(q_0 - q_1) + (\Phi(\Gamma) - \Phi(v : A)) \geq p_0 - p_1$.

Theorem 5.11 (Soundness of typing for list-tick language)**.** Let E be an Environment, Γ be a Context. Further, let $A \in \mathbb{T}$, $(p_0, p_1), (q_0, q_1) \in \mathcal{D}$ and an expression e that evaluates to the value v.

If $E \vDash e \overset{p_0\ p_1}{\rightsquigarrow} v$ and $\Gamma \ \vdash \ e : A^{q_0}_{q_1}$, then $v : A$ and $(q_0 + \Phi(\Gamma), q_1 + \Phi(v : A)) \succcurlyeq (p_0, p_1)$

Proof. (T:Nil): We know that the expression has the form $e = \textbf{nil}$. Also, $(p_0, p_1) = (0, 0)$ from (E:Nil), and $(q_0, q_1) = (0, 0)$ from (T:Nil). Together with $\Phi(\textbf{nil} : \text{List}^p(A)) = 0$, we can infer the following:

$$\begin{aligned}(q_0 + \Phi(\Gamma), q_1 + \Phi(\textbf{nil} : \text{List}^p(A)) &= (q_0 + \Phi(\Gamma), q_1) && (5.3)\\ &\succcurlyeq (q_0, q_1) && (5.4)\\ &= (p_0, p_1) && (5.5)\end{aligned}$$

Where Eq. (5.3) uses $\Phi(\textbf{nil} : \text{List}^p(A)) = 0$, and Eq. (5.4) uses Lemma 2.16.

(T:Cons): We know that $e = x :: xs$. We also assume that the tail xs has some potential p. Applying (T:Cons) gives us $q_0 = p$ and $q_1 = 0$. Applying (E:Cons) we get $p_0 = 0 = p_1$. The change in potential is given by $\Phi(xs : \text{List}^p(A)) - \Phi(x :: xs : \text{List}^p(A))$, as the context Γ comprises x and xs, where x has no associated potential. Therefore, the change in potential is precisely $-p$ - the list got elongated by one element. Thus:

$$\underbrace{q_0 - q_1}_{p} + \underbrace{\Phi(\Gamma) - \Phi(x :: xs : \text{List}^p(A))}_{-p} = 0 = p_0 - p_1$$

And:

$$q_0 + \Phi(\Gamma) = \Phi(\Gamma) \geq 0 = p_0$$

Which results in $(q_0 + \Phi(\Gamma), q_1 + \Phi(x :: xs : \text{List}^p(A)) \succcurlyeq (p_0, p_1)$.

(T:MatchList): We know that $e = \textbf{match } l \textbf{ with } \mid \textbf{nil} \to e_1 \mid x :: xs \to e_2$ and there are two possible cases. Either the list is **nil** or the list has the form x :: xs, which results in (E:MatchNil) or (E:MatchCons) being invoked for the evaluation derivation, respectively.

Let us start with the first case, $l = \textbf{nil}$. The type rule yields $p_0 = 0 = p_1$ and the evaluation rule yields $q_0 = 0 = q_1$. Furthermore, the context is empty, and the empty list has no potential:

$$\Phi(\Gamma) = 0 = \Phi(\textbf{nil} : \text{List}^p(A))$$

We apply Lemma 2.16 in Eq. (5.6) and the fact that $p_0 = q_0 = 0 = q_1 = p_1$ in Eq. (5.7) to get:

$$\begin{aligned}(q_0 + \Phi(\Gamma), q_1 + \Phi(\mathbf{nil} : \mathrm{List}^p(A))) &\succcurlyeq (q_0, q_1) && (5.6)\\ &= (p_0, p_1) && (5.7)\end{aligned}$$

The last case is $l = x :: xs$. This dictates the application of (E:MatchCons) for the evaluation. Given that the list l has potential p, we get the following assumptions:

$$\begin{aligned} p_0 = 0 \qquad p_1 &= 0 \\ q_0 = p \qquad q_1 &= 0 \\ \Phi(\Gamma) &= \Phi(x : A) + \Phi(xs : \mathrm{List}^p(A)) && (5.8)\\ \Phi(x :: xs : \mathrm{List}^p(A)) &= p + \Phi(x : A) + \Phi(xs : \mathrm{List}^p(A)) && (5.9)\end{aligned}$$

