A Type System for Static Latency Tracking

Tobias Reinhard

Master Thesis
Technische Universität Darmstadt
Department of Computer Science
Reactive Software Systems Group

Examiner: Prof. Dr. Guido Salvaneschi
Supervisor: Pascal Weisenburger

Submission Date: 20.12.2018

Abstract

Developing efficient geo-distributed applications is challenging as computations can easily introduce latency without the programmer noticing. We propose a language design which makes latency and locations explicit and extracts static type-level bounds for a computation's runtime latency. We present a full formalization and prove that all extracted latency bounds are correct. Furthermore, we present a prototype implementation that can be used to assess the usability of the design and the extracted bounds.

Contents

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Tobias Reinhard, die vorliegende Master-Thesis gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein. Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to §22 paragraph 7 and §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Tobias Reinhard, have written the submitted thesis independently pursuant to §22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to §23 paragraph 7 of APB identical in content. For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date

Unterschrift / Signature
(Tobias Reinhard)

Chapter 1

Introduction

Developing efficient geo-distributed applications remains a challenging task. Efficiency is largely determined by latency caused by remote communication. Avoiding high-latency remote communication and exploiting locality is therefore imperative [23]. Distributed components are, however, often interconnected and local computations can trigger a chain of events causing high-latency remote computations [20, 14, 15]. Determining which local computations eventually lead to remote communication and thereby introduce latency often requires a global view. This hinders modular development of geo-distributed software. Also, the exact location where a remote computation is placed matters. Communication among servers in a single data center, for instance, is much faster (under 2ms) than communication between geo-distributed data centers possibly located on different continents (over 100ms) [4].

We present a programming language design that makes locations and latency explicit. In this language, a computation's location and its entailed latency become part of its type. The type system can infer an upper bound on a computation's actual latency and can reject code containing wrong assumptions, e.g., on the latency caused by a method invocation. A method signature already describes the latency its invocation entails. Hence, no global view is required anymore and code becomes more modular.

For this work, we consider a distributed setting similar to that of applications for geo-distributed data centers. That is, we assume a fixed set of peers and fixed connections between them.

In Chapter 2 we present our formal language design and a type system that extracts static bounds on a program's runtime latency. The presented language λ^{lat} is a total extension of the typed λ-calculus [3] featuring a restricted form of recursion. We start by explaining the static setting we consider and the language's syntax in the Sections 2.1 and 2.2, respectively. In Section 2.4 we define our language's dynamic semantics in form of two small-step reduction relations. Our reduction semantics makes a program's runtime latency explicit which allows us to reason about extracted static bounds. In particular, we explain why general recursion prohibits the extraction of static latency bounds and provide motivation for a restricted form of recursion. In Section 2.5 we formalize a type system that extracts static upper bounds on a program's runtime latency. Our type system augments types with size and latency annotations. We use a special form of sized types [1] to encode termination in the type of recursive definitions. Afterwards we prove in Section 2.6 that our language is total and that our extracted static latency bounds are correct.

In Chapter 3 we present a prototype implementation Scala$^{\text{Lat}}$ of the formal language λ^{lat} introduced in Chapter 2. We implement Scala$^{\text{Lat}}$ as an embedded Scala-DSL. In the Sections 3.1 and 3.2 we start by explaining how our language represents locations and type annotations. In particular we describe how our DSL's type system makes latency and locations transparent to the programmer. Similar to λ^{lat}, our implementation uses sized types to statically reason about termination. In Section 3.3 we define arithmetic type-level computations that allow us to encode latency and size changes in a computation's type. Furthermore, we develop a proof system to reason about arithmetic type-level computations. In Section 3.4 we explain the primitives Scala$^{\text{Lat}}$ offers the programmer to express remote communication. We also show how their type signatures make the introduced latency and the involved locations transparent to the programmer. Afterwards we describe an implementation of recursion that provably terminates by a bounded number of recursive steps and hence allows us to extract latency bounds. We implement a fixpoint operator and show how we statically unroll recursive function applications. The unrolled form allows us to bound the recursion depth and hence the latency the application causes.

In Chapter 4 we present related approaches. Afterwards we draw a conclusion of our work in Chapter 5 and present ideas for future work in Chapter 6.

Chapter 2

λ^{lat}: Latency Tracking with Placement Types

In this chapter we present the formalization λ^{lat} of a programming language design that makes latency and locations transparent to the programmer. λ^{lat} is a total extension of the typed λ-calculus [3] including a restricted form of recursion. The type system we propose allows us to extract provably correct static upper bounds on a computation's runtime latency.

For our formalization we consider a distributed setting similar to geo-distributed data centers. In particular, we consider a fixed set of peers with fixed connections between them.

We start by describing the static context that λ^{lat}-programs assume in Section 2.1. This static program context defines which locations exist and which can communicate which each other. In practice, the time a remote message needs to reach its recipient depends on many factors such as geographical proximity and quality of the connection. In λ^{lat} we approximate this time by putting fixed weights $\mathcal{L}(P, P')$ on the connection between P and P'. Afterwards, in Section 2.2 we describe the syntax of λ^{lat}. In Section 2.3 we explain the difference between runtime and type-level locations.

In Section 2.4 we define the dynamic semantics of λ^{lat} in terms of a small-step reduction semantics. λ^{lat} contains terms to express computations and such to place computations on specific locations. For each we define a separate reduction relation. We need an extended intermediate language $\lambda^{\text{lat}'}$to represent intermediate reduction results. Hence, we define those relations for $\lambda^{\text{lat}'}$. Since $\lambda^{\text{lat}'}$does contain λ^{lat} these relations still define the dynamic semantics of λ^{lat}. Our reduction relations make arising runtime latency explicit. This allows us to prove in Section 2.6 that our type system extracts correct upper bounds on a program's runtime latency. In Section 2.4.3.3 we explain why general recursion prohibits the extraction of static latency bounds. Furthermore, we motivate a restricted form of recursion that guarantees termination and allows us to extract static bounds.

In Section 2.5 we present the type system and explain how we extract static upper bounds on a program's runtime latency. Our type system makes locations explicit. The static program context defines which peers in a distributed program can directly communicate with each other. Our type system ensures that well-typed programs display a communication behaviour that complies with the restrictions defined by the program context. Similar to our reduction semantics, we define our type system by two separate typing relations: One for terms that describe computations and one for such that place computations on specific locations. The latter one builds on the former. Hence, we start by defining the typing relation for terms describing computations in the Sections 2.5.1 to 2.5.8 and define the other typing relation in Section 2.5.9.

In Section 2.5.1 we explain which information our typing relation needs and define a corresponding typing context. Our type system annotates types with sizes and extracted latency bounds. We use the sizes to prove termination of recursive functions. We describe sizes and latencies as arithmetic type-level computations. Furthermore, we do not differentiate between types that describe the same size or latency bound but use different representations like 0 or $s - s$. In Section 2.5.2 we describe an arithmetic theory to reason about sizes and latencies. Furthermore, we define an equality relation for types which allows our typing relation to abstract over different representations. In Sections 2.5.3 to 2.5.5 we explain basic types and type constructors as well as variable access. In particular, we see in Section 2.5.5 how our typing relation differentiates between values placed on different peers. This is essential to both guaranteeing valid remote communication behaviour and extracting latency bounds.

In Section 2.5.7 we explain how our type system represents function types. In λ^{lat} function types reason about about input and output sizes as well as about the latency a function application causes. In Section 2.5.7.1 we define size-dependent function types. Those are a special form of dependent function types (Π types). They abstract over a set of input sizes and allow their result type to depend on the input size. In particular, they allow us to express latency bounds which depend on the input size. In Section 2.5.7.2 we use those to define size-decreasing recursion. That is a restricted recursion pattern where each recursive step is taken on a smaller argument. Since sizes in λ^{lat} are finite we thereby guarantee that all recursive functions terminate.

We need this property to extract static latency bounds for recursive function applications.

Afterwards, we describe in Section 2.5.8 how we type the extensions added by our intermediate language $\lambda^{\text{lat}'}$. In Section 2.5.9 we define a typing relation for those terms we use to place computations on specific locations.

In Section 2.6 we investigate correctness properties for λ^{lat}. That is, we prove progress and preservation in the Sections 2.6.1 and 2.6.2, respectively. From this, we obtain the correctness of all extracted latency bounds as corollary.

2.1 Program Context

λ^{lat} programs describe computations which are potentially distributed over multiple locations. In order to refute non-sensical programs and also to evaluate sensical ones the program context is crucial. This section illustrates the necessity of contextual information and presents definitions for key concepts related to the program context. Consider the expression

```
place  $x$ : (Unit, $[0]$, $[0]$) := unit on $P_x$ in
           place  $y$ : (Option (Unit, $[0]$), $[0]$, $[2]$) := get $p.x$ on $P_y$ in
           end
```

A variable x is placed on peer P_x and bound to a local value unit. The third component of the variable's type (Unit, $[0], [0]$) represents an upper bound on the latency its evaluation involves. Since the computation of unit does not involve any remote communication, the latency is 0. In the second line a variable y is placed on a different peer P_y and references x. Evaluating y on peer P_y means requesting evaluation of x from an instance p of remote peer P_x. The type of y reflects the need for remote communication by containing a strictly positive latency 2.

In order to decide, however, whether this expression makes sense or not we need to decide whether the subexpression get $p.x$ can be evaluated. It describes an evaluation request of variable x from a peer instance p. Since x was placed on peer P_x we need to know whether p is an instance of P_x. We also need to know whether the peer P_y sending the request has a connection to peer P_x.

Furthermore, y's type (Option (Unit, $[0]$), $[0], [2]$) gives an upper bound on the latency y's evaluation involves. Determining whether 2 indeed is an upper bound necessitates us to know how fast or slow the connection between the two peers actually is.

The program context provides all these information. Before we, however, can define what exactly this context is we have to define the parts it consists of.

Definition 1 (Peer Typing). *Let $\mathbb{I}$, $\mathbb{P}$ be sets of peer instances and peer types, respectively. A peer typing $\mathcal{P}$ for $\mathbb{I}$ and $\mathbb{P}$ is a function $\mathcal{P} : \mathbb{I} \to \mathbb{P}$.*

Definition 2 (Tie Context). *Let $\mathbb{P}$ be a set of peer types. A tie context $\mathcal{T}$ for the peers in $\mathbb{P}$ is represented by a function $\mathcal{T} : \mathbb{P} \times \mathbb{P} \to \{\mathsf{none}, \mathsf{connected}\}$.*

Notation 1 (Mutual Ties). *Let $\mathbb{P}$ be a set of peer types and $\mathcal{T}$ a tie context. For peer types $P, P' \in \mathbb{P}$ we write $P \leftrightarrow P'$ if $\mathcal{T}(P, P') = \mathcal{T}(P', P) = \mathsf{connected}$ holds.*

With a peer typing $\mathcal{P}$ and a tie context $\mathcal{T}$ we can determine whether the remote request $\mathsf{get}\, p.x$ from the previous example can be accepted. Variable x is placed on peer P_x and requested via instance p. Hence, we have to check whether p is an instance of P_x, that is $\mathcal{P}(p) = P_x$. The remote request is placed on peer P_y. Therefore, we also have to check whether there is a connection between P_y and P_x. The evaluation of $\mathsf{get}\, p.x$ entails bidirectional communication. A request is sent from peer P_y to P_x and the result is sent back from P_x to P_y. By definition 2 ties between peers are not necessarily symmetric. Hence, we have to check that both $\mathcal{T}(P_y, P_x) = \mathsf{connected}$ and $\mathcal{T}(P_x, P_y) = \mathsf{connected}$ hold.

In the previous example we ascribed variable y with the type $(\mathsf{Option}\,(\mathsf{Unit}, [0]), [0], [2])$. The last component $[2]$ gives an upper bound for the latency that the evaluation of y entails. Since y was bound to the remote request $\mathsf{get}\, p.x$, we have to check whether 2 indeed bounds the latency caused by the remote request. Counting the number of remote messages sent during evaluation is not sufficient. We also have to take into account how fast or slow the involved connections are.

Definition 3 (Latency Context). *Let $\mathbb{P}$ be a set of peer types. A latency context $\mathcal{L}$ for the peers in $\mathbb{P}$ is a function $\mathcal{L} : \mathbb{P} \times \mathbb{P} \to \mathbb{N} \setminus \{0\}$.*

By using a latency context $\mathcal{L}$ we can check whether 2 is an upper bound for the latency caused by $\mathsf{get}\, p.x$. In the context of this expression's evaluation, latency is caused by three actions: Sending the remote request from P_y to P_x, evaluating variable x and sending the result back from P_x to P_y. Variable x's type $(\mathsf{Unit}, [0], [0])$ indicates that x's evaluation does not cause any latency. Therefore, adding up the latencies caused by the remote messages exchanged between P_y and P_x suffices to calculate the overall latency caused by $\mathsf{get}\, p.x$. Hence, we only have to check whether $\mathcal{L}(P_y, P_x) + \mathcal{L}(P_x, P_y) \leq 2$ holds.

Definition 4 (Program Context)**.** *Let* $\mathbb{X}$*,* $\mathbb{I}$*,* $\mathbb{P}$ *be countably infinite sets. A program context is a tuple* $(\mathbb{X}, \mathbb{I}, \mathbb{P}, \mathcal{P}, \mathcal{T}, \mathcal{L})$ *where*

- $\mathbb{X}$ *is a set of variable names*
- $\mathbb{I}$ *is a set of peer instance names*
- $\mathbb{P}$ *is a set of peer type names*
- $\mathcal{P}$ *is a peer typing for* $\mathbb{I}$ *and* $\mathbb{P}$
- $\mathcal{T}$ *is a tie context for* $\mathbb{P}$
- $\mathcal{L}$ *is a latency context for* $\mathbb{P}$*.*

2.2 Syntax

In this section we give an overview over the syntax of λ^{lat} presented in Figure 2.1. A syntactically correct program is a tuple (C, q). The first component is a program context $C = (\mathbb{X}, \mathbb{I}, \mathbb{P}, \mathcal{P}, \mathcal{T}, \mathcal{L})$ according to definition 4. It defines the static environment in which the program is executed. The second component q is a placement term following the syntactic definition presented in Figure 2.1. Note that this definition directly references the variable set $\mathbb{X}$, the peer instance set $\mathbb{I}$ and the peer type set $\mathbb{P}$ from the program context C.

Terms There are three types of terms in λ^{lat}: (i) *Placement terms* q, (ii) *placed terms* t and (iii) *arithmetic terms* h. Placed terms specify computations and are executed on specific locations and placement terms place those computations. Both are executed during the program's runtime as we see in Section 2.4. In contrast to this, arithmetic terms h represent type-level computations. They do not affect the program's runtime behaviour but are essential to type check a program. The type system we propose in Section 2.5 expects us to annotate types by their size and the latency which the evaluation an element might cause. We use arithmetic terms to formulate those annotations.

Placement Terms A program contains a sequence of placement terms q. Such terms can have the form $\mathsf{place}\, x : T \coloneqq t\, \mathsf{on}\, P\, \mathsf{in}\, q'$ or the form end in which case they mark the end of the sequence. In the former case a term t is placed on peer P and bound to variable x of type T. This variable is

$q ::= \mathsf{end} \mid \mathsf{place}\, x : T := t\, \mathsf{on}\, P\, \mathsf{in}\, q$	placement terms
$t ::= \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{none}\,(B,[h]) \mid \mathsf{some}\, t \mid \mathsf{nil}\,(B,[h]) \mid \mathsf{cons}\, t\, t \mid$	standard terms
$\quad x \mid \lambda x : (B,[h]).t \mid t\, t \mid \mathsf{let}\, x : T := t\, \mathsf{in}\, t \mid \mathsf{if}\, t\, \{t\}\, \{t\} \mid$	
$\quad t\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t\}\{\mathsf{none}\,(B,[h]) \Rightarrow t\} \mid$	
$\quad t\, \mathsf{match}\{\mathsf{cons}\, x\, x \Rightarrow t\}\{\mathsf{nil}\,(B,[h]) \Rightarrow t\} \mid$	
$\quad p \mid \mathsf{get}\, t.x \mid \mathsf{remoteCall}\, t.x\, t \mid \mathsf{eval}\, t\, \mathsf{on}\, t \mid$	remote communication
$\quad \mathsf{fix}\, t \mid Q\, .\, t$	recursion
$S ::= (B,[h])\, \mathsf{on}\, P$	placed types
$T ::= (B,[h],[h])$	annotated types
$B ::= \mathsf{Unit} \mid \mathsf{Boolean} \mid \mathsf{Option}\,(B,[h]) \mid \mathsf{List}(B,[h]) \mid$	basic types
$\quad (B,[h]) \to (B,[h],[h]) \mid Q\, .\, B \mid$	
$\quad P$	
$Q ::= \forall(x : \mathbb{N}) \mid \forall(x : \mathbb{N}) < h$	quantification
$h ::= x \mid 0 \mid S \mid h + h \mid h \dot{-} h \mid h \cdot h$	arithmetical terms
$x \in \mathbb{X}$	variables
$p \in \mathbb{I}$	peer instances
$P \in \mathbb{P}$	peer types

Figure 2.1: Syntax of λ^{lat}

available to the rest of the sequence q' and all peers connected to P regarding tie context $\mathcal{T}$. The structure of programs q as a sequence of placements models a sequential computation model. λ^{lat} does not contain facilities to specify parallel computations.

Placed Terms t represent computations and contain the standard terms known from the typed λ-calculus with a single base type Unit [3]. These include a constructor unit for the base type Unit, variables, lambda abstraction $\lambda x : (B, [h]).t$ and function application.

In contrast to the standard λ-calculus, however, λ^{lat} additionally features a $\mathsf{Boolean}$ type and type constructors Option and List. Hence, placed terms also contain respective constructors for instances of those types as well as matching deconstructors in form of if- and match-expressions.

Furthermore placed terms contain three types of expressions to express different types of remote communication. Terms of the form $\mathsf{get}\, p.x$ and $\mathsf{remoteCall}\, p.x_f\, t_2$ request the value of remote variable x and the result of a remote function application $x_f\, t_2$, respectively, from peer instance p. Similiarly, terms of the form $\mathsf{eval}\, t\, \mathsf{on}\, p$ request the result of term t evaluated on peer instance p.

Placed terms also contain fixpoint operator application $\mathsf{fix}\, t$ to express recursion and abstraction over sizes $Q\,.\,t$, where Q is an unrestricted or restricted quantifier of the form $\forall (x : \mathbb{N})$ or $\forall (x : \mathbb{N}) < h$, respectively, and h is an arithmetic term.

Types As in the standard typed λ-calculus every term t in λ^{lat} is assigned a unique type B which captures the kind of computation t describes. In λ^{lat}, however, every such basic type B is additionally annotated by t's size h_s and an upper bound h_l of the latency the computation of t involves. We use arithmetic terms to express sizes and latency.

Term representations, however, are not unique. For instance, the terms $S\,S\,0$ and $S\,0 + S\,0$ though syntactically different yield the same result when evaluated. When used as type ascriptions syntactical differences should be irrelevant. Hence, in the context of size and latency ascriptions we treat arithmetic terms as representatives of the result their evaluation yields. To syntactically highlight this special treatment we write terms in brackets $[\cdot]$ when used as representatives.

Thus fully annotated types T in λ^{lat} have the form $(B, [h_s], [h_l])$ where B is a basic type and h_s, h_l are arithmetic terms representing size and latency, respectively. In case of partial annotations $(B, [h])$ the annotation

$[h]$ always denotes a size.

Whether basic types B have to be fully or partially annotated depends on the syntactical context in which B occurs. In placement terms as well as in let- and place-terms for instance the defined variable's type must be given fully annotated. We can see this in the term $\mathsf{let}\, x : T \coloneqq t_1 \,\mathsf{in}\, t_2$. In λ-expressions on the other hand only the variable's basic type and size are specified, as in $\lambda x : (B, [h_s]).t$.

Basic types B include Unit and $\mathsf{Boolean}$. More complex types can be constructed via the type constructors Option and List as well as $\rightarrow$ for function types and $\forall(x : \mathbb{N})$ or $\forall(x : \mathbb{N}) < h_s$ for size abstraction, respectively.

In the following let B, B' be basic types, h_s, h'_s be sizes and h'_l be a latency. The type $(B, [h_s]) \rightarrow (B', [h'_s], [h'_l])$ denotes the type of a function transforming an argument of basic type B and size h_s into a result of basic type B' with size h'_s and latency at most h'_l.

Size Abstraction Universal quantification over natural numbers allows us to parameterize types and terms. In particular we can describe the type of functions accepting arguments of arbitrary size x_s and producing results whose size and latency depend on x_s as in $\forall(x_s : \mathbb{N}) \,.\, (B, [x_s]) \rightarrow (B', [x_s + S\,S\,0], [x_s + x_s])$. We can further restrict the input size x_s by restricting the qantification to natural numbers below an upper bound h_b as in $\forall(x_s : \mathbb{N}) < h_b \,.\, (B, [x_s]) \rightarrow (B', [x_s + S\,S\,0], [x_s + x_s])$. Upper bounds are given by arithmetic terms. Analogously, we can specify arbitrary and bounded sizes in the context of placed terms as in $\forall(x_s : \mathbb{N}) \,.\, \lambda x_B : (B, [x_s]).x_B$ and $\forall(x_s : \mathbb{N}) < h_b \,.\, \lambda x_B : (B, [x_s]).x_B$, respectively.

2.3 Locations

We use types to define specifications and approximate runtime behavior. Consider the functions

$$\begin{aligned} \mathit{isEmpty} = \;& \lambda x : (\mathsf{Option}\,(\mathsf{Unit}, [0]),\ [0]). \\ & \quad x\, \mathsf{match}\{\mathsf{some}\, x' \Rightarrow \mathsf{false}\}\{\mathsf{none}\,(\mathsf{Unit}, [0]) \Rightarrow \mathsf{true}\} \\ \mathit{isNotEmpty} = \;& \lambda x : (\mathsf{Option}\,(\mathsf{Unit}, [0]),\ [0]). \\ & \quad x\, \mathsf{match}\{\mathsf{some}\, x' \Rightarrow \mathsf{true}\}\{\mathsf{none}\,(\mathsf{Unit}, [0]) \Rightarrow \mathsf{false}\} \end{aligned}$$

While *isEmpty* maps empty Option-values to true and non-empty ones to false, the second function *isNotEmpty* does the exact opposite. It maps non-empty Option-values to true and empty ones to false. Both share the same type: $(\mathsf{Option}\,(\mathsf{Unit}, [0]), [0]) \rightarrow (\mathsf{Boolean}, [0], [0])$. It represents a function that accepts arguments of the form $\mathsf{some}\,\mathsf{unit}$ and $\mathsf{none}\,(\mathsf{Unit}, [0])$

$q ::= \dots$	placement terms
$t ::= \dots$	standard terms
$\mid \mathsf{get}\, t.t \mid \mathsf{remoteCall}\, t.t\, t \mid \dots$	remote communication
$\dots$	recursion
$\mid (\langle t \rangle_{\mathcal{I}}, [h])$	peer context
$S ::= \dots$	placed types
$T ::= \dots$	annotated types
$B ::= \dots$	basic types
$Q ::= \dots$	quantification
$h ::= \dots$	arithmetical terms
$x \in \mathbb{X}$	variables
$p \in \mathbb{I}$	peer instances
$P \in \mathbb{P}$	peer types
$\mathcal{I} \subseteq \mathbb{I}$	subsets of peer instances

Figure 2.2: Syntax of $\lambda^{\mathrm{lat}'}$

and transforms them into $\textsf{Boolean}$-values, i.e., $\textsf{true}$ or $\textsf{false}$.

We see that a term's type does not specify the term's exact runtime behaviour but an approximation of it. Its advantage is that it allows us to determine whether an expression makes sense without needing to know the exact values involved. Consider the term $\textsf{if}\,(\mathit{isEmpty}\,(\textsf{some}\,\textsf{unit}))\,\{...\}\,\{...\}$. We do not need to know whether the result of $\mathit{isEmpty}\,(\textsf{some}\,\textsf{unit})$ is $\textsf{true}$ or $\textsf{false}$ to determine whether or not it makes sense as a condition in an $\textsf{if}$-term. It suffices to inspect its type and see that it evaluates to a $\textsf{Boolean}$-value.

Runtime Locations λ^{lat}-programs are distributed applications and in the next section we define their dynamic semantics. To define realistic runtime-behaviour in a distributed setting, it is crucial to know the concrete locations data and computations are placed on.

Suppose we have two remote entities p and p' that make a variable x accessible to the outside. The value of x on p is $\textsf{unit}$ and and that on p' is $\textsf{false}$. Now consider the term $\textsf{get}\,p.x$. It requests the value of x from p. This term should evaluate to $\textsf{unit}$ and not to $\textsf{false}$. Hence, it is important to differentiate between the data bound to x on different locations.

In the dynamic semantics we present in Section 2.4, we use peer instances $p \in \mathbb{I}$ to denote runtime locations.

Location Abstraction Consider a distributed application involving one server s and two clients c_1, c_2. Assume that both clients are connected to the server but have no means to communicate directly with each other. The term $\textsf{get}\,s.x$ requests a variable x from the server s. In order to determine whether this computation makes sense or not we have to know whether it is evaluated on a location connected to s. It suffices to know that the computation is placed on one of the clients c_1 or c_2 to accept the term. However, which concrete client sends the request is irrelevant.

Similiar to the examples above, we can introduce *peer types* to group the *peer instances* according to their shared properties. Let S denote the type of s and let C be the type of both c_1 and c_2. Since both clients are tied to the server we can lift this mutual tie to the peer types. Hence, we define our tie context $\mathcal{T}$ as following: $\mathcal{T}(S, C) = \mathcal{T}(C, S) = \textsf{connected}$.

Then it is enough to know that the computation $\textsf{get}\,s.x$ is placed on some peer instance belonging to C. Therefore, we use peer types to abstract over runtime locations in the type system which we present in Section 2.5. Also, to simplify the semantics of our language, a placement term $\textsf{place}\,x : T := t\,\textsf{on}\,P'\,\textsf{in}\,q'$ places the computation t all instances of peer type P. The peer P then becomes part of the type we assign to x. Consider a remote request $\mathsf{get}\, p.x$ coming from a peer P'. Variable x has the type $T\, \mathsf{on}\, P$. Knowing this, our type system can check whether p is an instance of P and whether P and P' are mutually tied. Thereby, it can determine whether the remote request $\mathsf{get}\, p.x$ is legal or should be rejected. Furthermore, the type-level locations contain enough information to statically determine that the request entails a message sent from P' to P and vice-versa. Hence, our type system can extract the latency $\mathcal{L}(P', P) + \mathcal{L}(P, P')$.

2.4 Dynamic Semantics

In this section we present the dynamic semantics of λ^{lat}. For the complete section we assume a fixed program context $(\mathbb{X}, \mathbb{I}, \mathbb{P}, \mathcal{P}, \mathcal{T}, \mathcal{L})$ according to definition 4. A language's dynamic semantics describe how expressions are evaluated. We describe this evaluation in form of a step-wise reduction.

Since λ^{lat}-programs are distributed applications it essential to know on which exact location a term is reduced. We therefore need an extended intermediate representation that allows us to annotate terms by their reduction location. Furthermore, the goal of λ^{lat} is to make the latency a computation causes transparent in the computation's type. For this, however, we first need a notion of a computation's latency. That is, we need to make the latency caused during a term's reduction explicit in our dynamic semantics.

We therefore introduce *peer contexts* that allow us to annotate a term by its location and by the latency its reduction has caused so far. Figure 2.2 presents the extended intermediate representation $\lambda^{\text{lat}'}$. It contains the original language λ^{lat} and includes peer contexts as placed terms and a relaxed syntax for get-terms and remote applications.

Notation 2 (Peer Context)**.** *Let t be a placed term, $\mathcal{I} \subseteq \mathbb{I}$ be a set of peer instances, l an arithmetic term. Then the peer context $(\langle t \rangle_{\mathcal{I}}, [l])$ denotes that term t is to be reduced on all peer instances $p \in \mathcal{I}$. Furthermore, l denotes the latency that the reduction leading to this term has caused so far.*

In the following we define two reduction relations for $\lambda^{\text{lat}'}$, namely $\overset{\mathcal{I}}{\rightsquigarrow}$ to express reduction of placed terms and $\overset{\mathcal{I}}{\hookrightarrow}$ to express reduction of placement terms. Since $\lambda^{\text{lat}'}$is an extension of λ^{lat} these relations also define how λ^{lat} programs are being reduced.

For the rest of this section, whenever we address terms we implicitly refer to placed terms in $\lambda^{\text{lat}'}$. This allows us avoid redundant, separate definitions for λ^{lat} and $\lambda^{\text{lat}'}$.

2.4.1 Values

A language's expressions can be devided into the ones needing further evaluation and the ones that do not, the latter of which we call values. Intuitively, a function application $f\,a$ for a function f and an argument a should be evaluated further. On the other hand, an empty list $\mathsf{nil}\,(\mathsf{Unit}, [0])$ needs no further evaluation. Therefore $\mathsf{nil}\,(\mathsf{Unit}, [0])$ should be considered a value. The same holds for a function definition $\lambda x : (B, [s]).t$.

Now consider a term of the form $\forall(s : \mathbb{N})\,.\,t$. Whether or not it should be evaluated further depends on the use of s in its scope t. For instance, in the term $\forall(s : \mathbb{N})\,.\,\mathsf{nil}\,(\mathsf{Unit}, [0])$ the quantified variable s is never used within its scope and does therefore not represent any meaningful information. The term should be simplified to the value $\mathsf{nil}\,(\mathsf{Unit}, [0])$ and therefore not be regarded a value itself. In contrast to this, consider the term $\forall(s : \mathbb{N})\,.\,\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).x$. It represents the identity function for boolean lists of arbitrary size. The quantified variable s is referenced in its scope because it occurs free in the function's argument type $(\mathsf{List}(\mathsf{Boolean}, [0]), [s])$. This term should therefore be considered a value.

As this example illustrates, we need a precise definition which of the variables occuring in a term are free before we can give a definition of values. The variables we are interested in can only occur in arithmetic terms and when we quantify about natural numbers. We therefore call them *arithmetic variables.* To make this precise we define a function which maps placed terms to their free arithmetic variables. This function's domain is the set of placed terms denoted by t in Figure 2.1. The following definition allows us to refer to this and similar sets in an terse and precise way.

Definition 5. *For any non-terminal ξ used in a syntax definition, deriv(ξ) denotes the set objects derivable from ξ.*

With this notation we can use *deriv*(t) to denote the set of placed terms according to Figure 2.1.

Basic types like $\mathsf{List}(B, [s])$ and arithmetic terms like $s \cdot 2$ can contain variables as well. Terms like $\forall(s : \mathbb{N})\,.\,\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).x$ also contain basic types and arithmetic terms. Hence, we also need a definition for basic types and arithmetic terms.

In the following we define three functions $\widehat{FAV} : \mathit{deriv}(h) \to \wp(\mathbb{X})$, $\overline{FAV} : \mathit{deriv}(B) \to \wp(\mathbb{X})$ and $FAV : \mathit{deriv}(t) \to \wp(\mathbb{X})$ which give a precise notion of free arithmetic variables in arithmetic terms, basic types and terms, respectively.

Definition 6 (Free Arithmetic Variables in Arithmetic Terms)**.** *The function* $\widehat{FAV} : deriv(h) \to \wp(\mathbb{X})$*, is defined by induction over the structure of arithmetic terms.*

$$\begin{array}{rcl}
\widehat{FAV}(0) & := & \varnothing \\
\widehat{FAV}(x) & := & \{x\} \\
\widehat{FAV}(h + h') & := & \widehat{FAV}(h) \cup \widehat{FAV}(h') \\
\widehat{FAV}(h \dot{-} h') & := & \widehat{FAV}(h) \cup \widehat{FAV}(h') \\
\widehat{FAV}(h \cdot h') & := & \widehat{FAV}(h) \cup \widehat{FAV}(h')
\end{array}$$

The definition of a basic type's free arithmetic variables builds upon the definition free arithmetic variables in arithmetic terms.

Definition 7 (Free Arithmetic Variables in Basic Types)**.** *The function* $\overline{FAV} : deriv(B) \to \wp(\mathbb{X})$ *is defined by induction over the structure of basic types.*

$$\begin{array}{rcl}
\overline{FAV}(\mathsf{Unit}) & := & \overline{FAV}(\mathsf{Boolean}) \;:=\; \overline{FAV}(P) \;:=\; \varnothing \\
\overline{FAV}(\mathsf{List}(B, [h])) & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \\
\overline{FAV}(\mathsf{Option}\,(B, [h])) & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \\
\overline{FAV}((B, [h]) \to (B', [h'], [h''])) & := & \overline{FAV}(B)\overline{FAV}(B') \cup \\
 & & \cup\, \widehat{FAV}(h) \cup \widehat{FAV}(h') \cup \widehat{FAV}(h'') \\
\overline{FAV}(\forall (x : \mathbb{N}) \,.\, B) & := & \overline{FAV}(B) \smallsetminus \{x\} \\
\overline{FAV}(\forall (x : \mathbb{N}) < h \,.\, B) & := & \big(\widehat{FAV}(h) \cup \overline{FAV}(B)\big) \smallsetminus \{x\}
\end{array}$$

We built upon the definitions of $\widehat{FAV}$ and $\overline{FAV}$ to define which are a term's free variables.

Definition 8 (Free Arithmetic Variables in Terms)**.** *The function* FAV :

deriv$(t) \to \wp(\mathbb{X})$ *is defined by induction over the structure of terms.*

$$\begin{array}{rcl}
FAV(\mathsf{unit}) & := & FAV(\mathsf{true}) \;:=\; FAV(\mathsf{false}) \;:=\; \varnothing \\
FAV(x) & := & FAV(p) := \varnothing \\
FAV(\mathsf{none}\,(B,[h])) & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \\
FAV(\mathsf{some}\,t) & := & FAV(t) \\
FAV(\mathsf{nil}\,(B,[h])) & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \\
FAV(\mathsf{cons}\,t\,t') & := & FAV(t) \cup FAV(t') \\
FAV(\lambda x : (B,[h]).t) & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \cup FAV(t) \\
FAV(t\,t') & := & FAV(t) \cup FAV(t') \\
FAV(\mathsf{let}\,x\,:\,(B,[h],[h']) := t\,\mathsf{in}\,t') & := & \overline{FAV}(B) \cup \widehat{FAV}(h) \cup \widehat{FAV}(h') \\
 & & \cup\, FAV(t) \cup FAV(t') \\
FAV(\mathsf{if}\,t\,\{t'\}\,\{t''\}) & := & FAV(t) \cup FAV(t') \cup FAV(t'') \\
\multicolumn{3}{l}{FAV(t\,\mathsf{match}\{\mathsf{some}\,x \Rightarrow t'\}\{\mathsf{none}\,(B,[h]) \Rightarrow t''\})} \\
 & := & FAV(t) \cup FAV(t') \cup FAV(t'') \\
 & & \cup\, \overline{FAV}(B) \cup \widehat{FAV}(h) \\
\multicolumn{3}{l}{FAV(t\,\mathsf{match}\{\mathsf{cons}\,x\,x' \Rightarrow t'\}\{\mathsf{nil}\,(B,[h]) \Rightarrow t''\})} \\
 & := & FAV(t) \cup FAV(t') \cup FAV(t'') \\
 & & \cup\, \overline{FAV}(B) \cup \widehat{FAV}(h) \\
FAV(\mathsf{get}\,t.t') & := & FAV(t) \cup FAV(t') \\
FAV(\mathsf{remoteCall}\,t.t'\,t'') & := & FAV(t) \cup FAV(t') \cup FAV(t'') \\
FAV(\mathsf{eval}\,t\,\mathsf{on}\,t') & := & FAV(t) \cup FAV(t') \\
FAV(\mathsf{fix}\,t) & := & FAV(t) \\
FAV(\forall(x:\mathbb{N})\,.\,t) & := & FAV(t) \setminus \{x\} \\
FAV(\forall(x:\mathbb{N}) < h\,.\,t) & := & \big(\widehat{FAV}(h) \cup FAV(t)\big) \setminus \{x\} \\
FAV((\langle t\rangle_{\mathcal{I}},[h])) & := & \varnothing
\end{array}$$

We see that according to the above definitions, an arithmetic variable s is considered free as long as it does not occur in the scope t of a quantifier $\forall(s:\mathbb{N})\,.\,t$ or $\forall(s:\mathbb{N}) < h\,.\,t$. This aligns with the intuition built during the examples presented before.

Now we specify which terms are values. We therefore define a relation *value* $\subseteq$ *deriv*(t) and call a term t a value if *value*(t) holds.

Definition 9 (Value). *The relation value $\subseteq$ deriv(t) is defined inductively. Let t be a placed term, h an arithmetic term and x a variable. Furthermore, let B be a basic type and $\mathcal{I} \subseteq \mathbb{I}$ a set of peer instances.*

$$\begin{aligned}
\mathsf{unit} &\in \mathit{value} \\
\mathsf{true} &\in \mathit{value} \\
\mathsf{false} &\in \mathit{value} \\
\mathsf{none}\,(B, [h]) &\in \mathit{value} \\
\mathsf{nil}\,(B, [h]) &\in \mathit{value} \\
\lambda x : (B, [h]).t &\in \mathit{value} \\
p &\in \mathit{value}
\end{aligned}$$

Let v, v' be placed terms with value(v), value(v').
Furthermore, let v' be not a peer context. Then:

$$\begin{aligned}
\mathsf{some}\, v &\in \mathit{value} \\
\mathsf{cons}\, v\, v' &\in \mathit{value} \\
\mathsf{fix}\, v &\in \mathit{value} \\
\forall (x : \mathbb{N}) \,.\, v &\in \mathit{value} && \textit{if } x \in FAV(v) \\
\forall (x : \mathbb{N}) < h \,.\, v &\in \mathit{value} && \textit{if } x \in FAV(v) \\
\big(\langle v' \rangle_{\mathcal{I}}, [h]\big) &\in \mathit{value}
\end{aligned}$$

According to this definition the empty list $\mathsf{nil}\,(\mathsf{Unit}, [0])$ is a value. The same holds for the identity function for boolean lists, i.e., $\forall (s : \mathbb{N}) \,.\, \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).x$. This is because $FAV(\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).x) = \{s\}$ and s is bound by the quantification $\forall (s : \mathbb{N})$.

The quantified term $\forall (s : \mathbb{N}) \,.\, \mathsf{nil}\,(\mathsf{Unit}, [0])$ on the other hand is no value because $FAV(\forall (s : \mathbb{N}) \,.\, \mathsf{nil}\,(\mathsf{Unit}, [0])) = \varnothing$ and therefore the quantifier $\forall (s : \mathbb{N})$ does not bind any occurence of s in its scope.

We see that the above definition of values aligns with the examples presented earlier. Later we see that terms we do not consider values can be evaluated further. In particular we will see that unnecessary quantifiers like in the term $\forall (s : \mathbb{N}) \,.\, \mathsf{nil}\,(\mathsf{Unit}, [0])$ can be eliminated, yielding their former scope $\mathsf{nil}\,(\mathsf{Unit}, [0])$.

Note that the presented definition is purely syntactical and does not refer to any form of evaluation. However, we want to call those terms in $\lambda^{\mathrm{lat}'}$values which do not need any further evaluation.

We later present the reduction relations $\overset{\mathcal{I}}{\rightsquigarrow}$, $\overset{\mathcal{I}}{\hookrightarrow}$ to define how terms are evaluated. Afterwards we see that values remain uneffected by those and thereby that our definitions comply with our intuition.

2.4.2 Reduction of Placement Terms

In $\lambda^{\text{lat}'}$every computation must be placed on a specific location. While placed terms describe computations, we use placement terms $\mathsf{place}\, x : T \coloneqq t\, \mathsf{on}\, P\, \mathsf{in}\, q$ to place a computation.

In the following we explain how placement terms are evaluated. For this we define a small-step reduction relation $\hookrightarrow$. Hence, we use the term *reduce* instead of *evaluate* from now on. Before, we need, however, to introduce some notation.

Definition 10 (Peer Instance Set)**.** *For a peer type $P \in \mathbb{P}$ the set of P-instances $\mathcal{I}^P$ is defined as follows:*

$$\mathcal{I}^P \coloneqq \{\, p \in \mathbb{I} \;:\; \mathcal{P}(p) = P \,\}$$

A placement term $\mathsf{place}\, x : T \coloneqq t\, \mathsf{on}\, P\, \mathsf{in}\, q$ describes that the term t is evaluated on all P-instances $\mathcal{I}^P$. The resulting value is then bound to variable x with scope q. (The notation $x : T$ describes the type of value bound to x, which we explain in Section 2.5.)

The following nested placement term describes a computation distributed over three locations P_x, P_y, P_z.

```
place  $x : T_x \coloneqq t_x$ on $P_x$ in
        place  $y : T_y \coloneqq t_y$ on $P_y$ in
                place  $z : T_z \coloneqq t_z$ on $P_z$ in
                       end
```

The terms t_x, t_y, t_z describe local computations taking place on the peers P_x, P_y, P_z, respectively.

The goal of $\lambda^{\text{lat}'}$is to make latency and locations explicit its type system. Hence, we first need a notion a program's *runtime latency* and *runtime location* that the type system can refer to. Otherwise we had no ground to prove or disprove our type system's correctness. We therefore make runtime latency and locations explicit in both our reduction relations $\hookrightarrow$ and $\rightsquigarrow$. Note, however, that this is different for real-world programming language implementations. There, we can simply take the number of sent remote messages as reference point for the type system or statistical data on the program's average runtime-latency.

$$\frac{}{(\Xi;\ \mathsf{place}\, x : T \coloneqq t \,\mathsf{on}\, P \,\mathsf{in}\, q) \overset{\mathcal{I}^P}{\hookrightarrow} (\Xi;\ \mathsf{place}\, x : T \coloneqq (\langle t \rangle_{\mathcal{I}^P}, [0]) \,\mathsf{on}\, P \,\mathsf{in}\, q)} \quad \text{(E-LocalContextIntro)}$$

$$\frac{(\langle t \rangle_{\mathcal{I}^P}, [l]) \overset{\mathcal{I}^P}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}^P}, [l'])}{(\Xi;\ \mathsf{place}\, x : T \coloneqq (\langle t \rangle_{\mathcal{I}^P}, [l]) \,\mathsf{on}\, P \,\mathsf{in}\, q) \overset{\mathcal{I}^P}{\hookrightarrow} (\Xi;\ \mathsf{place}\, x : T \coloneqq (\langle t' \rangle_{\mathcal{I}^P}, [l']) \,\mathsf{on}\, P \,\mathsf{in}\, q)} \quad \text{(E-PlacementContext)}$$

$$\frac{value(v)}{(\Xi;\ \mathsf{place}\, x : T \coloneqq (\langle v \rangle_{\mathcal{I}^P}, [l]) \,\mathsf{on}\, P \,\mathsf{in}\, q) \overset{\mathcal{I}^P}{\hookrightarrow} (\Xi, x \mapsto (\langle v \rangle_{\mathcal{I}^P}, [l])\,;\ q[x \mapsto (\langle v \rangle_{\mathcal{I}^P}, [0])])} \quad \text{(E-PlacedVal)}$$

Figure 2.3: Placement Term Reduction Rules

In the next section we define the placed-term reduction relation $\rightsquigarrow$ which reduces terms t_i to a peer context $(\langle v_i \rangle_{\mathcal{I}_i}, [l_i])$. It describes the produced value v_i, the caused latency l_i and the value's location in form of a set of peer instances $\mathcal{I}_i$ on which v_i has been computed. Our reduction relation for placement terms $\hookrightarrow$ therefore builds an environment Ξ mapping placed variables to their associated peer contexts. That is, $x \mapsto (\langle v_x \rangle_{\mathcal{I}^{P_x}}, [l_x])$, $y \mapsto \big(\langle v_y \rangle_{\mathcal{I}^{P_y}}, [l_y]\big)$, $z \mapsto (\langle v_z \rangle_{\mathcal{I}^{P_z}}, [l_z])$ for this example.

Definition 11 (Placed Value Environment)**.** *Let x, t, h, P denote the corresponding syntactic definitions presented in Figure 2.2. A placed value environment Ξ is defined syntactically as follows:*

$$\Xi \;\coloneqq\; \varnothing \;\mid\; \Xi, x \mapsto (\langle t \rangle_{\mathcal{I}}, [h])$$

Figure 2.3 presents the rules defining the reduction relation $\hookrightarrow$. A reduction step has the form $(\Xi;\ q) \overset{\mathcal{I}}{\hookrightarrow} (\Xi';\ q')$. It describes that placement term q is reduced to q' and that placed value environment Ξ is extended to Ξ'.

The presented rules use the term reduction relation $\rightsquigarrow$ presented in the next section. For now, it suffices to know that a term reduction step has the form $(\langle t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}}, [l'])$. It describes term t being reduced on a set of peer instances $\mathcal{I}$ to t' while increasing the latency from l to l'.