With these assumptions, we can prove the desired relaxation. First, using Eq. (5.8) and Eq. (5.9) we get:

$$p + \underbrace{\Phi(\Gamma) - \Phi(x :: xs : \mathrm{List}^p(A))}_{-p} = 0 \tag{5.10}$$

We use Lemma 2.16 to get the desired relaxation:

$$(q_0 + \Phi(\Gamma), q_1 + \Phi(x :: xs : \mathrm{List}^p(A))) \succcurlyeq (p_0, p_1)$$

(T:Application): We know that $e = f\ x$ for some function f and some variable x. Our hypothesis yields the following assumptions, where the first two rows result from applying (E:Application), and the remaining constraints result from applying (T:Application):

$$\begin{aligned} &E(f) = e_f && E[y_f \mapsto v_x] \vDash e_f \overset{p_0\ p_1}{\rightsquigarrow} v \\ &E(x) = v_x && E \vDash f\ x \overset{p_0\ p_1}{\rightsquigarrow} v \\ &\Gamma \ \vdash\ f : A \to B^{q_0}_{q_1} && \Gamma; x : A \ \vdash\ f\ x : B^{q_0}_{q_1} \\ &(q_0, q_1 + \Phi(v : B)) \succcurlyeq (p_0, p_1) && && (5.11)\end{aligned}$$

Using Lemma 2.16 we get $(q_0 + \Phi(\Gamma), q_1 + \Phi(v : B)) \succcurlyeq (p_0, p_1)$.

(T:LetFun): We know that the expression e had the form **letfun** $f = x \to e_f$ **fin** e_2. We assume that e_2 is typed soundly, which gives us three additional assumptions:

$$\Gamma \vdash e_2 : A^{r_0}_{r_1} \qquad E \vDash e_2 \overset{s_0\ s_1}{\rightsquigarrow} v \qquad (r_0, r_1 + \Phi(v : A)) \succcurlyeq (s_0, s_1)$$

Application of (E:LetFun) then gives us that $(p_0, p_1) = (s_0, s_1)$. Additionally, applying (T:LetFun) yields $(q_0, q_1) = (r_0, r_1)$. Using that $(r_0, r_1 + \Phi(v : A)) \succcurlyeq (s_0, s_1)$ from our assumptions, we get that $(q_0, q_1 + \Phi(v : A)) \succcurlyeq (p_0, p_1)$.

Using Lemma 2.16 we get $(q_0 + \Phi(\Gamma), q_1 + \Phi(v : A)) \succcurlyeq (p_0, p_1)$ as desired.

$\square$

A Learnings

- Introduced Share to avoid Linear Type system
- Push the costs of function calls to the return type.
  - The specific cost does not matter, keeps it clean.
- Use of environment E instead of heap and stack. Implementation is irrelevant.
- Introduce relaxation relation to streamline soundness proof
  - Along with naming initial and residual resources. Makes language disambiguous.
- In order to declaretively have resource demands that grow in length of list, introduce resource demands as recurrence relations

B Unit Test Functions

```
def add1 l = match l with
   | nil -> nil
   | cons(x, xs) -> let _ = tick 1 in
                    let x' = x + 1 in
                    let xs' = add1 xs in
                    cons(x', xs')
```

Figure B.1: add1 :: $\text{List}(\text{Int}) \to \text{List}(\text{Int})$

```
def sum l n = match l with
   | nil -> n
   | cons(x, xs) -> let _ = tick 2 in
                    let x' = n + x in
                    sum xs x'
```

Figure B.2: sum :: $\text{List}(\text{Int}) \to \text{Int} \to \text{Int}$

```
def map f l = match l with
   | nil -> nil
   | cons(x, xs) -> let x' = f x in
                    let xs' = map f xs in
                    conx(x', xs')
```

Figure B.3: map :: $(A \to B) \to \text{List}(A) \to \text{List}(B)$

```
def fold op l acc = match l with
   | nil -> acc
   | cons(x, xs) -> let acc' = op acc x in
                    fold op xs acc'
```

Figure B.4: fold :: $(A \to B \to B) \to \mathrm{List}(A) \to B \to B$

List of Figures

List of Tables