Reduction Rules As mentioned before, a placement term $\mathsf{place}\, x : T \coloneqq t \,\mathsf{on}\, P \,\mathsf{in}\, q$ places the computation t on all instances of peer P, that is on $\mathcal{I}^P$. We need the information on which peer instances a term is placed during the term's reduction by our $\rightsquigarrow$ explained in Section 2.4.3. Reduction rule E-LocalContextIntro therefore wraps t into a peer context $(\langle\cdot\rangle_{\mathcal{I}^P}, [0])$. Thereby, it specifies that t is to be reduced on all instances belonging to peer type P and that no latency has been caused so far. Meanwhile, the placed value environment Ξ remains unchanged.

Rule E-PlacementContext states that whenever we can reduce term t on the P-instances $\mathcal{I}^P$, we can lift this reduction to the placement term. That is, provided a placed term reduction step $(\langle t\rangle_{\mathcal{I}^P}, [l]) \overset{\mathcal{I}^P}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}^P}, [l'])$, we can reduce the placement term $\mathsf{place}\, x : T \coloneqq (\langle t\rangle_{\mathcal{I}^P}, [l])\ \mathsf{on}\, P\, \mathsf{in}\, q$ to $\mathsf{place}\, x : T \coloneqq (\langle t'\rangle_{\mathcal{I}^P}, [l'])\ \mathsf{on}\, P\, \mathsf{in}\, q$. This reduction step leaves the placed value environment Ξ unchanged.

After the placed peer context $(\langle t\rangle_{\mathcal{I}^P}, [l])$ has been reduced to a value $(\langle v\rangle_{\mathcal{I}^P}, [l_v])$, we can according to rule E-PlacedVal extend our placed value environment Ξ by the binding $x \mapsto (\langle v\rangle_{\mathcal{I}^P}, [l_v])$. We also have to make the binding visible in its scope q. Accessing the value bound to x does not trigger a recomputation of v on $\mathcal{I}^P$. Hence, we substitute all occurences of x in q by the value $(\langle v\rangle_{\mathcal{I}^P}, [0])$. However, requesting the value bound to x from a remote location does involve remote communication and should increase the tracked runtime-latency accordingly. We later see in Section 2.4.3.2 that our reduction relation for placed terms $\rightsquigarrow$ indeed complies with this intuition.

In the standard λ-calculus variables only occur in terms, not in types. In λ^{lat}, however, the various forms of terms (placement, placed, arithmetic) and types (basic and fully annotated) can contain variables. As a consequence, a substitution's scope is not always obvious. Consider the term $t_\forall = \forall(\hat{x} : \mathbb{N})\,.\,\lambda x : (B, [s]).t$ and a substitution $t_\forall[\hat{x} \mapsto z]$. The scope of $\hat{x}$ is $\lambda x : (B, [s]).t$. Since the substitution targets a placed term, one could assume that it also only affects placed terms, i.e., t in this case. Meanwhile, $\hat{x}$ is bound by a quantifier ranging over natural numbers. Hence, one could also assume that is only affects arithmetic terms, i.e., s in this example. In fact, however, it affects both placed and arithmetic terms as well as basic types. That is $t_\forall[\hat{x} \mapsto z] = \lambda x : (B[\hat{x} \mapsto z], [s[\hat{x} \mapsto z]]).t[\hat{x} \mapsto z]$. The following definitions are necessary to make this precise.

Definition 12 (Subsitition in Arithmetic Terms)**.** *Let h be an arithmetic term, t^* a placed term and x a variable. We define the substitution $h[x \mapsto t^*]$ by induction over the structure of h.*

In the following, let h', h'' be arithmetic terms and let x' be a variable with $x \neq x'$.

$$
\begin{aligned}
x[x \mapsto t^*] &:= t^* \\
x'[x \mapsto t^*] &:= x' \\
(S\,h')[x \mapsto t^*] &:= S\,(h'[x \mapsto t^*]) \\
(h' + h'')[x \mapsto t^*] &:= (h'[x \mapsto t^*]) + (h''[x \mapsto t^*]) \\
(h' \dot{-} h'')[x \mapsto t^*] &:= (h'[x \mapsto t^*]) \dot{-} (h''[x \mapsto t^*]) \\
(h' \cdot h'')[x \mapsto t^*] &:= (h'[x \mapsto t^*]) \cdot (h''[x \mapsto t^*])
\end{aligned}
$$

Definition 13 (Substitution in Basic Types)**.** *Let B be a basic type, t^* a placed term and x a variable. We define the substitution $B[x \mapsto t^*]$ by induction over the structure of B.*

In the following let B', B'' be basic types and P a peer type. Furthermore, let t' be a placed term, s, s', l be arithmetic terms and let x' be a variable with $x \neq x'$.

$$
\begin{aligned}
\mathsf{Unit}[x \mapsto t^*] &:= \mathsf{Unit} \\
\mathsf{Boolean}[x \mapsto t^*] &:= \mathsf{Boolean} \\
P[x \mapsto t^*] &:= P \\
(\mathsf{Option}\,(B', [s]))[x \mapsto t^*] &:= \mathsf{Option}\,((B'[x \mapsto t^*]), [(s[x \mapsto t^*])]) \\
(\mathsf{List}(B', [s]))[x \mapsto t^*] &:= \mathsf{List}((B'[x \mapsto t^*]), [(s[x \mapsto t^*])]) \\
((B', [s]) \to (B'', [s'], [l]))[x \mapsto t^*] &:= (B'[x \mapsto t^*], [s[x \mapsto t^*]]) \to \\
&\qquad (B''[x \mapsto t^*], [s'[x \mapsto t^*]], [l[x \mapsto t^*]]) \\
(\forall (x : \mathbb{N}) \,.\, t')[x \mapsto t^*] &:= \forall (x : \mathbb{N}) \,.\, t' \\
(\forall (x' : \mathbb{N}) \,.\, t')[x \mapsto t^*] &:= \forall (x : \mathbb{N}) \,.\, (t'[x \mapsto t^*]) \\
(\forall (x : \mathbb{N}) < h \,.\, t')[x \mapsto t^*] &:= \forall (x : \mathbb{N}) < h \,.\, t' \\
(\forall (x' : \mathbb{N}) < h \,.\, t')[x \mapsto t^*] &:= \forall (x' : \mathbb{N}) < (h[x \mapsto t^*]) \,.\, (t'[x \mapsto t^*])
\end{aligned}
$$

Definition 14 (Substitution in Types)**.** *Let B be a basic type and s, l arithmetic terms and let t^* be a placed term. We define the substitution $(B, [s], [l])[x \mapsto t^*]$ as follows:*

$$
(B, [s], [l])[x \mapsto t^*] \quad := \quad (B[x \mapsto t^*],\ [s[x \mapsto t^*]],\ [l[x \mapsto t^*]])
$$

Definition 15 (Substitution in Placed Terms)**.** *Let t, t^* be placed terms and x a variable. We define the substitution $t[x \mapsto t^*]$ by induction over the structure of t.*

In the following let t', t'', t''' be placed terms and let $x', x'', \widehat{x}', \widehat{x}''$ be variables with $x \neq x', x \neq x''$. Furthermore, let B be a basic type, s and arithmetic term and T a fully annotated type.

$$\begin{aligned}
(\mathsf{none}\,(B,[s]))[x \mapsto t^*] &:= \mathsf{none}\,(B[x \mapsto t^*],[s[x \mapsto t^*]])\\
(\mathsf{some}\,t')[x \mapsto t^*] &:= \mathsf{some}\,(t'[x \mapsto t^*])\\
(\mathsf{nil}\,(B,[s]))[x \mapsto t^*] &:= \mathsf{nil}\,(B[x \mapsto t^*],[s[x \mapsto t^*]])\\
(\mathsf{cons}\,t'\,t'')[x \mapsto t^*] &:= \mathsf{cons}\,(t'[x \mapsto t^*])\,(t''[x \mapsto t^*])\\
x[x \mapsto t^*] &:= t^*\\
x'[x \mapsto t^*] &:= x'\\
\lambda x : (B,[s]).t'[x \mapsto t^*] &:= \lambda x : (B[x \mapsto t^*],[s[x \mapsto t^*]]).t'\\
\lambda x' : (B,[s]).t'[x \mapsto t^*] &:= \lambda x : (B[x \mapsto t^*],[s[x \mapsto t^*]]).(t'[x \mapsto t^*])\\
t'\,t''[x \mapsto t^*] &:= (t'[x \mapsto t^*])\,(t''[x \mapsto t^*])
\end{aligned}$$

$$\begin{aligned}
\mathsf{let}\,x : T := t'\,\mathsf{in}\,t''[x \mapsto t^*] &:= \mathsf{let}\,x : (T[x \mapsto t^*]) := (t'[x \mapsto t^*])\,\mathsf{in}\,t''\\
\mathsf{let}\,x' : T := t'\,\mathsf{in}\,t''[x \mapsto t^*] &:= \mathsf{let}\,x' : (T[x \mapsto t^*]) := (t'[x \mapsto t^*])\,\mathsf{in}\,t''[x \mapsto t^*]\\
\mathsf{if}\,t'\,\{t''\}\,\{t'''\}[x \mapsto t^*] &:= \mathsf{if}\,(t'[x \mapsto t^*])\,\{t''[x \mapsto t^*]\}\,\{t'''[x \mapsto t^*]\}\\
\mathsf{get}\,t'.t''[x \mapsto t^*] &:= \mathsf{get}\,(t'[x \mapsto t^*]).(t''[x \mapsto t^*])\\
\mathsf{remoteCall}\,t'.t''\,t'''[x \mapsto t^*] &:= \mathsf{remoteCall}\,(t'[x \mapsto t^*]).(t''[x \mapsto t^*])\,(t'''[x \mapsto t^*])\\
\mathsf{eval}\,t''\,\mathsf{on}\,t'[x \mapsto t^*] &:= \mathsf{eval}\,(t''[x \mapsto t^*])\,\mathsf{on}\,(t'[x \mapsto t^*])\\
\mathsf{fix}\,t'[x \mapsto t^*] &:= \mathsf{fix}\,(t'[x \mapsto t^*])\\
\forall (x : \mathbb{N})\,.\,t'[x \mapsto t^*] &:= \forall (x : \mathbb{N})\,.\,t'\\
\forall (x' : \mathbb{N})\,.\,t'[x \mapsto t^*] &:= \forall (x' : \mathbb{N})\,.\,(t'[x \mapsto t^*])\\
\forall (x : \mathbb{N}) < s\,.\,t'[x \mapsto t^*] &:= \forall (x : \mathbb{N}) < (s[x \mapsto t^*])\,.\,t'\\
\forall (x' : \mathbb{N}) < s\,.\,t'[x \mapsto t^*] &:= \forall (x' : \mathbb{N}) < (s[x \mapsto t^*])\,.\,(t'[x \mapsto t^*])\\
\big(\langle t'\rangle_{\mathcal{I}},[l]\big)\,[x \mapsto t^*] &:= \big(\langle t'[x \mapsto t^*]\rangle_{\mathcal{I}},[l]\big)
\end{aligned}$$

$$\begin{aligned}
&t' \,\mathsf{match}\{\mathsf{some}\, x \Rightarrow t''\}\{\mathsf{none}\,(B,[s]) \Rightarrow t'''\}[x \mapsto t^*] := \\
&\quad t'[x \mapsto t^*]\,\mathsf{match}\{\mathsf{some}\, x \Rightarrow t''\}\{\mathsf{none}\,(B[x \mapsto t^*],[s[x \mapsto t^*]]) \Rightarrow t'''[x \mapsto t^*]\} \\
&t' \,\mathsf{match}\{\mathsf{some}\, x' \Rightarrow t''\}\{\mathsf{none}\,(B,[s]) \Rightarrow t'''\}[x \mapsto t^*] := \\
&\quad t'[x \mapsto t^*]\,\mathsf{match}\{\mathsf{some}\, x' \Rightarrow t''[x \mapsto t^*]\}\{\mathsf{none}\,(B[x \mapsto t^*],[s[x \mapsto t^*]]) \Rightarrow t'''[x \mapsto t^*]\} \\
&t' \,\mathsf{match}\{\mathsf{cons}\, \widehat{x}'\, \widehat{x}'' \Rightarrow t''\}\{\mathsf{nil}\,(B,[s]) \Rightarrow t'''\}[x \mapsto t^*] \qquad \textit{for} \;\; x \in \{\widehat{x}', \widehat{x}''\} \\
&\quad := \quad t'[x \mapsto t^*]\,\mathsf{match}\{\mathsf{cons}\, x'\, x'' \Rightarrow t''\}\{\mathsf{nil}\,(B[x \mapsto t^*],[s[x \mapsto t^*]]) \Rightarrow t'''[x \mapsto t^*]\} \\
&t' \,\mathsf{match}\{\mathsf{cons}\, x'\, x'' \Rightarrow t''\}\{\mathsf{nil}\,(B,[s]) \Rightarrow t'''\}[x \mapsto t^*] := \\
&\quad t'[x \mapsto t^*]\,\mathsf{match}\{\mathsf{cons}\, x'\, x'' \Rightarrow t''[x \mapsto t^*]\}\{\mathsf{nil}\,(B[x \mapsto t^*],[s[x \mapsto t^*]]) \Rightarrow t'''[x \mapsto t^*]\}
\end{aligned}$$

Otherwise

$$t[x \mapsto t^*] := t$$

Definition 16 (Substitution in Placement Terms)**.** *Let q be a placement term, t^* a placed term and x a variables. We define the substitution $q[x \mapsto t^*]$ by induction on the structure of q.*

In the following, let q' be a placement term, t a placed term and x' a variable with $x \neq x'$. Furthermore, let P and T be a peer type and a fully annotated type, respectively.

$$\begin{aligned}
(\mathsf{place}\, x : T := t\, \mathsf{on}\, P\, \mathsf{in}\, q')[x \mapsto t^*] \quad &:= \quad (\mathsf{place}\, x : T := t\, \mathsf{on}\, P\, \mathsf{in}\, q') \\
(\mathsf{place}\, x' : T := t\, \mathsf{on}\, P\, \mathsf{in}\, q')[x \mapsto t^*] \quad &:= \quad \big(\mathsf{place}\;\; x' : T := (t[x \mapsto t^*])\, \mathsf{on}\, P\, \mathsf{in} \\
&\qquad\qquad (q'[x \mapsto t^*])\big) \\
\mathsf{end}[x \mapsto t^*] \quad &:= \quad \mathsf{end}
\end{aligned}$$

2.4.3 Reduction of Placed Terms

A Placement term $\mathsf{place}\, x : T \coloneqq t\, \mathsf{on}\, P\, \mathsf{in}\, q$ places the computation t on some peer P. By binding the result to a variable x with scope q we make it accessible to all computations in q. Accessing the variable's value from a remote peer involves network communication and thereby introduces latency.

The time a message sent from a peer P to P' needs to reach its recipient depends on the geographical proximity of P and P' as well as on their network connection. As mentioned in Section 2.1 we assume those times to be fixed. We use a latency context $\mathcal{L}$ to assign weights $\mathcal{L}(P, P')$ (i.e. latency approximations) to the messages sent from P to P'.

Consider the following computation distributed over two peers P_x, P_y.

```
place  $x : T_x \coloneqq \mathsf{true}\,\mathsf{on}\, P_x\, \mathsf{in}$
          place  $y : T_y \coloneqq \mathsf{get}\, p_x.x\, \mathsf{on}\, P_y\, \mathsf{in}$
                 end
```

Runtime Latency The program places a boolean value on P_x and binds it to variable x. Peer P_z requests the value bound to x. Therefore P_y must send a request for x to P_x and P_x must send the value of x back to P_y. This accumulates to a latency of $\mathcal{L}(P_y, P_x) + \mathcal{L}(P_x, P_y)$.

Remote Communication Failure In reality remote communication can fail due to message loss or time-out and is hence non-deterministic from a message's sender's point-of-view. All terms in $\lambda^{\mathrm{lat}'}$involving remote communication (i.e. containing $\mathsf{get}\, t.t'$, $\mathsf{remoteCall}\, t.t'\, t''$, $\mathsf{eval}\, t'\, \mathsf{on}\, t$) therefore reduce non-deterministically to an Option-value. In the above example variable x is placed on P_x and bound to the value true. The term $\mathsf{get}\, p_x.x$ placed on P_y hence reduces to either $\mathsf{some}\,\mathsf{true}$ or $\mathsf{none}\,(\mathsf{Boolean}, [0])$.

Locations Above we calculated the latency caused during the reduction of $\mathsf{get}\, p_x.x$. The result $\mathcal{L}(P_y, P_x) + \mathcal{L}(P_x, P_y)$ depends on the locations the variable x and term $\mathsf{get}\, p_x.x$ are placed on, respectively. As explained in Section 2.3 we use peer instances $p \in \mathbb{I}$ to specify runtime locations and peer types $P \in \mathbb{P}$ to specify type-level locations. The placement term $\mathsf{place}\, x : T_x \coloneqq \mathsf{true}\, \mathsf{on}\, P_x\, \mathsf{in} \ldots$ places variable x on peer P_x. This has two consequences: (i) The computation bound to variable x is executed on all instances $\mathcal{I}^{P_x}$ of peer type P_x and therefore x is accessible via all peer instances $p \in \mathcal{I}^{P_x}$. (ii) As we later see in Section 2.5 the peer P_x becomes part of x's type. Locations thereby become transparent in the type system.

Peer Context In the above example, the term $\mathsf{get}\, p_x.x$ is reduced on $\mathcal{I}^{P_x}$. The term requests variable x from peer instance p_x. The reduction of x hence takes place on p_x. In our reduction semantics we make a reduction's runtime location explicit by wrapping the reduced term in a peer context $(\langle\cdot\rangle_{\mathcal{I}}, [l])$. Here, $\mathcal{I}$ denotes the set of peer instances the term is reduced on and l denotes the latency the term's reduction has caused so far.

Reduction Relation In the rest of this section we define the reduction relation for placed terms $\rightsquigarrow$. Figures 2.4 and 2.5 present the defining rules. A reduction step has the form $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}}, [l'])$. It describes that term t is reduced on a set of peer instances $\mathcal{I} \subseteq \mathbb{I}$ to term t'. Meanwhile the tracked runtime latency increased from l to l'. Since figures 2.4 and 2.5 contain no latency-reducing rules, we know $l \leq l'$.

2.4.3.1 Local Reduction

In the following we explain the reduction rules presented in Figure 2.4. These rules define reduction steps not introducing any remote communication. We therefore call them *local reduction rules* and *local reduction steps*, respectively. Most of these rules only allow reduction steps $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}}, [l])$ which leave the set of runtime locations $\mathcal{I}$ and the tracked runtime latency l unchanged. The only exception will be presented at the end of this section and bridges the gap between local reduction rules and remote reduction rules explained in the next section.

Conditions Rules E-IfTrue and E-IfFalse define how if-terms $\mathsf{if}\, v_c\, \{t_t\}\, \{t_f\}$ with a boolean value v_c as condition are reduced on a set of peer instances $\mathcal{I}$. If the condition holds (i.e. $v_c = \mathsf{true}$) the term is reduced to its first branch t_t. The reduction step is taken on $\mathcal{I}$ and has the form $\big(\langle \mathsf{if}\, \mathsf{true}\, \{t_t\}\, \{t_f\}\rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_t\rangle_{\mathcal{I}}, [l])$. The tracked latency l remains unchanged as well as the term's set of runtime locations $\mathcal{I}$. Otherwise (i.e. $v_c = \mathsf{false}$) the term is reduced to its second branch t_f and the reduction step is analogous.

Pattern Matching match-terms allow to deconstruct Option- and List-values. Both types have two constructors. Therefore, the reduction rules for match-terms are similiar to those for if-terms.

The rules E-MatchNone and E-MatchSome define how Option-values are deconstructed. For a none-value the match-term is reduced to its none-branch. This leaves the tracked latency and the runtime location unchanged.

$$\frac{E[t] \neq t \qquad (\langle t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}}, [l'])}{(\langle E[t] \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[t'] \rangle_{\mathcal{I}}, [l'])}\ \text{(E-LocalContext)}$$

$$\frac{\mathit{value}(v)}{(\langle \mathsf{let}\, x : T := v \,\mathsf{in}\, t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t[x \mapsto v] \rangle_{\mathcal{I}}, [l])}\ \text{(E-Let)}$$

$$\frac{}{\big(\langle \mathsf{if\ true}\, \{t_t\}\, \{t_f\} \rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_t \rangle_{\mathcal{I}}, [l])}\ \text{(E-IfTrue)}$$

$$\frac{}{\big(\langle \mathsf{if\ false}\, \{t_t\}\, \{t_f\} \rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle t_f \rangle_{\mathcal{I}}, [l]\big)}\ \text{(E-IfFalse)}$$

$$\frac{}{\big(\langle \mathsf{none}\, (B, [s])\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t_s\}\{\mathsf{none}\, (\widehat{B}, [\widehat{s}]) \Rightarrow t_n\} \rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_n \rangle_{\mathcal{I}}, [l])}\ \text{(E-MatchNone)}$$

$$\frac{\mathit{value}(v)}{(\langle \mathsf{some}\, v\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t_s\}\{\mathsf{none}\, (B, [s]) \Rightarrow t_n\} \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_s[x \mapsto v] \rangle_{\mathcal{I}}, [l])}\ \text{(E-MatchSome)}$$

$$\frac{}{\big(\langle \mathsf{nil}\, (B, [s])\, \mathsf{match}\{\mathsf{cons}\, x_h\, x_t \Rightarrow t_c\}\{\mathsf{nil}\, (\widehat{B}, [\widehat{s}]) \Rightarrow t_n\} \rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_n \rangle_{\mathcal{I}}, [l])}\ \text{(E-MatchNil)}$$

$$\frac{\mathit{value}(v_h) \qquad \mathit{value}(v_t)}{(\langle \mathsf{cons}\, v_h\, v_t\, \mathsf{match}\{\mathsf{cons}\, x_h\, x_t \Rightarrow t_c\}\{\mathsf{nil}\, (B, [s]) \Rightarrow t_n\} \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_c[x_h \mapsto v_h][x_t \mapsto v_t] \rangle_{\mathcal{I}}, [l])}\ \text{(E-MatchCons)}$$

$$\frac{\mathit{value}(\overline{Q}[\lambda x : (B, [s]).t]) \qquad \mathit{value}(v)}{\big(\langle \overline{Q}[\lambda x : (B, [s]).t]\, v \rangle_{\mathcal{I}}, [l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle \overline{Q}[t[x \mapsto v]] \rangle_{\mathcal{I}}, [l]\big)}\ \text{(E-LocalApp)}$$

$$\frac{\begin{array}{c} f = \forall (u : \mathbb{N}) \,.\, \lambda x : (\forall (s : \mathbb{N}) < u \,.\, B, [h]).t \\ f' = \forall (u : \mathbb{N}) \,.\, \lambda x : (\forall (s : \mathbb{N}) \,.\, B, [h]).BE(t) \\ \mathit{value}(f) \qquad \mathit{value}(v) \end{array}}{(\langle \mathsf{fix}\, f\, v \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle f'\, (\mathsf{fix}\, f)\, v \rangle_{\mathcal{I}}, [l])}\ \text{(E-FixApp)}$$

(a) Standard Terms

$$\frac{s \notin FAV(t)}{(\langle \forall (s : \mathbb{N}) \,.\, t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t \rangle_{\mathcal{I}}, [l])}\ \text{(E-ForAllElim)}$$

$$\frac{s \notin FAV(t)}{(\langle \forall (s : \mathbb{N}) < u \,.\, t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t \rangle_{\mathcal{I}}, [l])}\ \text{(E-BoundedForAllElim)}$$

(b) Quantifier Elimination

Figure 2.4: Local Reduction Rules for Placed Terms

A corresponding reduction step has the form
$\big(\langle \mathsf{none}\,(B,[s])\,\mathsf{match}\{\mathsf{some}\,x \Rightarrow t_s\}\{\mathsf{none}\,(\widehat{B},[\widehat{s}]) \Rightarrow t_n\}\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_n\rangle_{\mathcal{I}},[l])$.

The deconstruction of a value $\mathsf{some}\,v$ is similiar. However, the some-branch's pattern $\mathsf{some}\,x$ introduces a variable x. Hence, when we reduce a match-term to its some-branch t_s, we substitute every free occurence of x in t_s by v. A reduction step therefore has the form
$\big(\langle \mathsf{some}\,v\,\mathsf{match}\{\mathsf{some}\,x \Rightarrow t_s\}\{\mathsf{none}\,(\widehat{B},[\widehat{s}]) \Rightarrow t_n\}\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_s[x \mapsto v]\rangle_{\mathcal{I}},[l])$.
Note that we can only apply E-MatchSome to deconstruct the term $\mathsf{some}\,v$ if $\mathsf{some}\,v$ is a value. The rule contains the precondition $\mathit{value}(v)$. According to definition 9 this is equivalent to $\mathsf{some}\,v$ being a value.

Terms $\mathsf{some}\,t$ for which the precondition $\mathit{value}(t)$ does not hold, must be reduced to a value before we can apply E-MatchSome. At the end of this section we present the reduction rule E-LocalContext to handle this and similiar cases.

The rules E-MatchNil and E-MatchCons define decomposition of List-values and are analogous to E-MatchNone and E-MatchSome, respectively.

Functions can be defined by λ-terms $\lambda x : (B,[s]).t$. Thereby we introduce a variable x to the function body t where it shadows every previous introduction of x. According to rule E-LocalApp, we reduce an application to a value v by substituting every occurence of x in t by v. The corresponding reduction step has the form $(\langle(\lambda x : (B,[s]).t)\,v\rangle_{\mathcal{I}},[l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t[x \mapsto v]\rangle_{\mathcal{I}},[l])$.

E-LocalApp contains the precondition $\mathit{value}(v)$. Thus, analogous to the rule E-MatchSome explained above, we must reduce the function argument to a value before we can apply E-LocalApp.

Note that when we defined substitution for placed terms (see def. 15) we ensured that it respects the substituted variable's scope. The term $\lambda x : (\mathsf{Unit},[0]).\lambda x : (\mathsf{Boolean},[0]).\mathsf{some}\,x$ introduces two variables named x. For clarity we refer to these as $x_{(1)}$ and $x_{(2)}$, respectively, but remain to assume $x_{(1)} = x_{(2)}$. According to the definition we expect the first variable $x_{(1)}$ to be substituted by an Unit-value within its scope $\lambda x_{(2)} : (\mathsf{Boolean},[0]).\mathsf{some}\,x_{(2)}$. By introducing the second variable $x_{(2)}$ within the scope of the first one, we shadow $x_{(1)}$. Thus, any binding to the first variable becomes ineffective in the scope of the second one. Following definition 15, the subsitution $(\lambda x_{(2)} : (\mathsf{Boolean},[0]).\mathsf{some}\,x_{(2)})[x_{(1)} \mapsto \mathsf{unit}]$ yields the unaltered body $\lambda x_{(2)} : (\mathsf{Boolean},[0]).\mathsf{some}\,x_{(2)}$. In particular the substitution $[x_{(1)} \mapsto \mathsf{unit}]$ is not applied to $\mathsf{some}\,x_{(2)}$, because it lies within the scope of $x_{(2)}$ and thus outside of $x_{(1)}$'s scope.

Stuck Reductions When we define a function via some λ-term $\lambda x : (B, [s]).t$ we specify which kind of argument we expect x to be substituted by. Violating this expectation can result in a stuck reduction. Consider the function $neg = \lambda x : (\mathsf{Boolean}, [0]).\mathsf{if}\, x\, \{\mathsf{false}\}\, \{\mathsf{true}\}$ which computes the negation of the passed argument. For any boolean b, the application $neg\ b$ is reduced to the correct result $\neg b$. However, supplying unit instead of a $\mathsf{Boolean}$ results in a stuck reduction. The application $neg\, \mathsf{unit}$ is reduced to $\mathsf{if}\, \mathsf{unit}\, \{\mathsf{false}\}\, \{\mathsf{true}\}$. An if-term can only be reduced applying E-IfTrue or E-IfFalse. We can, however, only apply these if the condition is true or false, respectively. Hence, we cannot reduce the term any further.

In order to ensure that we consider only terms for reduction which behave in a sensical way, in Section 2.5 we assign types and sizes to terms. By not assigning a type to non-sensical terms like $neg\, \mathsf{unit}$ we ensure that every term we assign a type to correctly reduces to a value. In particular we forbid any function application $(\lambda x : (B, [s]).t)\, a$ where the supplied argument a does not comply with the expected argument type B and size s. This includes ill-sized applications like $(\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [2]).\ ...)\, \mathsf{nil}\, (\mathsf{Boolean}, [0])$ where the function expects a boolean list with 2 elements but is applied to an empty one.

Abstraction over Sizes We can also formulate functions which compute correct results for more than one specific size. For instance, consider the identity function for boolean lists $id_{\bar{s}} = \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\bar{s}]).x.$ for any arithmetic term $\bar{s}$. For any boolean list l the application $id_{\bar{s}}\ l$ is correctly reduced to l. However, if the arguments size does not equal $\bar{s}$, the type system proposed in Section 2.5 refutes the application $id_{\bar{s}}\ l$.

We can convey the type system to accept our application by modifying the identity function. We can introduce a universally quantified size variable s and replace $\bar{s}$ by s. This yields $id_{\forall} = \forall (s : \mathbb{N}) \,.\, \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).x$. Analogous, we can abstract over a range of sizes $\{0, ..., u - 1\}$ by restricting the universal quantification in the form $\forall (s : \mathbb{N}) < u$. Intuitively, the application $id_{\forall}\, l$ should be reduced to l. Consider the general case of a function application $(Q.f)\, a$ accepted by the type system. The function f lies in the scope of a quantifier Q while a does not. The reduction result r might contain a variable s quantified in Q. In this case the the type system considers r with the free variable s to be not correctly typed. It, however, accepts $Q.r$. We have to consider this typing behavior in the definition of our reduction relation.

Considering quantification, local function application can have three forms: (i) $f\,a$, (ii) $(\forall(s:\mathbb{N})\,.\,f)\,a$ and (iii) $(\forall(s:\mathbb{N}) < u\,.\,f)\,a$. Hence, we could define three separate reduction rules of the form: (i) $f\,a \rightsquigarrow r$, (ii) $(\forall(s:\mathbb{N})\,.\,f)\,a \rightsquigarrow \forall(s:\mathbb{N})\,.\,r$ and (iii) $(\forall(s:\mathbb{N}) < u\,.\,f)\,a \rightsquigarrow \forall(s:\mathbb{N}) < u\,.\,r$. However, additional reduction rules complicate any analysis of our language. Therefore, we propose the following definitions that allow us to define a single type-preserving reduction rule E-LocalApp.

Definition 17 (Syntactic Quantifier Context)**.** *The syntactic quantifier context $\overline{Q}$ is defined as follows:*

$$\overline{Q} ::= [\cdot] \mid \forall(x:\mathbb{N})\,.\,[\cdot] \mid \forall(x:\mathbb{N}) < h\,.\,[\cdot]$$

Definition 18 (Syntactic Quantifier Context Application)**.** *The application $\overline{Q}[]$ of a syntactic quantifier context $\overline{Q}$ to a term t^* is defined as follows:*

$$\begin{aligned}[\cdot][t^*] &:= t^*\\ (\forall(x:\mathbb{N})\,.\,[\cdot])\,[t^*] &:= \forall(x:\mathbb{N})\,.\,t^*\\ (\forall(x:\mathbb{N}) < h\,.\,[\cdot])\,[t^*] &:= \forall(x:\mathbb{N}) < h\,.\,t^*\end{aligned}$$

Our identity function for boolean lists of arbitrary size matches the pattern $id_\forall = \overline{Q}[f]$ for a quantifier context $\overline{Q}$ and a λ-term f.

We can apply reduction rule E-LocalApp to reduce terms of this pattern. By applying reduction rule E-LocalApp, we can reduce applications of the pattern $\overline{Q}[\lambda x:(B,[s]).t]\,v$ to $\overline{Q}[t[x\mapsto v]]$. Accordingly, a reduction step has the form $\big(\langle\overline{Q}[\lambda x:(B,[s]).t]\,v\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle\overline{Q}[t[x\mapsto v]]\rangle_{\mathcal{I}},[l]\big)$. The rule's premises require $\overline{Q}[\lambda x:(B,[s]).x]$ and a to be values. Otherwise both have to be reduced further before E-LocalApp can be applied.

Note that this application preserves the quantifier context $\overline{Q}$. As mentioned before, this is necessary to preserve the typing since $\overline{Q}$ might bind a variable occuring free in the reduced body $t[x\mapsto v]$. For instance, consider the function $g = \forall(s:\mathbb{N})\,.\,\lambda x:(\mathsf{Unit},[0]).\lambda x:(\mathsf{List}(\mathsf{Boolean},[0]),[s]).x$. The quantified variable s occurs free in its body. Reducing an application $g\,\mathsf{unit}$ mereley to the substituted function body and thereby eliminating the quantifier yields $\lambda x:(\mathsf{List}(\mathsf{Boolean},[0]),[s]).x$. This term contains s free and is therefore rejected by the type system. However, by applying E-LocalApp we reduce it to $\forall(s:\mathbb{N})\,.\,\lambda x:(\mathsf{List}(\mathsf{Boolean},[0]),[s]).x$. This is just $id_\forall$ and therefore accepted by the type system.

Quantifier Elimination As mentioned above the reduction of a function application $\overline{Q}[f]\,a$ preserves the quantifier context $\overline{Q}$. For instance, by applying E-LocalApp we reduce the term $\forall(s : \mathbb{N}) \,.\, \lambda x : (\mathsf{Unit}, [0]).\mathsf{true}$ to $\forall(s : \mathbb{N}) \,.\, \mathsf{true}$. The quantifier is unnecessary since s does not occur free in its scope true. For the same reason, $\forall(s : \mathbb{N}) \,.\, \mathsf{true}$ is not a value according to definition 9.

The reduction rule E-ForAllElim allows us to eliminate universal quantifiers which bind a variable not occuring free in its scope. Thereby, we can reduce $\forall(s : \mathbb{N}) \,.\, \mathsf{true}$ to the value true. A reduction step has the form $(\langle \forall(s:\mathbb{N}) \,.\, t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t\rangle_{\mathcal{I}}, [l])$. The rule's premise $s \notin FAV(t)$ ensures that we do not eliminate a quantifier needed by the type system to accept term t. Analogous, we can apply reduction rule E-BoundedForAllElim to eliminate unneeded restricted quantifiers $\forall(s:\mathbb{N}) < u$.

Let-terms allow us to make the result of some comptation available via a variable in a limited scope. We can reduce a term $\mathsf{let}\, x : T := v \,\mathsf{in}\, t_s$ where v is a value by rule E-Let to $t_s[x \mapsto v]$. Thereby, every occurence of x within in scope t_s is replaced by the value v. A reduction step has the form $(\langle \mathsf{let}\, x : T := t \,\mathsf{in}\, t_s\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_s[x \mapsto v]\rangle_{\mathcal{I}}, [l])$.

According to definition 15, this substitution respects other variables' scopes inside t_s. This holds for variables introduced by other let-terms as well as for such introduced via λ-terms. For instance, the term

```
let $x$ : $T$ := unit in
    let $y$ : $T'$ := true in
        $x$
```

reduces to $\mathsf{let}\, y : T_y := \mathsf{true}\,\mathsf{in}\,\mathsf{unit}$ and then to unit. In contrast, consider the term

```
let $x$ : $T$ := unit in
    let $x$ : $T'$ := true in
        $x$
```

Here, the outer x (bound to unit) is shadowed by the inner one (bound to true). Since substitution respects that, the term reduces to $\mathsf{let}\, x : T' := \mathsf{true}\,\mathsf{in}\, x$ and then to true.

According to the premise of rule E-Let, we can only apply it to reduce let-terms where the bound computation is a value. That is, before we can apply it to a term $\mathsf{let}\, x : T := t \,\mathsf{in}\, t_s$, we have to reduce the bound computation t to a value v.

Eager Reduction We earlier saw that in order to reduce a function application $(\lambda x : (B, [s]).t_s)\, a$ to $t_s[x \mapsto a]$ we have to ensure the argument a has already been reduced to a value. A similiar restriction holds when we want to reduce a term $\mathsf{let}\, x : T := a \,\mathsf{in}\, t_s$. In both cases we bind a computation a

to a variable x and make it accessible in a restricted scope t_s.

Our reduction relation $\rightsquigarrow$ implements an eager reduction strategy. That is, when we bind a term t to some variable x, we reduce t to a value v before we substitue every references to x by v. In consequence, we reduce the bound term exactly once, independent on the how often the corresponding variable is referenced.

In the next section we present reduction steps $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}}, [l'])$ involving remote communication. These steps increase the tracked runtime latency from l to $l' > l$. Consider a term t_r whose reduction causes latency l_r.

We could also choose a lazy reduction strategy. Then we would reduce a function application $(\lambda x : (B, [s]).t_s)\, t_r$ and a let-term $\mathsf{let}\, x : T \coloneqq t_r \,\mathsf{in}\, t_s$ to $t_s[x \mapsto t_r]$ without reducing t_r to a value first. Now consider the scope $t_s = \mathsf{if}\, x\, \{x\}\, \{x\}$ and assume t_r reduces to a $\mathsf{Boolean}$-value. Then we have to reduce t_r two times during the reduction of $t_s[x \mapsto t_r]$. This causes the latency $2 \cdot l_r$.

In contrast, with our eager reduction strategy, we reduce t_r exactly once to a value v_r, causing latency l_r. Then, we substitute v_r for x in t_s. The reduction of $t_s[x \mapsto v_r] = \mathsf{if}\, v\, \{v\}\, \{v\}$ does not cause any latency. This leaves us with an overall runtime latency of l_r instead of $2 \cdot l_r$.

This example shows, how we can reduce runtime latency by exploiting our eager reduction strategy. In cases where lazy reduction of a term t would be preferable, we can simulate it by wrapping t in function $f = \lambda x : (\mathsf{Unit}, [0]).t$ and replacing every occurence of t by $f\, \mathsf{unit}$.

Reduction Order So far, we presented several reduction rules (including E-Let, E-LocalApp, E-MatchSome) that required that some subterm had already been reduced to a value. Thereby, we assumed a specific reduction order which we need to make explicit in our defining rules. This order depends on the structure of the term we want to reduce. To allow for an easier order-defining rule we define an auxiliary construct.

Definition 19 (Syntactic Peer Context)**.** *Let x, t, T, h denote the corresponding syntactic definitions presented in Figure 2.2. Let v denote the definition of a value as presented in definition 9.*

A syntactic peer context E is defined synactically as follows:

$$\begin{aligned}E ::= {} & [\cdot] \mid \mathsf{some}\, E \mid \mathsf{cons}\, E\, t \mid \mathsf{cons}\, v\, E \mid E\, t \mid v\, E \mid \mathsf{fix}\, E \mid \\ & \mathsf{let}\, x : T := E\, \mathsf{in}\, t \mid \\ & E\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t\}\{\mathsf{none}\,(B, [h]) \Rightarrow t\} \mid \\ & E\, \mathsf{match}\{\mathsf{cons}\, x\, x \Rightarrow t\}\{\mathsf{nil}\,(B, [h]) \Rightarrow t\} \mid \\ & \mathsf{get}\, E.x \mid \mathsf{get}\, v.E \mid \mathsf{eval}\, t\, \mathsf{on}\, E \mid \\ & \mathsf{remoteCall}\, E.t\, t \mid \mathsf{remoteCall}\, v.E\, t \mid \mathsf{remoteCall}\, v.v\, E \mid \\ & \forall (x : \mathbb{N}) \,.\, E \mid \forall (x : \mathbb{N}) < h \,.\, E\end{aligned}$$

Note that every peer context E by definition contains exactly one hole $[\cdot]$.

Definition 20 (Syntactic Peer Context Application)**.** *The application $E[t^*]$ of a syntactic peer context E to a term t^* is defined by induction on the structure of E.*

Let E' be a syntactic peer context and let B and T be basic and fully annotated types, respectively. Furthermore, let t, t' be placed terms, h an arithmetic term, x, x' variables and v, v' values.

$$\begin{aligned}[\cdot][t^*] &:= t^* \\ (\mathsf{some}\, E')[t^*] &:= \mathsf{some}\,(E'[t^*]) \\ (\mathsf{cons}\, E'\,)[t^*] &:= \mathsf{cons}\,(E'[t^*]) \\ (\mathsf{cons}\, v\, E')[t^*] &:= \mathsf{cons}\, v\,(E'[t^*]) \\ (E'\, t)[t^*] &:= (E'[t^*])\, t \\ (v\, E')[t^*] &:= v\,(E'[t^*]) \\ (\mathsf{fix}\, E')[t^*] &:= \mathsf{fix}\,(E'[t^*]) \\ (\mathsf{let}\, x : T := E'\, \mathsf{in}\, t)[t^*] &:= \mathsf{let}\, x : T := (E'[t^*])\, \mathsf{in}\, t \\ (\mathsf{get}\, E'.t)[t^*] &:= \mathsf{get}\,(E'[t^*]).t \\ (\mathsf{get}\, v.E')[t^*] &:= \mathsf{get}\, v.(E'[t^*]) \\ (\mathsf{remoteCall}\, E'.t\, t')[t^*] &:= \mathsf{remoteCall}\,(E'[t^*]).t\, t' \\ (\mathsf{remoteCall}\, v.E'\, t)[t^*] &:= \mathsf{remoteCall}\, v.(E'[t^*])\, t \\ (\mathsf{remoteCall}\, v.v'\, E')[t^*] &:= \mathsf{remoteCall}\, v.v'\,(E'[t^*]) \\ (\mathsf{eval}\, t\, \mathsf{on}\, E')[t^*] &:= \mathsf{eval}\, t\, \mathsf{on}\,(E'[t^*]) \\ (\forall (x : \mathbb{N}) \,.\, E')[t^*] &:= \forall (x : \mathbb{N}) \,.\, (E'[t^*]) \\ (\forall (x : \mathbb{N}) < h \,.\, E')[t^*] &:= \forall (x : \mathbb{N}) < h \,.\, (E'[t^*])\end{aligned}$$

$$\begin{aligned}
&(E' \,\mathsf{match}\{\mathsf{some}\, x \Rightarrow t\}\{\mathsf{none}\,(B,[h]) \Rightarrow t'\})[t^*] \\
&\quad := \quad (E'[t^*])\,\mathsf{match}\{\mathsf{some}\, x \Rightarrow t\}\{\mathsf{none}\,(B,[h]) \Rightarrow t'\} \\
&(E' \,\mathsf{match}\{\mathsf{cons}\, x\, x' \Rightarrow t\}\{\mathsf{nil}\,(B,[h]) \Rightarrow t'\})[t^*] \\
&\quad := \quad (E'[t^*])\,\mathsf{match}\{\mathsf{cons}\, x\, x' \Rightarrow t\}\{\mathsf{nil}\,(B,[h]) \Rightarrow t'\}
\end{aligned}$$

As noted before, every peer context E contains exactly one hole $[\cdot]$ to fit in a term t^*. Its place depends on the context's structure as well as on which of its parts have already been reduced to a value. Note that whenever we view a term t as a context application $E[t^*]$ for a subterm t^*, both context E and t^* are uniquely determined.

For instance, we can view the term $\mathsf{cons\ true}\ \big((\lambda x : (\mathsf{Unit}, [0]).\mathsf{false})\,\mathsf{unit}\big)$ as a peer context $E_r = \mathsf{cons\ true}\ [\cdot]$ applied to the function term $t_\lambda = (\lambda x : (\mathsf{Unit}, [0]).\mathsf{false})\,\mathsf{unit}$. According to definition 20 this is only possible because true is a value. In contrast, since t_λ is no value we can only view the term $\mathsf{some}\, t_\lambda t_\lambda$ as context $E_l = \mathsf{cons}\,[\cdot]\,t_\lambda$ applied to t_λ.

We exploit this to formulate a reduction order for complex terms by lifting reduction steps to peer contexts. According to rule E-LocalContext, whenever we have a context application $E[t]$ and can reduce t to t', we can also reduce $E[t]$ to $E[t']$. Before we noted that decomposition of some term into a context application $E[t]$ is unique (if it exists). Hence, the rule E-LocalContext defines a unique reduction order.

For instance, the term t_λ reduces to false. According to E-LocalContext the application $E_r[t_\lambda]$ reduces to $E_r[\mathsf{false}] = \mathsf{cons\ true\ false}$. Meanwhile, the application $E_l[t_\lambda]$ reduces to $E_l[\mathsf{false}] = \mathsf{cons\ false}\ t_\lambda$.

In Section 2.4.3.1 we present reduction steps that cause remote communication and thereby cause runtime latency. When we lift a latency-increasing reduction step to a peer context, we have to consider the latency increase. Accordingly, whenever we have a reduction step $(\langle t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}}, [l'])$ we can apply rule E-LocalContext to a non-empty context $E[t]$ and obtain the reduction step $(\langle E[t] \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[t'] \rangle_{\mathcal{I}}, [l'])$. The rule's premise $E[t] \neq t$ forbids application to an empty context $E = [\cdot]$. Thereby, we avoid diverging applications of E-LocalContext.

2.4.3.2 Remote Reduction

In the following we explain the reduction rules presented in Figure 2.5. These rules define reduction steps involving remote communication. Hence, we call them *remote reduction rules* and *remote reduction steps*, respectively. In contrast to the runtime-latency-preserving rules presented in Section 2.4.3.1, most of the rules we present in the following are latency-increasing. That is, they have the form $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}}, [l'])$ for $l' > l$.

In reality remote communication can fail due to message loss or timeout and is therefore not deterministic from the sender's point-of-view. The remote reduction rules we present here respect that and are therefore in part also non-deterministic.

Suppose a client sends a message to some server and waits for the response. Considering that in this setting messages might get lost brings us to three different scenarios:

1. The message from client to server is transmitted and received successfully.
   (a) The server sends a response to the client and the latter receives it successfully.
   (b) The server sends a response to the client but it gets lost.
2. The message from client to server is lost. Hence, the client will receive no response from the server.

These three scenarios differ from a global perspective. From the client's point-of-view, however, we cannot observe any difference between scenario 1b and 2. In both cases the client never hears back from the server.

The Structure of Remote Communication To simplify our reduction rules, we do not differentiate between scenarios 1b and 2, either. Instead we only model the latter one. That is, in a reduction sequence representing the setting above the first message from client to server is always transmitted successfully.

Let $\mathcal{I} \subseteq \mathcal{I}^P$ be set of P-instances and $p' \in \mathcal{I}^{P'}$ a P'-instance with peer types $P \neq P'$. Assume that P and P' are tied according to tie context $\mathcal{T}$. Further, let term t represent some request to be send from the $\mathcal{I}$ to p' and l the runtime-latency tracked so far.

In general, a reduction sequence representing remote communication has the following structure:

1. **Sending the request:**
   There are different forms of remote requests and corresponding reduction rules which we explain later in detail. However, independent of the concrete request they all lead to a reduction step of the

$$\frac{(\langle t\rangle_{\mathcal{I}'},[l_r]) \overset{\mathcal{I}'}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}'},[l'_r])}{(\langle E[(\langle t\rangle_{\mathcal{I}'},[l_r])]\rangle_{\mathcal{I}},[l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[(\langle t'\rangle_{\mathcal{I}'},[l'_r])]\rangle_{\mathcal{I}},[l])} \text{(E-RemoteContext)}$$

$$\frac{\mathit{value}(v) \qquad \mathcal{P}(p') = P' \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad P \neq P'}{\big(\langle E[\big(\langle v\rangle_{\{p'\}},[l']\big)]\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[\mathsf{some}\, v]\rangle_{\mathcal{I}},[l + l' + \mathcal{L}(P',P)])}$$
(E-RemoteResultSuccess)

$$\frac{\mathit{value}(v) \qquad \mathcal{P}(p') = P' \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad P \neq P'}{\big(\langle E[\big(\langle v\rangle_{\{p'\}},[l']\big)]\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[\mathsf{none}\,(\beta(v),[\sigma(v)])]\rangle_{\mathcal{I}},[l + l' + \mathcal{L}(P',P)])}$$
(E-RemoteResultFail)

$$\frac{\mathit{value}(v) \qquad \forall p \in \mathcal{I}' : \mathcal{P}(p) = P \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P}{(\langle E[(\langle v\rangle_{\mathcal{I}'},[l'])]\rangle_{\mathcal{I}},[l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[v]\rangle_{\mathcal{I}},[l + l'])}$$
(E-LocalContextElim)

$$\frac{\mathit{value}(p') \qquad \mathit{value}(v) \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad \mathcal{P}(p') = P' \qquad P \neq P'}{\big(\big\langle \mathsf{get}\, p'.v\big\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\big\langle \big(\langle v\rangle_{\{p'\}},[0]\big)\big\rangle_{\mathcal{I}},[l + \mathcal{L}(P,P')]\big)}$$
(E-Get)

$$\frac{\mathit{value}(p') \qquad \mathit{value}(v_f) \qquad \mathit{value}(v_a) \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad \mathcal{P}(p') = P' \qquad P \neq P'}{\big(\big\langle \mathsf{remoteCall}\, p'.v_f\, v_a\big\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\big\langle \big(\langle v_f\, v_a\rangle_{p'},[0]\big)\big\rangle_{\mathcal{I}},[l + \mathcal{L}(P,P')]\big)}$$
(E-RemoteApp)

$$\frac{\mathit{value}(p') \qquad \forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad \mathcal{P}(p') = P' \qquad P \neq P'}{\big(\big\langle \mathsf{eval}\, t\, \mathsf{on}\, p'\big\rangle_{\mathcal{I}},[l]\big) \overset{\mathcal{I}}{\rightsquigarrow} \big(\big\langle \big(\langle t\rangle_{\{p'\}},[0]\big)\big\rangle_{\mathcal{I}},[l + \mathcal{L}(P,P')]\big)}$$
(E-PlacedEval)

Figure 2.5: Remote Reduction Rules for Placed Terms

form $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle\big(\langle t_r\rangle_{\{p'\}}, [0]\big)\rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P')]\big)$. It expresses that all peer instances $p \in \mathcal{I}$ send the request encoded by t to p' where they are received successfully. The time these messages need to reach their recipient p' depends on the connection from peer P to P'. Accordingly, the reduction step increases the tracked runtime latency by the weight $\mathcal{L}(P, P')$ assigned to this connection.

The term t_r represents the computation producing the response from p'. The nested context $\big(\langle t_r\rangle_{\{p'\}}, [0]\big)$ expresses that the computation t_r is to be reduced on the remote peer instance p' and that its reduction starts without any latency initially tracked on p'.

2. **Computing the response:**
If t_r is a value it can be transmitted directly. Otherwise we first reduce it on p' to a value. Let v_r denote the computed reponse. The computation might involve further remote communication initiated by p' causing additional latency. Let l_r be the runtime-latency tracked on p' by the time v_r has been computed. Then the context representing the remote computation's result is $\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)$.

3. **Transmitting the response:**
After the response $\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)$ has been computed on p', it is sent back to the peer instances $\mathcal{I}$. The time the response needs to reach its recipients depends on the connection from P' to P. A corresponding reduction step has the form $\big(\langle\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)\rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P')]\big) \overset{\mathcal{I}}{\rightsquigarrow}$ $(\langle ...\rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P') + l_r + \mathcal{L}(P', P)])$. It takes place on the receivers' site $\mathcal{I}$ and hence represents the receivers' perspective. The time the receivers wait for the response consists of the time needed for transmission and also on the delay caused by the response's remote computation. Hence, the step increases the runtime-latency tracked on $\mathcal{I}$ by the remote-latency l_r and by the weight $\mathcal{L}(P', P)$ put on the connection from P' to P.

Our reduction rules model the possibility of transmission failure during this last step of remote communication. The corresponding rules E-RemoteResultSuccess and E-RemoteResultFail do not differ in their applicability but only in the reduction result. The received value is not the originally sent response but an Option-value explicitly encoding the transmission's success or failure. This introduces non-determinism to our reduction semantics.

Requesting Placed Values As explained is Section 2.4.2 we can use placement terms $\mathsf{place}\, x : T \coloneqq t\, \mathsf{on}\, P'\, \mathsf{in}\, q'$ to place a computation t on a peer P'. Term t is then reduced on all P'-instances to some value v_x and bound to variable x. Hence, we can request the value bound to x from any such instance $p' \in \mathcal{I}^{P'}$ via the term $\mathsf{get}\, p'.x$.

We can reduce a term of the form $\mathsf{get}\, p'.v$ for some value v by applying reduction rule E-Get. This leads us to the reduction step $(\langle \mathsf{get}\, p'.v \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle \big(\langle v \rangle_{\{p'\}}, [0] \big) \rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P')] \big)$. It represents the successful transmission of the request for v to peer instance p'. The time the request needs to reach its recipient depends on the connection from peer P to P'. Hence, the step increases the tracked runtime-latency by the weight $\mathcal{L}(P, P')$ put on this connection.

The premises of E-Get require all instances $p \in \mathcal{I}$ to belong to peer type P and p' to belong to P' for $P \neq P'$. This way, we ensure that the term's reduction indeed involves a remote message from P to P' and that the latency increase by $\mathcal{L}(P, P')$ is correct.

The rule's premises also require p' and v to be values. That is, before we can apply E-Get to a term $\mathsf{get}\, t_{p'}.t_v$ we have to reduce $t_{p'}$ and t_v to values. Depending on the terms we can either use the rule E-LocalContext previously presented in Section 2.4.3.1 or the rule E-RemoteContext which we explain later in this section.

Remote Calls have the form $\mathsf{remoteCall}\, p'.f\, a$. Similiar to get-terms they allow us to access some function f placed on a remote peer P' via any P'-instance $p' \in \mathcal{I}^{P'}$. The difference, however, is that the accessed value f is not transmitted to the call site. Suppose, we reduce the remote call on a set of P-instances $\mathcal{I} \subseteq \mathcal{I}^P$ for $P \neq P'$. Every instance $p \in \mathcal{I}$ sends the local argument a to remote peer instance p'. Afterwards, p' locally reduces the application $f\, a$ and sends the result back to $\mathcal{I}$. Again, our reduction relation assumes that the request's transmission is successful.

We can use rule E-RemoteApp to reduce a remote call $\mathsf{remoteCall}\, p'.f\, a$ where all arguments p', f, a are values. The following reduction step has the form $(\langle \mathsf{remoteCall}\, p'.f\, a \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle \big(\langle f\, a \rangle_{\{p'\}}, [0] \big) \rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P')] \big)$. It expresses that peer instances $\mathcal{I}$ successfully transmit the value a and their request for the result of $f\, a$ to p'. This transmission increases the tracked runtime-latency by the weight $\mathcal{L}(P, P')$ put on the involved connection.

The peer context $\big(\langle f\, a \rangle_{\{p'\}}, [0] \big)$ expresses that the application $f\, a$ has to be reduced on p'. The remote reduction starts without any runtime-latency tracked on p'. As mentioned before, any potential latency arising during this reduction is added to the latency tracked on $\mathcal{I}$ when the result is sent. This happens regardless of whether the result is received is successfully or not.

The premises of E-RemoteApp ensure that the rule is only applied when the involved arguments are values. Otherwise we have to reduce them first. The premises also require that the caller site $\mathcal{I}$ only consists of P-instances and p' belongs to a remote peer P' with $P \neq P'$. This way, we ensure that the latency increase $\mathcal{L}(P, P')$ is correct.

Placed Computations A computation can involve multiple remote calls to the same peer instance $p' \in \mathcal{I}^{P'}$. Consider the following computation $t_{sep} = \mathsf{cons}\,(\mathsf{remoteCall}\,p'.f\,\mathsf{true})\,(\mathsf{cons}\,(\mathsf{remoteCall}\,p'.f\,\mathsf{false})\,\mathsf{nil}\,(...,[...]))$. Here, we request two remote invokations of f from p'. During the term's reduction both requests as well as their results are sent independently. Let l_1 and l_2 denote the tracked latency values arising during the reduction of $f\,\mathsf{true}$ and $f\,\mathsf{false}$, respectively, on p'. Then, the reduction of t_{sep} on a location $\mathcal{I} \subseteq \mathcal{I}^P$ with $P \neq P'$ accumulates to a total latency of $\mathcal{L}(P, P') + l_1 + \mathcal{L}(P', P) + \mathcal{L}(P, P') + l_2 + \mathcal{L}(P', P)$.

We can avoid the overhead caused by the separate transmissions. First, we compine the remote applications by requesting the result of a combined computation $t_{comb} = \mathsf{cons}\,(f\,\mathsf{true})\,(\mathsf{cons}\,(f\,\mathsf{false})\,\mathsf{nil}\,(...,[...]))$. Then, we can request the result of t_{comb} placed p' in form of the term $\mathsf{eval}\,t_{comb}\,\mathsf{on}\,p'$. Since only one request and result have to be transmitted, the reduction leads to an overall latency of $\mathcal{L}(P, P') + l_1 + l_2 + \mathcal{L}(P', P)$.

Reduction rule E-PlacedEval allows us to reduce a term of the form $\mathsf{eval}\,t\,\mathsf{on}\,p'$ on a set of peer instances $\mathcal{I} \subseteq \mathcal{I}^P$. It produces the reduction step $(\langle \mathsf{eval}\,t\,\mathsf{on}\,p'\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\leadsto} \big(\langle\big(\langle t\rangle_{\{p'\}}, [0]\big)\rangle_{\mathcal{I}}, [l + \mathcal{L}(P, P')]\big)$. This step expresses that the request has been sent successfully to p'. Hence, it increases the tracked runtime-latency by $\mathcal{L}(P, P')$. The peer context $\big(\langle t\rangle_{\{p'\}}, [0]\big)$ expresses that the transmitted computation t has to be reduced on p'. The latency this remote reduction entails is added to the latency tracked on $\mathcal{I}$ when the result is transmitted.

The premesises of E-PlacedEval require require $\mathcal{I}$ to be a set of P-instances and p' to be a P'-instance for $P \neq P'$. This ensures the correctness of the latency increase by $\mathcal{L}(P, P')$.

In particular p' has to be a value. Hence, for a term $\mathsf{eval}\,t\,\mathsf{on}\,t'_p$ where t'_p is no value, we have to reduce t'_p on $\mathcal{I}$ before we can apply E-PlacedEval.

Transmitting Results In practice, remote communication can fail and hence seems non-deterministic from a restricted point-of-view. As explained above, our reduction rules model a simplified setting. According to $\rightsquigarrow$ sending a remote request always succeeds and only the response's transmission might fail.

A request's response that is ready for transmission always has the form $\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)$ for a value v_r. Here, v_r and p' represent the response and the peer instance sending it. l_r is the latency tracked during the computation of v_r on p'.

The transmission of v_r can succeed or fail. To make the possibility explicit in our reduction semantics, the sender of the request does not receive the actual response v_r but an Option-value. Transmission success and failure are modelled by the reduction rules E-RemoteResultSuccess and E-RemoteResultFail, respectively. Both can be applied to in the same settings but differ in the received Option-value. This makes the reduction of terms involving remote requests non-deterministic. When we apply E-RemoteResultSuccess, the response's transmission succeeds and the request's sender receives the value $\mathsf{some}\, v_r$. Similiarly, applying the rule E-RemoteResultSuccess lets the transmission fail and she receives none-value to represent the failure.

Remote requests can occur as part of larger local terms. Consider for instance the term $(\mathsf{get}\, p'.x)\, \mathsf{match}\{\mathsf{some}\, x' \Rightarrow t_s\}\{\mathsf{none}\,(B, [s]) \Rightarrow t_n\}$ containing the request $\mathsf{get}\, p'.x$. The rules E-RemoteResultSuccess and E-RemoteResultFail must respect the context in which a remote request occurs. For instance, an application of E-RemoteResultSuccess should reduce this example to $(\mathsf{some}\, v_r)\, \mathsf{match}\{\mathsf{some}\, x' \Rightarrow t_s\}\{\mathsf{none}\,(B, [s]) \Rightarrow t_n\}$.

We can reuse the syntactic peer contexts from definition 19 to define the reduction steps produced by E-RemoteResultSuccess and E-RemoteResultFail. Let $\mathcal{I} \subseteq \mathcal{I}^P$ be a set of P-instances and $p' \in \mathcal{I}^{P'}$ a P'-instance. Let further be E a peer context and as above let v_r be a value. We can reduce a context of the form $\big(\langle E[\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)]\rangle_{\mathcal{I}}, [l_c]\big)$ by applying E-RemoteResultSuccess or E-RemoteResultFail.

Transmission Success Rule E-RemoteResultSuccess produces the step $\big(\langle E[\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)]\rangle_{\mathcal{I}}, [l_c]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[\mathsf{some}\, v_r]\rangle_{\mathcal{I}}, [l_c + l_r + \mathcal{L}(P', P)])$. This step respects the context E and expresses that all peer instances $\mathcal{I}$ successfully receive the response to their request. Furthermore, this step increases the runtime-latency tracked on $\mathcal{I}$ by the latency caused on p' during the response's computation and by weight $\mathcal{L}(P', P)$ assigned to the connection from P' to P.

According to the rule's premises we can only apply it v_r is a value, all instances in $\mathcal{I}$ belong to peer P and p' belongs to P' for $P \nleq P'$. This ensures that the response v_r has been fully reduced on p' before we transmit it to $\mathcal{I}$ and that the latency increase $\mathcal{L}(P', P)$ is correct.

Transmission Failure The reduction rule E-RemoteResultFail is similiar to E-RemoteResultSuccess. Both have the same premises and share the left side of the reduction step. Hence, they can both be applied to reduce the same terms. As mentioned above they only differ in the transmission result. E-RemoteResultFail produces the reduction step
$\big(\langle E[\big(\langle v_r\rangle_{\{p'\}}, [l_r]\big)]\rangle_{\mathcal{I}}, [l_c]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[\mathsf{none}\,(\beta(v_r), [\sigma(v_r)])]\rangle_{\mathcal{I}}, [l_c + l_r + \mathcal{L}(P', P)])$.
It represents that the peer instances $\mathcal{I}$ do not receive a response to their request. The type system we propose in Section 2.5 requires type and size annotations for none-values. Hence, β and σ are functions that map the value v_r to the type and size, respectively, that the type system assigns to v_r. A correct annotation is necessary for the type system to accept the reduction result.

Eliminating Nested Local Contexts Above we saw that nested remote contexts $\big(\langle E[\big(\langle v\rangle_{\{p'\}}, [l']\big)]\rangle_{\mathcal{I}}, [l]\big)$ represent results of remote requests awaiting transmission. During reduction we can also encounter nested local contexts $(\langle E[(\langle v\rangle_{\mathcal{I}'}, [l'])]\rangle_{\mathcal{I}}, [l])$ where $\mathcal{I}$ and $\mathcal{I}'$ belong to the same peer. Consider the program

$$\begin{array}{l}\mathsf{place}\ x : T \mathrel{:=} \mathsf{true}\,\mathsf{on}\,P\,\mathsf{in}\\ \quad \mathsf{place}\ y : T' \mathrel{:=} (\mathsf{if}\,x\,\{\mathsf{false}\}\,\{\mathsf{true}\})\,\mathsf{on}\,P\,\mathsf{in}\\ \quad\quad \mathsf{end}\end{array}$$

Here, we place a variable x on peer P and bind it to the value unit. Then the same peer binds the value of x to another variable y. Clearly, no remote communication is involved everything happens on one single peer. According to the reduction rules for placement terms presented in Section 2.4.2, the program is (in multiple steps) reduced to $\mathsf{place}\,y : T' \mathrel{:=} (\mathsf{if}\ (\langle\mathsf{true}\rangle_{\mathcal{I}^P}, [0])\ \{\mathsf{false}\}\,\{\mathsf{true}\})\,\mathsf{on}\,P\,\mathsf{in}\,\mathsf{end}$. Afterwards we can apply rule E-LocalContextIntro. Thereby we obtain the placement term:
$\mathsf{place}\,y : T' \mathrel{:=} (\langle(\mathsf{if}\ (\langle\mathsf{true}\rangle_{\mathcal{I}^P}, [0])\ \{\mathsf{false}\}\,\{\mathsf{true}\})\rangle_{\mathcal{I}^P}, [0])\ \mathsf{on}\,P\,\mathsf{in}\,\mathsf{end}$.
Reducing the placed computation further leads us to the nested local context $(\langle(\langle\mathsf{false}\rangle_{\mathcal{I}^P}, [0])\rangle_{\mathcal{I}^P}, [0])$. Clearly we should be able to reduce this term to $(\langle\mathsf{false}\rangle_{\mathcal{I}^P}, [0])$ without involving any remote communication. Reduction rule E-LocalContextElim allows us to do this.

In general, a nested local context has the form $(\langle E[(\langle t\rangle_{\mathcal{I}'}, [l'])]\rangle_{\mathcal{I}}, [l])$ for some peer context E, a peer P and $\mathcal{I}, \mathcal{I}' \subseteq P$. If the nested term t is no value we have to reduce t further using the rule E-LocalContext until we reach a value v. Otherwise we can eliminate the local context by applying E-LocalContextElim. This produces a reduction step $(\langle E[(\langle v\rangle_{\mathcal{I}'}, [l'])]\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[v]\rangle_{\mathcal{I}}, [l + l'])$. By eliminating the nested context this step also transfers the runtime-latency l' tracked by the inner context into the remaining one.

Remote Context In Section 2.4.3.1 we explained how we can use the rule E-LocalContext to lift local reduction steps $(\langle t\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}}, [l'])$ to a peer context E. The resulting reduction step then is $(\langle E[t]\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[t']\rangle_{\mathcal{I}}, [l'])$. We can apply the same concept to define a reduction rule E-RemoteContext lifting remote reduction steps to peer contexts.

Let $\mathcal{I} \subseteq \mathcal{I}^{P}, \mathcal{I}' \subseteq \mathcal{I}^{P'}$ be sets of P- and P'-instances, respectively, for $P \neq P'$. Suppose we have a term $E[(\langle t\rangle_{\mathcal{I}'}, [l_r])]$ nested in a peer context that is to be reduced on $\mathcal{I}$. Suppose further that we can make a remote reduction step $(\langle t\rangle_{\mathcal{I}'}, [l_r]) \overset{\mathcal{I}'}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}'}, [l'_r])$. Then we can apply E-RemoteContext to make the reduction step $(\langle E[(\langle t\rangle_{\mathcal{I}'}, [l_r])]\rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle E[(\langle t'\rangle_{\mathcal{I}'}, [l'_r])]\rangle_{\mathcal{I}}, [l])$.

2.4.3.3 Recursion

In order to formulate realistic programs we need recursion. $\lambda^{\text{lat}'}$offers a fixpoint operator fix to formulate recursive computations. We start by explaining the standard way of adding a fixpoint operator to the typed λ-calculus as found in [17]. Afterwards we modify this definition to match our needs and present the corresponding reduction rule E-FixApp.

Diverging Reduction In the standard implementation, the fixpoint operator fix expects a function of the form $f = \lambda g : T.t$. Here, f expects us to supply a function g of type T in order to compute a function which again is of the same type T. We can intuitively see parameter g as a continuation for f. An application $\mathsf{fix}\, f$ is then expanded to $f\,(\mathsf{fix}\, f)$. The fixpoint operator turns f into its own continuation and thereby transforms it into a recursive function.

This implementation allows the formulation of diverging terms. Consider the identity function $id = \lambda g : T.g$. The application $\mathsf{fix}\, id$ expands to $id\,(\mathsf{fix}\, id)$, which we reduce to $id[g \mapsto (\mathsf{fix}\, id)] = \mathsf{fix}\, id$.

As tshown in Section 2.4.3.2, the reduction of some terms involves remote communication and therefore increases the tracked runtime latency. Let b_{lat} be such a term reducing to a $\mathsf{Boolean}$-value on some location $\mathcal{I}$ and thereby causing latency l_b. Consider the identity function $id_{lat} = \lambda g : T.\mathsf{if}\; b_{lat}\, \{g\}\, \{g\}$.

Independent of the concrete value b_{lat} reduces to, an application $id_{lat}\, a$ reduces to a and thereby causes latency l_b. Application of the fixpoint operator turns it into a diverging function which causes an infinite amount of latency during its reduction. That is, a context $(\langle \mathsf{fix}\, id_{lat} \rangle_{\mathcal{I}}, [0])$ reduces (in multiple steps) to $(\langle \mathsf{fix}\, id_{lat} \rangle_{\mathcal{I}}, [l_b])$, then to $(\langle \mathsf{fix}\, id_{lat} \rangle_{\mathcal{I}}, [2 \cdot l_b])$, then to $(\langle \mathsf{fix}\, id_{lat} \rangle_{\mathcal{I}}, [3 \cdot l_b])$ and so forth.

In Section 2.5 we present a type system which extracts an upper bound for a term's runtime latency. This is not possible for diverging functions with latency-increasing recursive steps such as id_{lat}. Therefore, we restrict our language to terminating recursive functions.

In Section 2.5 we also introduce a static notion of size and use it to statically guarantee termination. The proposed type system, only allows the formulation of recursive functions f, that when applied to an argument a of size s_a, take the recursive step on a smaller argument a' of size $s'_a < s_a$. Since sizes are finite, we know the application $f\, a$ terminates after a finite number of recursive steps. In general, the type system we propose in Section 2.5 only accepts terminating computations. Hence, λ^{lat} and $\lambda^{\text{lat}'}$are total. In the following we show how we formulate such functions and how we implement the reduction of a fixpoint application $\mathsf{fix}\, f$.

Bound-increasing Functions When we formulate a function $\lambda x : (B, [s]).t$, we specify the type of argument B and the size s we expect it to be applied to. Our type system only allows applications $f\, a$ where the argument type and size match the ones expected by f. As mentioned before, thereby we prevent stuck reductions.

We can use this to ensure size-decreasing recursion. Consider some function type $B^{\rightarrow}$ which contains a free variable s and represents functions that transform a $\mathsf{Boolean}$-list of size s into a $\mathsf{Boolean}$-value. Then $\forall (s : \mathbb{N}) \,.\, B^{\rightarrow}$ represents functions that can handle any size s. Similiarly, $\forall (s : \mathbb{N}) < u \,.\, B^{\rightarrow}$ represents functions that only handle arguments of size $s < u$. Now consider the function

$$
\begin{aligned}
conj_{u \to u+1} = \;& \forall (u : \mathbb{N}). \\
&\quad \lambda g : (\forall (s : \mathbb{N}) < u \,.\, B^{\rightarrow},\ [0]). \\
&\qquad \forall (s : \mathbb{N}) < u + 1. \\
&\qquad\quad \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]),\ [s]). \\
&\qquad\qquad x\ \ \mathsf{match}\ \{\mathsf{cons}\, x_h\, x_t \Rightarrow \mathsf{if}\, x_h\, \{g\, x_t\}\, \{\mathsf{false}\}\} \\
&\qquad\qquad\qquad \{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{true}\}
\end{aligned}
$$

For an arbitrary bound u, this function expects a continuation g which can process lists of size $s < u$ and a list x of size $s < u + 1$. In particular, a passed list's size might reach u, which makes it too large for g to process. The body

$$\begin{aligned} x \ \mathsf{match}\ & \{\mathsf{cons}\, x_h\, x_t \Rightarrow \mathsf{if}\, x_h\, \{g\, x_t\}\, \{\mathsf{false}\}\} \\ & \{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{true}\} \end{aligned}$$

deconstructs the passed list into a head x_h and a tail x_t. If the head equals true, the tail x_t is passed to the continuation g. Note that since x_t has one element less than the original list, it is small enough for g to process. Otherwise, the function directly returns false without invoking its continuation. Similiarly, if the original list x is empty, the function returns true without invoking g.

Alternatively, we can view $conj_{u \to u+1}$ as a function with only a single parameter g, that again produces a function with a single parameter x. Then, for any upper bound u, $conj_{u \to u+1}$ expects a function g which can process only arguments of size $s < u$ and transforms it into one that can process arguments up to size $s < u + 1$. We call such functions *bound-increasing* and give a precise definition in Section 2.5.

Building Terminating Recursive Functions Let $conj_{<u}$ denote the function performing elementwise conjunction on Boolean-lists of size up to $u - 1$. Then we can use $conj_{u \to u+1}$ to build $conj_{<u}$ for any $u \in \mathbb{N}$. Independend of the supplied continuation $\widehat{g}$, the application $conj_{u \to u+1}\, \widehat{g}$ maps empty lists to false. (This complies with the general definition of conjunction for an empty list $\bigwedge_{b \in \varnothing} b = \mathsf{false}$.) Hence, we can define $conj_{<1}$. For any $u \in \mathbb{N}$, we can build $conj_{<u}$ by applying $conj_{u \to u+1}$ exactly u times, that is $conj_{<u} = \underbrace{conj_{u \to u+1} \dots conj_{u \to u+1}}_{u \text{ times}}\, \widehat{g}$.

Since the start value $\widehat{g}$ does not matter, we can also set $\widehat{g} = conj_{u \to u+1}$. Then, $conj_{u \to u+1}$ is its own continuation. Using the fixpoint operator's standard implementation, explained above, we can build a recursive and unrestricted conjunction $\mathsf{fix}\, conj_{u \to u+1}$.

Let a be a Boolean-list of size s_a and consider the function application $(\mathsf{fix}\, conj_{u \to u+1})\, a$ (using the standard implementation for fix). It expands to $(conj_{u \to u+1}\, (\mathsf{fix}\, conj_{u \to u+1}))\, a$ and then (in muiltiple steps) reduces to

$$\begin{aligned} a \ \mathsf{match}\ & \{\mathsf{cons}\, x_h\, x_t \Rightarrow \mathsf{if}\, x_h\, \{(\mathsf{fix}\, conj_{u \to u+1})\, x_t\}\, \{\mathsf{false}\}\} \ . \\ & \{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{true}\} \end{aligned}$$

If a is the empty list, the term reduces to true. Otherwise we decompose it into its head x_h and tail x_t. If x_h equals false, the whole term reduces to false. Otherwise, it reduces to $(\mathsf{fix}\, conj_{u \to u+1})\, x_t$. That is, we take a recursive step on a list of size $s_a - 1$. We immediately see that independent of the concrete value a and its size s_a the application $(\mathsf{fix}\, conj_{u \to u+1})\, a$ terminates.

A Well-typed Fixpoint Operator We can generalize the above observation to all bound-increasing functions f and fixpoint applications $\mathsf{fix}\,f$. The type system we propose in Section 2.5 only allows such applications for bound-increasing functions f. Thereby, we ensure that the resulting recursive function implements a terminating computation like in the previous example.

As mentioned before, we also use the type system to avoid stuck computations. Hence, it only allows applications $g\,a$ where a matches the expectations of g. This particularly holds for the expected size. That is, it accepts an application $(\lambda x : (B, [s]).t)\,a$ for an argument a of size s_a only if one of the following holds:

- $s = s_a$ (modulo representation)
- s is universally quantified by a quantifier application $\forall(s : \mathbb{N})$
- s is boundedly quantified by a quantifier application $\forall(s : \mathbb{N}) < u$ and $s_a < u$.

Consider the example $\mathsf{fix}\,conj_{u\to u+1}$, again, with the standard implementation of fix. It expands to $conj_{u\to u+1}\,(\mathsf{fix}\,conj_{u\to u+1})$ and then reduces to

$$\begin{aligned} c = \quad & \forall(u : \mathbb{N}). \\ & \quad \forall(s : \mathbb{N}) < u + 1. \\ & \quad\quad \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]). \\ & \quad\quad\quad x \ \mathsf{match} \ \{\mathsf{cons}\, x_h\, x_t \Rightarrow \mathsf{if}\, x_h \,\{(\mathsf{fix}\, conj_{u\to u+1})\, x_t\}\, \{\mathsf{false}\}\} \\ & \quad\quad\quad\quad\quad\quad \{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{true}\} \end{aligned}$$

An application $c\,a$ does not match any of the valid cases listed above. Hence, the type system rejects it. One could argue that a bounded quantification $\forall(u : \mathbb{N}) \,.\, \forall(s : \mathbb{N}) < u$ where the bound u is universally quantified, is equivalent to a universal quantification $\forall(s : \mathbb{N})$ and that the type system should hence treat both equivalently. However, in practice more complicated cases like $\forall(u : \mathbb{N}) \,.\, \forall(z_1 : \mathbb{N}) < w_1 \ldots \forall(z_n : \mathbb{N}) < w_n \,.\, \forall(s : \mathbb{N}) < 2 \cdot u - 100$ or $\forall(u : \mathbb{N}) \,.\, \Big((B, [z]) \to ((\forall(s : \mathbb{N}) < u \,.\, B', [0]) \to (\mathsf{Unit}, [0], [0]), [0], [0])\Big)$ can occur. Treating all special cases in the type system would complicate its design significantly. However, we believe that the type system should be kept simple and concentrate on its original purpose: Extracting static bounds for a computation's runtime latency.

Eliminating Bounds Instead we propose a syntactical approach to eliminate problematic bounds during expansion of the fixpoint operator. That is, we first syntactically eliminate the bound u of s in the term $conj_{u\to u+1}$

and thereby obtain the function

$$
\begin{aligned}
\mathit{conj}_{\forall\to\forall} = \ & \forall(u:\mathbb{N}).\\
& \quad \lambda g : (\forall(s:\mathbb{N}) . B^{\to},\ [0]).\\
& \quad\quad \forall(s:\mathbb{N}).\\
& \quad\quad\quad \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]),\ [s]).\\
& \quad\quad\quad\quad x\ \ \mathsf{match}\ \ \{\mathsf{cons}\, x_h\, x_t \Rightarrow \mathsf{if}\, x_h\, \{g\, x_t\}\, \{\mathsf{false}\}\}\\
& \quad\quad\quad\quad\quad\quad\quad\quad \{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{true}\}
\end{aligned}
$$

Afterwards we expand $\mathsf{fix}\, \mathit{conj}_{u\to u+1}$ to $\mathit{conj}_{\forall\to\forall}\,(\mathsf{fix}\, \mathit{conj}_{u\to u+1})$. Since $\mathsf{fix}\, \mathit{conj}_{u\to u+1}$ is a function that can process lists of arbitrary size and $\mathit{conj}_{\forall\to\forall}$ is a function expecting one of this format as argument, our type system accepts this application.

We later see in Section 2.5 that bound-increasing function values have the form $\forall(u:\mathbb{N}) . \lambda x : (\forall(s:\mathbb{N}) < u . B, [0]).t$. Eliminating the bound within the argument type is straightforward, since the format $\forall(s:\mathbb{N}) < u . B$ is fixed and s is free in B. The body t, however, does not necessarily have the form $\forall(s:\mathbb{N}) < w(u) . t^*$ where s is free in t^* and $w(u)$ is the increased bound. For instance, the body t can have the form $\forall(s:\mathbb{N}) < u+1 . \forall(s:\mathbb{N}) < u+2 . t^*$. Here, eliminating the outermost quantifier's bound will leave the term $\forall(s:\mathbb{N}) . \forall(s:\mathbb{N}) < u+2 . t^*$ where the the occurences of s in t^* are still bounded by $u+2$. Therefore, we eliminate irrelevant quantifiers before we start to eliminate the relevant bounds.

However, the body t does not necessarily expose its quantifiers as the outermost subterms. It can also have a form that only when reduced leads to a term of structure $\forall(s:\mathbb{N}) < w(u) . t^*$. For instance, it can have the form $t = \mathsf{if}\, t' \{\forall(s:\mathbb{N}) < w(u) . t''\} \{\forall(s:\mathbb{N}) < w(u) . t'''\}$. Therefore, we choose an approach similiar to prenexing formulas in first-order logic. We pull the quantifiers $\forall(s:\mathbb{N}) < w(u)$ outside and avoid scope conflicts by renaming conflicting variables. For instance, the condition t' might contain free occurences of the variable s that are bound by some quantifier outside of the if-term (e.g. $\forall(s:\mathbb{N}) . \ldots \mathsf{if}\, t' \{\forall(s:\mathbb{N}) < w(u) . t''\} \{\forall(s:\mathbb{N}) < w(u) . t'''\}$). In this case pulling the quantifiers $\forall(s:\mathbb{N}) < w(u)$ from the branches infront of the if-term, would bind the free occurences of s in the condition t'. We can circumvent this confict by renaming the restricted variable s into a new one $\hat{s}$ not occuring anywhere in the if-term. This leaves us with the prenexed version $\forall(\hat{s}:\mathbb{N}) < w(u) . \mathsf{if}\, t' \{t''[s \mapsto \hat{s}]\} \{t'''[s \mapsto \hat{s}]\}$.

As mentioned before, a bound-increasing function value f has the form $f = \forall(u:\mathbb{N}) . \lambda x : (\forall(s:\mathbb{N}) < u . B, [0]).t$. After prenexing its body t, it has the form $\forall(u:\mathbb{N}) . \lambda x : (\forall(s:\mathbb{N}) < u . B, [0]).\forall(\widehat{s}:\mathbb{N}) < w(u) . t^*$, where $\widehat{s}$ is the potentially renamed size variable. With this structure given, we can eliminate the bounds in a straightforward manner, getting $f' = \forall(u :$

$\mathbb{N}) \,.\, \lambda x : (\forall (s : \mathbb{N}) \,.\, B, [0]).\forall (\widehat{s} : \mathbb{N}) \,.\, t^*$.

We see that for any bound-increasing function value f and its bound-eliminated form f', the fixpoint application $\mathsf{fix}\, f$ matches the argument type expected by f'. Accordingly, the type system accepts the application $f' \,(\mathsf{fix}\, f)$.

Later we present a reduction rule E-FixApp, that follows our approach and performs the expansion $\mathsf{fix}\, f \rightsquigarrow f' \,(\mathsf{fix}\, f)$. However, in order to define the rule's behaviour, we need some fixed function $BE : \mathit{deriv}(t) \to \mathit{deriv}(t)$ mapping the function value f to its bound-eliminated form $BE(f) = f'$. An important step towards this function is prenexing f successfully without changing its computational behaviour. Prenexing a $\lambda^{\mathrm{lat}'}$-term is much more complicated than prenexing a formula in first-order logic. Hence, we sketch the definition of a function $\mathit{prenex} : \mathbb{X} \times \mathit{deriv}(t) \to \mathbb{X} \times \mathit{deriv}(t)$ that for some size variable s and term t (i) pulls the innermost quantifier for s to the front and (ii) renames s where necessary to avoid scope conflicts. The result then is a tuple $(\widehat{s}, t')$. If t does not contain any bound quantifications $\forall (s : \mathbb{N}) < h$ to prenex, then $t' = t$. Otherwise, $t' = \forall (\widehat{s} : \mathbb{N}) < h \,.\, t^*$ is the prenexed version of t regarding s and $\widehat{s}$ is the potenially renamed size variable s.

Definition 21 (Prenex)**.** *The function $\mathit{prenex} : \mathbb{X} \times \mathit{deriv}(t) \to \mathbb{X} \times \mathit{deriv}(t)$ is defined by induction over the term structure.*

Let t be a term and $\hat{s}, \bar{s}$ be variables with $\hat{s} \neq \bar{s}$.

In the following let $\hat{s}''$ and $\hat{s}'''$ denote new variables not occuring in any subterm.

$$\begin{aligned}
\mathit{prenex}(\hat{s}, \mathsf{unit}) &:= (\hat{s}, \mathsf{unit})\\
\mathit{prenex}(\hat{s}, \mathsf{true}) &:= (\hat{s}, \mathsf{true})\\
\mathit{prenex}(\hat{s}, \mathsf{none}\,(B, [s])) &:= (\hat{s}, \mathsf{none}\,(B, [s]))\\
\mathit{prenex}(\hat{s}, \mathsf{some}\, t^*) &:= (\hat{s}, \mathsf{some}\, t^*)\\
\mathit{prenex}(\hat{s}, \mathsf{nil}\,(B, [s])) &:= (\hat{s}, \mathsf{nil}\,(B, [s]))\\
\mathit{prenex}(\hat{s}, \mathsf{cons}\, t^*\, t^{**}) &:= (\hat{s}, \mathsf{cons}\, t^*\, t^{**})\\
\mathit{prenex}(\hat{s}, x) &:= (\hat{s}, x)\\
\mathit{prenex}(\hat{s}, \lambda x : (B, [s]).t^*) &:= (\hat{s}, \lambda x : (B, [s]).t^*)\\
\mathit{prenex}(\hat{s}, \mathsf{fix}\, t^*) &:= (\hat{s}, \mathsf{fix}\, t^*)
\end{aligned}$$

$$
\begin{aligned}
prenex(\hat{s}, \mathsf{let}\, x : T := t^{*}\, \mathsf{in}\, t^{**}) &:= (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,(\mathsf{let}\, x : T := t^{*}\, \mathsf{in}\, t^{**\prime})) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}[x \mapsto t^{*}]) = prenex(\hat{s}, t^{**})[x \mapsto t^{*}] \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,t^{**\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \notin FAV(t^{*}) \\
&:= (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u\,.\,(\mathsf{let}\, x : T := t^{*}\, \mathsf{in}\, t^{**\prime}[\hat{s}' \mapsto \hat{s}''])) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}[x \mapsto t^{*}]) = prenex(\hat{s}, t^{**})[x \mapsto t^{*}] \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,t^{**\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \in FAV(t^{*}) \\
&:= (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,(\mathsf{let}\, x : T := t^{*\prime}\, \mathsf{in}\, t^{**})) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}[x \mapsto t^{*}]) \neq prenex(\hat{s}, t^{**})[x \mapsto t^{*}] \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{*}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,t^{*\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \notin FAV(t^{**}) \\
&:= (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u\,.\,(\mathsf{let}\, x : T := t^{*\prime}[\hat{s}' \mapsto \hat{s}'']\, \mathsf{in}\, t^{**})) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}[x \mapsto t^{*}]) \neq prenex(\hat{s}, t^{**})[x \mapsto t^{*}] \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{*}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u\,.\,t^{*\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \in FAV(t^{**})
\end{aligned}
$$

$$
\begin{aligned}
prenex(\hat{s}, \mathsf{if}\, t^{*}\, \{t^{**}\}\, \{t^{***}\}) &:= (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, (\mathsf{if}\, t^{*}\, \{t^{**\prime}\}\, \{t^{***\prime}[\hat{s}'' \mapsto \hat{s}']\})) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{***}) = (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u' \,.\, t^{***\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \notin (FAV(t^{*}) \cup FAV(t^{***})) \\
&:= (\hat{s}''', \forall(\hat{s}''' : \mathbb{N}) < u \,.\, (\mathsf{if}\, t^{*}\, \{t^{**\prime}[\hat{s}' \mapsto \hat{s}''']\}\, \{t^{***\prime}[\hat{s}'' \mapsto \hat{s}''']\})) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{***}) = (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u' \,.\, t^{***\prime}) \\
&\qquad \mathit{and} \quad \hat{s}' \in (FAV(t^{*}) \cup FAV(t^{***})) \\
&:= (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, \mathsf{if}\, t^{*}\, \{t^{**\prime}\}\, \{t^{***}\}) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{***}) = (\hat{s}, t^{***}) \\
&\qquad \mathit{and} \quad \hat{s}' \notin (FAV(t^{*}) \cup FAV(t^{***})) \\
&:= (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u \,.\, \mathsf{if}\, t^{*}\, \{t^{**\prime}[\hat{s}' \mapsto \hat{s}'']\}\, \{t^{***}\}) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{***}) = (\hat{s}, t^{***}) \\
&\qquad \mathit{and} \quad \hat{s}' \in (FAV(t^{*}) \cup FAV(t^{***})) \\
&:= (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, \mathsf{if}\, t^{*}\, \{t^{**}\}\, \{t^{***\prime}\}) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{***}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{***\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{**}) = (\hat{s}, t^{**}) \\
&\qquad \mathit{and} \quad \hat{s}' \notin (FAV(t^{*}) \cup FAV(t^{**})) \\
&:= (\hat{s}'', \forall(\hat{s}'' : \mathbb{N}) < u \,.\, \mathsf{if}\, t^{*}\, \{t^{**}\}\, \{t^{***\prime}[\hat{s}' \mapsto \hat{s}'']\}) \\
&\qquad \mathit{if} \quad prenex(\hat{s}, t^{***}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{***\prime}) \\
&\qquad \mathit{and} \quad prenex(\hat{s}, t^{**}) = (\hat{s}, t^{**}) \\
&\qquad \mathit{and} \quad \hat{s}' \in (FAV(t^{*}) \cup FAV(t^{**}))
\end{aligned}
$$

$$
prenex(\hat{s}, t^{*}\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t^{**}\}\{\mathsf{none}\, (B, [s]) \Rightarrow t^{***}\}) := \textit{analogous to definition of } prenex(\hat{s}, \mathsf{if}\, t^{*}\, \{t^{**}\}\, \{t^{***}\})
$$

$$
prenex(\hat{s}, t^{*}\, \mathsf{match}\{\mathsf{cons}\, x\, x\prime \Rightarrow t^{**}\}\{\mathsf{nil}\, (B, [s]) \Rightarrow t^{***}\}) := \textit{analogous to definition of } prenex(\hat{s}, \mathsf{if}\, t^{*}\, \{t^{**}\}\, \{t^{***}\})
$$

$$\begin{aligned}
prenex(\hat{s}, t^{*}\, t^{**}) &\coloneqq \textit{Definition by induction on structure of } t^{*}. \\
&\quad\ \textit{Cases of the form}\quad prenex(\hat{s}, \lambda x : (B, [s]).t^{*\prime}\, t^{**\prime}) \\
&\quad\ \textit{are analogous to definition of} \\
&\quad\ prenex(\hat{s}, \mathsf{let}\, x \,:\, T \coloneqq t^{*\prime} \,\mathsf{in}\, t^{**\prime}).
\end{aligned}$$

$$\begin{aligned}
prenex(\hat{s}, \mathsf{get}\, t^{*}.t^{**}) &\coloneqq (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, \mathsf{get}\, t^{*}.t^{**\prime}) \\
&\qquad \textit{if}\quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \textit{and}\quad \hat{s}' \notin FAV(t^{*}) \\
&\coloneqq (\hat{s}'', \forall(\hat{s}' : \mathbb{N}) < u \,.\, \mathsf{get}\, t^{*}.t^{**\prime}[\hat{s}' \mapsto \hat{s}'']) \\
&\qquad \textit{if}\quad prenex(\hat{s}, t^{**}) = (\hat{s}', \forall(\hat{s}'' : \mathbb{N}) < u \,.\, t^{**\prime}) \\
&\qquad \textit{and}\quad \hat{s}' \in FAV(t^{*})
\end{aligned}$$

$$\begin{aligned}
prenex(\hat{s}, \mathsf{remoteCall}\, t^{*}.t^{**}\, t^{***}) &\coloneqq \textit{Analogous to definition of}\quad prenex(\hat{s}, t^{**}\, t^{***}) \\
&\quad\ \textit{but prenexed quantified variable must not be} \\
&\quad\ \textit{contained in}\quad FAV(t^{*}).
\end{aligned}$$

$$\begin{aligned}
prenex(\hat{s}, \forall(\hat{s} : \mathbb{N}) \,.\, t^{*}) &\coloneqq (\hat{s}', \forall(\hat{s}' : \mathbb{N}) \,.\, \forall(\hat{s} : \mathbb{N}) \,.\, t^{*\prime}) \\
&\qquad \textit{if}\quad prenex(\hat{s}, t^{*}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{*\prime}) \\
&\qquad \textit{and}\quad \hat{s} \neq \hat{s}' \\
&\coloneqq (\hat{s}, \forall(\hat{s} : \mathbb{N}) \,.\, t^{*}) \\
&\qquad \textit{otherwise} \\
prenex(\hat{s}, \forall(\bar{s} : \mathbb{N}) \,.\, t^{*}) &\coloneqq (\hat{s}', \forall(\hat{s}' : \mathbb{N}) \,.\, \forall(\bar{s} : \mathbb{N}) \,.\, t^{*\prime}) \\
&\qquad \textit{if}\quad prenex(\hat{s}, t^{*}) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^{*\prime}) \\
&\coloneqq (\hat{s}, \forall(\bar{s} : \mathbb{N}) \,.\, t^{*}) \\
&\qquad \textit{otherwise} \\
prenex(\hat{s}, \forall(\hat{s} : \mathbb{N}) < u \,.\, t^{*}) &\coloneqq \textit{analogous to definition of}\quad prenex(\hat{s}, \forall(\hat{s} : \mathbb{N}) \,.\, t^{*})
\end{aligned}$$

$$prenex(\hat{s}, (\langle t^{*} \rangle_{\mathcal{I}}, [h])) \coloneqq (\hat{s}, (\langle t^{*} \rangle_{\mathcal{I}}, [h]))$$

We see that the defined *prenex*-functions aligns with the examples presented earlier.

Later we assign types of the form $(B,\ s,\ l)$ to terms. The basic type B can also have the form $\forall(s : \mathbb{N}) < u \,.\, B^{*}$. In the following we define a function *btq* which for any such term returns the quantified variable s.

Definition 22. *Let ε denote a fixed element with $\varepsilon \notin \mathbb{X}$. The function $btq : deriv(t) \to \mathbb{X} \cup \{\varepsilon\}$ is defined by induction over the term structure.*

Analogous to the definition of prenex

Definition 23 (Syntactic Bound Elimination)**.** *The function $BE : deriv(t) \to deriv(t)$ is defined by induction over the term structure.*

$$BE(t) := \left\{ \begin{array}{ll} \forall(\hat{s}' : \mathbb{N}) \,.\, t^* & \textit{if } btq(t) = \hat{s} \in \mathbb{X} \textit{ and} \\ & \quad prenex(\hat{s}, t) = (\hat{s}', \forall(\hat{s}' : \mathbb{N}) < u \,.\, t^*) \\ t & \textit{otherwise} \end{array} \right\}$$

2.5 Static Semantics

In this section we present the type system for λ^{lat} and $\lambda^{\mathrm{lat}'}$. We therefore define two typing relations: $\vdash$ and $\Vdash$. We use the first one to assign types to placed terms and to extract static bounds on the runtime latency their reduction entails. Meanwhile, we use the second relation to check whether placement terms are well-typed without assigning any type to them. However, for this we need the types and latency bounds inferred by $\vdash$.

We start by defining the typing context we need to consider when we assign types to placed terms and extract latency bounds in Section 2.5.1. In particular, we see that the location of placed variables becomes part of their type.

The extraction of latency bounds involves type-level computations which we specify by arithmetic terms. Hence, Section 2.5.2 gives an overview over the arithmetic we use to reason about type-level computations.

In the Sections 2.5.3 to 2.5.5 we present our typing rules for standard values like true, type constructors like $\mathsf{List}(\cdot, [\cdot])$ and variable access (local and remote).

In general, programs can contain multiple possible paths from which only one is taken during runtime. Consider the branching term $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$. Depending on the condition t_c, one of the branches t_t, t_f is considered during reduction. This decision determines the term's runtime latency. In Section 2.5.6 we present our approach to abstract over branches and extract latency bounds.

In Sections 2.5.7 we explain function types and present how our type system can extract respective latency-bounds that depend on the input size. Furthermore, in this section we explain how our type system ensures that all accepted recursive functions terminate. With this guarantee, our language becomes total. This property is also necessary to extract latency bounds from recursive function applications.

The sections mentioned so far, defined our typing relation $\vdash$ for placed terms in λ^{lat}. In Section 2.5.8 we extend $\vdash$ to $\lambda^{\text{lat}'}$. Afterwards we define in Section 2.5.9 the typing relation $\Vdash$ for placement terms.

Type Annotations λ^{lat} is an extension of the typed λ-calculus and so is our type system. In the typed λ-calculus' standard version every term is assigned a type that describes the kind of value it reduces to. Illformed terms are not assigned any type. We call terms that are assigned a type *well-typed*. For instance, a function application application $(\lambda x : B.t')\, t$ is only well-typed if the argument t matches the type B expected by the function. That is, if t's type is different from B, the application is rejected. As a consequence, every well-typed term provably reduces to a value.

We adopt the same approach to prevent stuck reductions. Consider the term $\mathsf{if}\, v_c\, \{t_t\}\, \{t_f\}$ for a value v_c. In Section 2.4.3.1 we saw that we can only reduce it further if v_c is either true or false. We group those two values into the type $\mathsf{Boolean}$. Then we know that independent of the concrete value v_c, as long as it is of type $\mathsf{Boolean}$, we can take one reduction step on the if-term. We extend this approach to more complex terms and assign any term t_c the type $\mathsf{Boolean}$ if we can prove that it eventually reduces to a $\mathsf{Boolean}$-value. This way, we can also prove that the term $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$ eventually reduces to one of the two branches t_t, t_f.

The goal of λ^{lat} is to offer a type system to statically track the latency in distributed programs. Therefore, we extend the types by annotations telling about the latency that a term of this type causes during its reduction. Consider a term t of type $\mathsf{Boolean}$ with latency annotation l. This tells us that we can reduce t to a $\mathsf{Boolean}$ value and that this reduction can involve latency up to l. In general, terms can have multiple reduction sequences leading to different runtime-latencies. In Section 2.5.6 we investigate this problem and present an approach to extract upper bounds for a term's runtime-latency.

We also saw in Section 2.4.3.3 that unrestricted recursion allows to formulate diverging functions causing a diverging runtime-latency during reduction. We use sized types [1] to restrict well-typed terms to terminating recursion. Hence, we additionally annotate our types by sizes.

In λ^{lat}, we annotate types by size indices and latency bounds. A fully annotated type has the form $(B, [s], [l])$ where B is a basic type and s, l are arithmetic terms. We assign such a type to any term that provably reduces to a value of type B and size s with a runtime-latency below l.

Arithmetic term representations are not unique. Consider a boolean list of with two elements that we constructed without any remote communication. As we later see, we can assign it the type $(\mathsf{List}(\mathsf{Boolean}, [0]), [S\,S\,0], [0])$. Using a match-term we can deconstruct it into a head and a tail. The tail contains one element less and hence has the size $S\,S\,0 - S\,0$. By prepending the head again we increase its size by one, obtaining $(S\,S\,0 - S\,0) + S\,0$. That means, though the procedure did not alter our list we transformed its type from $(\mathsf{List}(\mathsf{Boolean}, [0]), [S\,S\,0], [0])$ to $(\mathsf{List}(\mathsf{Boolean}, [0]), [(S\,S\,0 - S\,0) + S\,0], [0])$. Assigning multiple types to a single term complicates a type system significantly. In particular, differentiating between different representations for the same size depending on a value's construction history would render our type system unusable. We hence regard arithmetic terms as representatives of the value they describe. We make this explicit in our annotations by wrapping arithmetic terms h in brackets $[\cdot]$. Later in Section 2.5.2 we see our type system treats an annotation $[h]$ as a representative of an equivalence class. This justifies the notation $[\cdot]$.

Throughout this section we assume a fixed program context $(\mathbb{X}, \mathbb{I}, \mathbb{P}, \mathcal{P}, \mathcal{T}, \mathcal{L})$ according to definition 4.

2.5.1 Typing Context

Whenever we assign a type T to some term t, we denote it by $... \vdash t : T$. On the left side of the typing relation $\vdash$ we specify the so-called *typing context*. This is a collection of information needed apart from the static program context can influence the typing of t.

Local Variables A term's type specifies the kind of value the term reduces to and the latency this reduction involves. Determining both is immediate for values like true and terms with an unambiguous reduction sequence like $\mathsf{if}\,\mathsf{true}\,\{\mathsf{unit}\}\,\{\mathsf{unit}\}$. It gets, however, more complicated when variables are involved. For instance, replace unit for a variable x in the previous example. We cannot determine which kind of value the term $\mathsf{if}\,\mathsf{true}\,\{x\}\,\{x\}$ reduces to without knowing the kind of value bound to x. To make the necessary information available during the term's typing we introduce a local typing environment Γ as part of the typing context. Thus our typing relation obtains the form $...; \Gamma; ... \vdash t : T$.

Definition 24 (Local Variable Typing Environment)**.** *Let x, B, h denote the* *corresponding definitions of variables, basic types and arithmetic terms presented in Figure 2.1. A* local variable typing environment Γ *is syntactically defined as follows:*

$$\Gamma \;:=\; \varnothing \;\mid\; \Gamma, x \mapsto (B, [h]) \;\mid\; \Gamma, x \mapsto \mathbb{N}$$

According to this definition, a local variable typing environment Γ can contain multiple bindings for the same variable. This can occur when we type a term t that lies in a nested scope

$$\begin{array}{l} \mathsf{let}\ x : T := t'\ \mathsf{in} \\ \quad \mathsf{let}\ x : T' := t''\ \mathsf{in} \\ \quad\quad t \end{array}$$

In such a case only the last declaration of x should be considered. The next definition makes this precise.

Definition 25 (Local Variable Typing Lookup)**.** *Let Γ be a local variable typing environment. We define the lookup $\Gamma(x)$ for a variable x in Γ as a partial function by induction over the structure of Γ.*

In the following let x, x' be variables with $x \neq x'$, B a basic type, s a size and Γ' a local variable typing environment.

$$\begin{array}{rcl} (\Gamma', x \mapsto (B, [s]))(x) & := & (B, [s]) \\ (\Gamma', x' \mapsto (B, [s]))(x) & := & \Gamma'(x) \\ (\Gamma', x \mapsto \mathbb{N})(x) & := & \mathbb{N} \\ (\Gamma', x' \mapsto \mathbb{N})(x) & := & \Gamma'(x) \end{array}$$

Locations and Placed Variables In λ^{lat} every computation is placed on a specific location. This becomes relevant as soon as we access data or communicate with remote entities.

Consider accessing a variable x placed on a remote peer via the term $\mathsf{get}\, p'.x$. Our dynamic semantics presented in Section 2.4 use peer instances to represent runtime-locations. Considering concrete locations in our type system is not feasible. Hence, we approximate runtime-locations by peer types.

The type we $(B, [s], [l])$ we assign to the term $\mathsf{get}\, p'.x$ determines the kind and size of value bound to x on p' and also the latency this remote request entails. The latency l depends on the connection between the peer P' that p' belongs to as well as on the P the request is send from.

Both information have to be stored in the typing context. Hence, the peer P this computation is placed becomes part of the context. Furthermore, we add a typing environment for placed variables Δ to it. Similiar to the environment for local variables, Δ stores the basic type and size of every declared variable. However, it also contains the peer it is placed on. The types we assign to placed variables on a peer P have the form $(B, [s])\,\mathsf{on}\,P$. Types of this form are called *placement types*.

Thereby, our typing relation obtains the form $\Delta; \Gamma; ...; P \vdash t : T$. With these information we know which remote peer P' a variable x is placed on as well as which peer the request $\mathsf{get}\,p'.x$ is sent from. Hence, we can estimate the latency the request entails.

Definition 26 (Placed Variable Typing Environment)**.** *Let x, B, h, P denote the corresponding definitions of variables, basic types, arithmetic terms and peer types presented in Figure 2.1. A* placed variable typing environment Δ *is syntactically defined as follows:*

$$\Delta \ := \ \varnothing \ \mid \ \Delta, x \mapsto (B, [h])\,\mathsf{on}\,P$$

Definition 27 (Placed Variable Typing Lookup)**.** *Let Δ be a placed variable typing environment. We define the lookup $\Delta(x)$ for a variable x in Δ as a partial function by induction over the structure of Δ.*

In the following let x, x' be variables with $x \neq x'$, B a basic type and s a size. Furthermore, let P be a peer type and Δ' a local variable typing environment.

$$\begin{aligned} (\Delta', x \mapsto (B, [s])\,\mathsf{on}\,P)(x) \ &:= \ (B, [s])\,\mathsf{on}\,P \\ (\Delta', x' \mapsto (B, [s])\,\mathsf{on}\,P)(x) \ &:= \ \Delta'(x) \end{aligned}$$

Arithmetic Assumptions Consider the $\lambda x : (B, [s]).t$. It expects an argument of basic type B and size s. Whenever we apply it to an argument a of basic type B_a and size s_a, our type system checks whether B_a matches B and s_a matches s. In case the a does not meet the expectations of f, our type system rejects the application.

λ^{lat} also allows us to specify a range of acceptable sizes as in $f = \forall(s : \mathbb{N}) < u \,.\, \lambda x : (B, [s]).t$. Here, the type system accepts $f\,a$ as long as we can proof that $s_a < u$.

However, when we assign a type to the function body t independent of any application, we do not know the eventual argument size. Instead we have to consider any size for x below the upper bound u.

Now consider the case that we want to assign a type to f independent of an application. Then we assume that x is bound to some value of basic type B and size s. We can express this by adding the binding $x \mapsto (B, [s])$ to our local typing environment Γ. However, we also have to express the constraint

$s < u$ in our context. Therefore, we introduce a set of arithmetic assumptions Λ in which we can include this constraint. In the next section we describe an arithmetic theory which we use to reason about arithmetic terms. Λ can contain any proposition expressible in this theory, e.g., $s < u$ and $s_1 = s_2$. Thereby, our typing relation finally obtains the form $\Delta; \Gamma; \Lambda; P \vdash t : T$.

2.5.2 Reasoning About Sizes and Latency

We use arithmetic terms to describe sizes and latency bounds in our type system. Consider modifying a list li of type $(\mathsf{List}(\mathsf{Boolean}, [0]), [s], [l])$. Here, s and l are arithmetic terms. s decribes the exact number of elements li contains. l is an upper bound for the runtime-latency the reduction of li causes. As we see in Section 2.5.4 prepending an element by a term $\mathsf{cons}\,\mathsf{true}\,li$ increases the size to $s+1$. Suppose, we remove the newly attached head again. We thereby decrease the size by 1 to $s+1-1$ and obtain a list $\hat{li}$ that is structurally equal to our original list li. The result of this computation has the type $(\mathsf{List}(\mathsf{Boolean}, [0]), [s+1-1], [l])$.

Type-Level Computations The arithmetic terms l, s and $s+1-1$ describe arithmetic *type-level* computations. Intuitively s and $s+1-1$ should yield the same result. However, to keep our type system simple it does not execute type-level computations. Instead we merely use the arithmetic terms as representatives.

As we see in the following sections our type system uses size and latency annotations to give static guarantees. Consider a well-typed function $f = \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [s]).t$. The argument li has the basic type $\mathsf{List}(\mathsf{Boolean}, [0])$ and size s expected by f. Using this information, our type system can prove that the application successfully reduces to a value without getting stuck.

Since li and $\hat{li}$ are structurally equal, the same should hold for the application $f\,\hat{li}$. However, $\hat{li}$ has the type $(\mathsf{List}(\mathsf{Boolean}, [0]), [s+1-1], [l])$. The arithmetic term $s+1-1$ does not match the arithmetic term s describing the expectations of f.

Proving Arithmetic Properties Differentiating between different arithmetic terms that describe the same size or latency would render our type-system quite useless. Hence, we need a way to reason about arithmetic terms and the values they represent. In the following sections we present typing rules that need to know how different arithmetic terms relate to each other.

In particular, we they should be able to determine that $s = s + 1 - 1$ and that $s < s + 1$.

Hence, we extend our type system by an arithmetic theory that allows us to prove above and similar properties. We could use *Peano Arithmetic* [11] extended by defining axioms for our choice of a latency context $\mathcal{L}$. Note that the defining axioms for the function $\mathcal{L}$ depend on the concrete weights we put on connections between peers. Since we assume a given program context $(\mathbb{X}, \mathbb{I}, \mathbb{P}, \mathcal{P}, \mathcal{T}, \mathcal{L})$ throughout Section 2.5, we can assume that all axioms regarding $\mathcal{L}$ are fixed.

However, we want to advocate for constructive reasoning. Hence, we choose a fragment of *Heyting Arithmetic* [11] extended by axioms for $\mathcal{L}$ instead. Heyting arithmeric is the intuitionistic counterpart of Peano Arithmetic. Intuitionistic logic and therefore also Heyting Arithmetic does not contain the *law of the excluded midde* (i.e., $a \vee a$ for arbitrary popositions a). As a consequence, a proof for a disjunction $a \vee b$ not only prove that at least one of a and b holds but also which one. Similarly, the proof an existential proposition $\exists x.c$ constructs a whitness w for x and proves that $c[x \mapsto w]$ holds. In general, this approach is called *constructive reasoning.*

Some properties provably cannot be proven without the law of the excluded middle (for instance the law itself). However, in Heyting Arithmetic we can prove a specialised version for quantifier-free proposition. That is, for any quantifier-free formula A_0, we can prove $A_0 \vee \neg A_0$. This enough to prove all arithmetic properties relevant for our type system.

Hence, we chose the smallest fragment of Heyting Arithmetic that defining axioms for the functions $S, +, \dot{-}, \cdot$ and the relations $=_0$, $<_0$ for type 0 (i.e., natural number) equality and strictly less-than.

Throughout the rest of Section 2.5 we see that the type system only relies on the existence of proofs but never refers to the proofs' internals. Therefore, our type system is actually modular regarding the concrete arithmetic theory it uses. The concrete choice, however, determines the concrete programs we can type. In Section 2.5.7 we present functions whose result type depend on the input's size. In order to type such a function we need to find an arithmetic term that expresses the output's size in regard to the input's size, e.g. input size s and output size $2 \cdot s$. Similarly, we need to find an arithmetic term that expresses a bound on the runtime latency which a function application causes, in respect to the input's size. We chose our arithmetic language and theory such that it is expressive enough to type all examples we present. Extending the arithmetic language and theory might allow to type more concrete programs (depending on the concrete extension) but does not change anything fundamental about the presented type system.

Consider a function *binSub* that expects a list x of arbitrary size and returns a list containing all pairs of elements of x. With our current set of arithmetic terms and the above mentioned fragment of Heyting Arithmetic we cannot type *binSub*. However, extending our arithmetic language and theory to contain symbols for division and factorial as well as defining axioms allows us express the result's size. Hence, we could type *binSub*.

Proof Relation Following the standard convention we name our proof relation $\vdash$. Let ϕ and Λ be a single formula and a set of formulas in the language of Heyting Arithmetic, respectively. Then $\Lambda \vdash \phi$ denotes that there is a proof for ϕ in our fragment of Heyting Arithmetic which might use the formulas from Λ as additional axioms.

Notations In Heyting Arithmetic, natural numbers can be constructed using the constant 0 and the successor function S. Hence, the standard way to construct a natural number n is to apply the successor function n times to 0. That is, $\underbrace{S \dots S}_{n \text{ times}} 0$. In the following we denote this application by the literal n to simplify the notation. In particular 1 is short for $S\,0$.

The typing rules for branching terms like $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$ need to compute the maximum of the branches' type-level latency bounds l_t, l_f. The maximum function is not part of arithmetic terms as presented in Figure 2.1. However, it can be easily defined in Heyting Arithmetic. Hence, we use $\max\big(l_t, l_f\big)$ as an abbreviation.

Furthermore, we denote the set of propositions we can formulate in Heyting Arithmetic by $\mathcal{P}_{arith}$.

Type Equality We can prove that different arithmetic terms like s and $s + 1 - 1$ are equal. As mentioned above, it is undesirable to differentiate between different representations of the same size or latency. Hence, in our type system we view size and latency annotations h as representatives of the equavilence class containing all arithmetic terms provably equal to h. We make this explicit in our syntax by the notation $[h]$.

As a consequence, consider types equal that only differ in the representatition of involved arithmetic terms. To make this precise we define equality relations $\approx$ and $\hat{\approx}$ for basic and fully annotated types, respctively.

Definition 28 (Basic Type Equality)**.** *Let B denote the corresponding syntactic definition of basic types presented in Figure 2.1. We define a relation*

$$\frac{}{\Lambda \Vdash B \approx B} \quad \text{(BEq-SynEq)}$$

$$\frac{\Lambda \Vdash B \approx B' \qquad \Lambda \vdash s =_0 s'}{\Lambda \Vdash \mathsf{Option}\,(B,[s]) \approx \mathsf{Option}\,(B',[s'])} \quad \text{(BEq-Option)}$$

$$\frac{\Lambda \Vdash B \approx B' \qquad \Lambda \vdash s =_0 s'}{\Lambda \Vdash \mathsf{List}(B,[s]) \approx \mathsf{List}(B',[s'])} \quad \text{(BEq-List)}$$

$$\frac{\Lambda \Vdash B_l \approx B_r \qquad \Lambda \vdash s_l =_0 s_r \qquad \Lambda \vdash B'_l =_0 B'_r \qquad \Lambda \vdash s'_l =_0 s'_r \qquad \Lambda \vdash l'_l =_0 l'_r}{\Lambda \Vdash (B_l,[s_l]) \to (B'_l,[s'_l],[l'_l]) \approx (B_r,[s_r]) \to (B'_r,[s'_r],[l'_r])} \quad \text{(BEq-Arrow)}$$

$$\frac{\Lambda \Vdash B \approx (B'[s' \mapsto s])}{\Lambda \Vdash \forall (s:\mathbb{N})\,.\,B \approx \forall (s':\mathbb{N})\,.\,B'} \quad \text{(BEq-ForAll)}$$

$$\frac{\Lambda \vdash u =_0 u' \qquad \Lambda \Vdash B \approx (B'[u' \mapsto u][s' \mapsto s])}{\Lambda \Vdash \forall (s:\mathbb{N}) < u\,.\,B \approx \forall (s':\mathbb{N}) < u'\,.\,B'} \quad \text{(BEq-ForAllBounded)}$$

Figure 2.6: Defining Rules for Basic Type Equality

$\cdot \Vdash \cdot \approx \cdot \subseteq \mathcal{P}_{arith} \times deriv(B) \times deriv(B)$ *between a set of arithmetic assumptions and two basic types by the following rules presented in Figure 2.6.*

By induction over the structure of basic types we can proof that for a fixed set of arithmetic assumptions Λ the above relation indeed is an equality relation.

Definition 29 (Type Equality)**.** *Let T denote the corresponding syntactic definition of fully annotated types presented in Figure 2.1. We define a relation $\cdot \Vdash \cdot \hat{\approx} \cdot \subseteq deriv(T) \times deriv(T)$ between a set of arithmetic assumptions and two fully annotated types.*
Let B, B' be basic types, s, s' sizes and l, l' be latencies. Then

$$\Lambda \Vdash (B,s,l)\,\hat{\approx}\,(B',s',l') \quad \textit{iff} \quad \Lambda \Vdash B \approx B',\ \Lambda \vdash s =_0 s',\ \Lambda \vdash l =_0 l'.$$

Consider a fixed set of arithmetic assumptions Λ. Then basic type equality $\Lambda \Vdash \cdot \approx \cdot$ is an equivalence relation. The same holds for provable type 0 equality $\Lambda \vdash \cdot =_0 \cdot$. Hence, $\Lambda \Vdash \cdot \hat{\approx} \cdot$ is an equivalence relation, too.

The typing rules we present in the following sections consider different representations h_1, h_2 of natural numbers and require equality proofs $\Lambda \vdash$

$$\frac{}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{unit} : (\mathsf{Unit}, [0], [0])} \quad \text{(T-Unit)}$$

$$\frac{}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{true} : (\mathsf{Boolean}, [0], [0])} \text{(T-True)} \qquad \frac{}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{true} : (\mathsf{Boolean}, [0], [0])} \text{(T-True)}$$

$$\frac{\mathcal{P}(p') = P'}{\Delta; \Gamma; \Lambda; P \vdash p' : (P', [0], [0])} \quad \text{(T-PeerInst)}$$

Figure 2.7: Typing Rules for Standard Values

$h_1 =_0 h_2$ were necessary. The same holds for basic and fully annotated types. However, in order to simplify the typing rules we make an exception for the size 0. The types Unit, Boolean, Option and function types always have size 0. As we present later, let-terms $\mathsf{let}\, x : (B, [s], [l]) := t \,\mathsf{in}\, t'$ allow us to chose a specific representation for the basic type, size and latency of t. Even for Unit which always has size 0, this can lead to verbose size terms like $s - s$. Hence, instead of writing $(\mathsf{Unit}, [s_0], [l])$ and requesting a proof $\Lambda \vdash s_0 =_0 0$, we simply write $(\mathsf{Unit}, [0], [l])$. We do the same for the types Boolean, Option and for all function types.

2.5.3 Embedding Types for Standard Values

Figure 2.7 presents the typing rules for the values unit, true, false and peer instances p. We follow the standard conventions for the values unit, true, false and assign them the basic types Unit and Boolean, respectively. According to our reduction rules presented in Section 2.4, we cannot reduce values any further. Hence, they cannot cause any latency and we annotate them with a latency of 0. They are also no composites, so we annotate their size with 0, too. Accordingly, the typing rule T-Unit assigns the type $(\mathsf{Unit}, [0], [0])$ to unit. Similiarly, the rules T-True and T-False assign the type $(\mathsf{Boolean}, [0], [0])$ to true and false, respectively. Note that these rules do not use the given typing context $\Delta, \Gamma, \Lambda, P$ in any way. That is, these typings hold in any context.

Our program context contains a set of peer instances $\mathbb{I}$, a set of peer types $\mathbb{P}$ and a peer typing $\mathcal{P}$ mapping instances to their respective type. Typing rule T-PeerInst embeds the peer typing into our annotated type system. Independent of the typing context, this rule assigns any peer instance $p \in \mathbb{I}$ the type $(\mathcal{P}(p), [0], [0])$.

$$\frac{\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l])}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{some}\, t : (\mathsf{Option}\,(B, [s]), [0], [l])} \quad \text{(T-Some)}$$

$$\frac{}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{none}\,(B, [s]) : (\mathsf{Option}\,(B, [s]), [0], [0])} \quad \text{(T-None)}$$

$$\frac{\begin{array}{cc} \Lambda \Vdash B_h \approx B_t & \Lambda \vdash s_{Bh} =_0 s_{Bt} \\ \Delta; \Gamma; \Lambda; P \vdash t_h : (B_h, [s_{Bh}], [l_h]) & \Delta; \Gamma; \Lambda; P \vdash t_t : (\mathsf{List}(B_t, [s_{Bt}]), [s_t], [l_t]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{cons}\, t_h\, t_t : (\mathsf{List}(B_t, [s_{Bt}]), [s_t + 1], [l_h + l_t])} \quad \text{(T-Cons)}$$

$$\frac{}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{nil}\,(B, [s]) : (\mathsf{List}(B, [s]), [0], [0])} \quad \text{(T-Nil)}$$

Figure 2.8: Typing Rules for Basic Type Constructors Option and List

2.5.4 Basic Type Constructors

As presented in Figure 2.1, λ^{lat} contains five basic type constructors: (i) $\mathsf{Option}\,(\cdot, [\cdot])$, (ii) $\mathsf{List}(\cdot, [\cdot])$, (iii) the arrow $\rightarrow$ to build function types, (iv) universal quantification $\forall(\cdot : \mathbb{N})\,.\,\cdot$ to abstract over sizes in types and (v) its restricted version $\forall(\cdot : \mathbb{N}) < \cdot\,.\,\cdot$. This section focuses only on numbers (i) List and (ii) Option. Due to their complexity and relation, we explain (iii) $\rightarrow$ as well as the quantifiers (iv) and (v) separately in Section 2.5.7.

λ^{lat} also contains term constructors $\mathsf{some}\,\cdot$, $\mathsf{none}\,(\cdot, [\cdot])$ and $\mathsf{cons}\,\cdot\,\cdot$, $\mathsf{nil}\,(\cdot, [\cdot])$ to instantiate the composite types Option and List, respectively. Figure 2.8 presents the corresponding typing rules.

Sized Lists For any basic type B and size s_{elem}, the type-constructor $\mathsf{List}(\cdot, [\cdot])$ allows us to construct the basic type $\mathsf{List}(B, [s_{elem}])$. It represents lists containing elements of basic type B and size s_{elem}. For instance, $\mathsf{List}(\mathsf{Boolean}, [0])$ is the basic type of lists containing booleans. Similiarly, basic type $\mathsf{List}(\mathsf{List}(\mathsf{Boolean}, [0]), [2])$ represents lists of boolean pairs.

We can construct an empty list using the nil-constructor. However, we want to assign any term in λ^{lat} at most one type. That is, we need to differentiate between empty lists of type $\mathsf{List}(\mathsf{Boolean}, [0])$ and such of type $\mathsf{List}(\mathsf{List}(\mathsf{Boolean}, [0]), [2])$. The constructor hence requires us to supply the element type B and size s_{elem}. The term $\mathsf{nil}\,(B, [s_{elem}])$ is already a value and therefore causes no remote communication. Thus, typing rule T-Nil assigns the term $\mathsf{nil}\,(B, [s_{elem}])$ type $(\mathsf{List}(B, [s_{elem}]), [0], [0])$.

Following the standard convention the cons-constructor allows us to form lists by appending a head t_h to an already available list t_t. Consider the term $\mathsf{cons}\, t_h\, t_t$ and let $(B_h, [s_{Bh}], [l_h])$ and $(\mathsf{List}(B_t, [s_{Bt}]), [s_t], [l_t])$ be the types of the head t_h and the tail t_t. We can type the term $\mathsf{cons}\, t_h\, t_t$ by applying typing rule T-Cons. The rule contains the premises $\Lambda \Vdash B_h \approx B_t$ and $\Lambda \vdash s_{Bh} =_0 s_{Bt}$. These require the head's basic type B_h and tail's element type B_t as well as their respective sizes s_{Bh} and s_{Bt} to be provably equivalent. Thereby, we ensure that prepending head t_h to the list t_t does result in a list containing elements of basic type B_t and size s_{Bt}, again. Here, we also see that our typing rule treats arithmetic terms as representatives of an equivalence.

The type of our list t_t contains a size annotation s_t describing the number of elements the list contains. Prepending t_h creates a new list with an additional element. Hence, we also increase its size annotation by 1.

Tracking the list's size in its type allows us to extract better latency approximations. Consider a list processing function f and suppose our type system extracts that the processing of each list element causes some latency l. Let a be a list of size s and consider the application $f\, a$. By knowing the list's size we can extract the overall latency $l \cdot s$.

According to definition 9, the term $\mathsf{cons}\, t_h\, t_t$ is a value iff both t_h and t_t are values. That is, reducing $\mathsf{cons}\, t_h\, t_t$ to a value means we have to reduce both subterms to values. The latency annotation l_h and l_t in their respective types describe the latency a reduction of t_h and t_t entails, respectively. Hence, the sum $l_h + l_t$ captures the latency the reduction of $\mathsf{cons}\, t_h\, t_t$ causes.

Therefore typing rule T-Cons assigns the type $(\mathsf{List}(B_t, [s_{Bt}]), [s_t{+}1], [l_h{+}l_t])$ to the term $\mathsf{cons}\, t_h\, t_t$. Note that we continue to use the tail's element type B_t and size s_{Bt}. Since the B_t and B_h as well as s_{Bt} and s_{Bh} are provably equivalent, it does not matter which representation we use.

Options can be instantiated through the constructors $\mathsf{some}\cdot$ and $\mathsf{none}\,(\cdot, [\cdot])$ as well as through remote calls (get, $\mathsf{remoteCall}$, $\mathsf{eval} \cdot \mathsf{on} \cdot$). We can use the typing rules T-Some nad T-None to type some- and none-terms, respectively. T-None is completely analogous to T-Nil. Rule T-Some is very similiar to T-Cons but with a slight difference. The result type's size annotation does not depend on the wrapped term but is always 0. That is, for a term t of type $(B, [s], [l])$, rule T-Some assigns the type $(\mathsf{Option}\,(B, [s]), [0], [l])$ to the term $\mathsf{some}\, t$.

In contrast, a cons-aplication increases a list's size. As explained above, this allows us to calculate latencies that depend on the number of elements a list contains. This works because we can only construct lists in a deterministic manner through the constructors $\mathsf{cons} \cdot \cdot$ and $\mathsf{nil}\,(\cdot, [\cdot])$. Options, however, can also be instantiated non-deterministically through remote communication. Consider the term $\mathsf{get}\, p.x$ and suppose it requests a boolean value v bound to x. As presented in Section 2.4 retrieving the value can succeed or fail. Therefore, the term reduces non-deterministically either to $\mathsf{some}\, v$ or to $\mathsf{none}\,(B, [s])$. Suppose a some-application increased the term's size similar to cons. Then $\mathsf{some}\, t$ had type $(\mathsf{Option}\,(\mathsf{Boolean}, [0]), [1], [l])$ and $\mathsf{none}\,(\mathsf{Boolean}, [0])$ had type $(\mathsf{Option}\,(\mathsf{Boolean}, [0]), [0], [l])$. Hence, it is not clear which type we should assign to $\mathsf{get}\, p.x$.

In our type system every well-typed term has a unique type and size annotations describe a value's exact size rather than an upper bound. Annotating the type of $\mathsf{some}\, v$ with a size different from 0 means we have to do the same for the term $\mathsf{get}\, p.x$. Thereby, we treat size annotations as upper bounds rather than exact values. Our approach, however, allows us to extract better latency approximations.

2.5.5 Variable Access

There are two kinds of variables in λ^{lat}: (i) Local and (ii) placed variables. They have different scopes and require different access mechanisms. Therefore, we store their respective type information in different typing environments Γ, Δ. Figure 2.9 presents the respective typing rules.

Local Variables x can be introduced by function definitions $\lambda x : (B, [s]).t_s$, let-terms $\mathsf{let}\, x : (B, [s], [l]) := t\, \mathsf{in}\, t_s$ and match-terms. In each case they are bound to a value of specific type B and size s within a restricted scope t_s. This scope lives on the same location as the value bound to x. Hence, within this scope we can access x without any remote communication. Our reduction rules presented in Section 2.4 ensure that whenever we bind the result of a computation t to x, we reduce t to a value before we start to reduce the variable's scope t_s. That is, by the time we encounter a reference to x in t_s, the value bound to x is already available and referencing it does not trigger a recomputation. Hence, the variable access does not introduce any latency. For this reason, our typing envoronment for local variables Γ stores mappings of the form $x \mapsto (B, [s])$ not including any latency information.

The typing rule we use for local variables x is T-LocalVar. It refers to the local variable typing environment Γ and assigns a variable x the lookup result $\Gamma(x)$ augmented by the latency annotation 0.

$$\frac{\Gamma(x) = (B, [s])}{\Delta; \Gamma; \Lambda; P \vdash x : (B, [s], [0])} \quad \text{(T-LocalVar)}$$

$$\frac{\Delta(x) = (B, [s]) \,\mathsf{on}\, P}{\Delta; \Gamma; \Lambda; P \vdash x : (B, [s], [0])} \quad \text{(T-LocallyPlacedVar)}$$

(a) Variables

$$\frac{\begin{array}{c} P \not\equiv P' \qquad P \leftrightarrow P' \\ \Delta; \Gamma; \Lambda; P \vdash p' : (P', [0], [l_{p'}]) \qquad \Delta; \varnothing; \varnothing; P' \vdash t : (B, [s], [l_t]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{get}\, p'.t : (\mathsf{Option}\,(B, [s]), [0], [l_{p'} + \mathcal{L}(P, P') + l_t + \mathcal{L}(P', P)])} \quad \text{(T-Get)}$$

$$\frac{\begin{array}{c} \Delta; \Gamma; \Lambda; P \vdash p' : (P', [0], [l_{p'}]) \qquad P \not\equiv P' \qquad P \leftrightarrow P' \\ \Delta; \varnothing; \varnothing; P' \vdash t : (B, [s], [l_t]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \mathsf{eval}\, t\, \mathsf{on}\, p' : (\mathsf{Option}\,(B, [s]), [0], [l_{p'} + \mathcal{L}(P, P') + l_t + \mathcal{L}(P', P)])} \quad \text{(T-PlacedEval)}$$

(b) Remote Access

Figure 2.9: Typing Rules for Variable Access

Placed Variables We use placement terms $\mathsf{place}\, x : (B, [s], [l]) := t\, \mathsf{on}\, P\, \mathsf{in}\, q$ to place a computation t on all instances of peer type P. Afterwards we make the result accessible through variable x in the scope q. This scope can contain computations placed on P as well as on other peers P'. All P-instances store the variable x as well as its bound value. Accessing x within a computation placed on P does hence not necessitate any remote communication but is analogous to a local variable access. Meanwhile, accessing x from a remote P'-instance means that we have to request the value bound to x and await a response. Therefore, we have to keep track of the location x is placed on as well as the location referencing it. We can use this information to differentiate between local and remote accesses to placed variables as well as computing the latency a remote access entails.

Locally Placed Variables can be accessed like local variables without any remote communication. The corresponding typing rule T-LocallyPlacedVar is almost analagous to T-LocalVar. The difference, however, is that we use a different typing environment for placed variables than for local ones. Consider a placed variable x and suppose our typing environment for placed variables Δ contains a mapping $x \mapsto (B, [s])\,\mathsf{on}\,P$. The typing rule T-LocallyPlacedVar refers to the lookup result of x in Δ and assigns the type $(B, [s], [0])$ to x.

This typing is, however, only correct for locally placed variables which can be accessed without any remote communication. Hence, typing rule T-LocallyPlacedVar requires that the lookup result for x has the form $\ldots\,\mathsf{on}\,P$. Thereby, it ensures that x is indeed placed on our current peer P.

Remotely Placed Variables can only be accessed by requesting the value explicitely from a remote peer instance p'. λ^{lat} offers three terms to access remotely placed variables: (i) get-terms to request remote values, (ii) remoteCall-terms to invoke functions placed on remote peers and (iii) terms of the form $\mathsf{eval}\,t\,\mathsf{on}\,p$ to place a computation t on peer instance p and request its result.

Since remote function applications build on local function applications, we describe them together in Section 2.5.7. For now, we only explain (i) and (iii).

Remote Value Requests have the form $\mathsf{get}\,p'.t$ where p' is a peer instance belonging to some remote peer P' and t is a computation placed on P'.

According to the reduction rules presented in Section 2.4, the get-term's reduction involves (i) reducing p' on our current peer P and (ii) reducing t on the remote peer instance p' and hence on remote peer P'. Therefore, we have to type p' on P and t on P'. The computation t should only have access to data available to P. For this reason we type is under the context $\Delta; \varnothing; \varnothing; P$. Note that get-terms are actually used to access variables placed on remote peers. However, our reduction rules reduce terms of the form $\mathsf{get}\,p'.x$ where x is bound to some value v to the term $\mathsf{get}\,p'.\big(\langle v\rangle_{\mathcal{I}^{P'}}, [0]\big)$. By allowing terms instead of variables in our typing rule T-Get we can use it to assign types to intermediate reduction steps as well. Consider a program

```
place  $x : T :=$ unit on $P$ in
            place  $x' : T' :=$ get $p.x$ on $P'$ in
               end
```

By applying reduction rule E-PlacedVal we substitute every occurence of x by $(\langle\mathsf{unit}\rangle_{\mathcal{I}^P}, [0])$ and obtain $\mathsf{place}\,x' : T' := \mathsf{get}\,p.\,(\langle\mathsf{unit}\rangle_{\mathcal{I}^P}, [0])\ \mathsf{on}\,P'\,\mathsf{in}\,\mathsf{end}$.

Suppose we have $\Delta; \Gamma; \Lambda; P \vdash p' : (P', [0], [l_{p'}])$ and $\Delta; \varnothing; \varnothing; P' \vdash t : (B, [s], [l_t])$. Then, t reduces on P' to some value v. Since this value lives on a remote peer we have to transfer it from P' to P. According to our reduction rules, this this transfer can succeed or fail. The result is a value of basic type $\mathsf{Option}\,(B,[s])$. Furthermore, according to the reduction rules sending the remote request from P to P' and transmitting the result from P' to P introduces the runtime latency $\mathcal{L}(P,P') + \mathcal{L}(P',P)$.

Therefore, typing rule T-Get assigns the type $(\mathsf{Option}\,(B,[s]),[0],[l_{p'} + \mathcal{L}(P,P') + l_t + \mathcal{L}(P',P)])$ to the term $\mathsf{get}\,p'.t$.

Our type system should only accept remote requests between peers that have a possibilty to communicate directly. Hence, we can only accepts the term $\mathsf{get}\,p'.t$ if our current peer P and the remote peer P' are tied. This is expressed by the rule's premise $P \leftrightarrow P'$. Furthermore, the the additional premise $P \neq P'$ ensures that P' is indeed a remote peer. Note that we exclude the scenario of multiple communicating peer instances p_1, p_2 of the same type P. Consider the program

```
place  $x : T := \mathsf{true}\,\mathsf{on}\,P\,\mathsf{in}$
          place  $x' : T' := \mathsf{get}\,p.x\,\mathsf{on}\,P\,\mathsf{in}$
                 end
```

We place a variable x on all instances of peer type P. Then, every instance $p' \in \mathcal{I}^P$ requests x from $p \in \mathcal{I}^P$. This is a remote request for every $p' \neq p$. However, for p it is not and we do not except $\mathsf{get}\,p.x$ as a remote request from p to itself. By including the premise $P \neq P'$ in typing rule T-Get we prohibit this scenario.

Placing Computations Similar to remote value requests, terms of the form $\mathsf{eval}\,t\,\mathsf{on}\,p'$ allow us to place a computation t on a remote peer instance p' and to request its result. The difference, however, is that remote value requests access remote variables and a term $\mathsf{eval}\,t\,\mathsf{on}\,p'$ requests the result of a new computation t.

Its reduction involves the reduction of p' on our current peer P and the reduction of t on p' and hence on P'. Hence, similarly to T-Get, we type p' on P and t on P' with empty local environments.

Suppose we have the typings $\Delta;\Gamma;\Lambda;P \vdash p' : (P',[0],[l_{p'}])$ and $\Delta;\varnothing;\varnothing;P' \vdash t : (B,[s],[l_t])$. Then, similar to typing rule T-Get we can assign the type $(\mathsf{Option}\,(B,[s]),[0],[l_{p'} + \mathcal{L}(P,P') + l_t + \mathcal{L}(P',P)])$ to the term $\mathsf{eval}\,t\,\mathsf{on}\,p'$.

We must ensure that P and P' can comumninicate. Furthermore, for the same reason we considered regarding typing rule T-Get we want to exclude the scenario of multiple communicating peer intances p_1, p_2 of the same peer type P. Hence, T-PlacedEval contains the same premises as T-Get: $P \leftrightarrow P'$ and $P \neq P'$.

2.5.6 Extracting Least Upper Latency Bounds

Whenever we assign a type $(B, [s], [l])$ to a term t, this expresses that t reduces to a value of basic type B and size s. The annotation l represents the latency this reduction entails. So far we thought of l as the exact runtime latency. This view works for non-branching terms like $\mathsf{get}\, p.x$ but not for branching terms like $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$. In the latter case, the condition t_c determines which branch we consider during reduction. Since these branches might entail different runtime latencies we have to reduce the condition before we can calculate the if-term's overall latency.

Non-deterministic Runtime Latency In general terms can have more than one reduction sequence and their reduction is non-deterministic. Consider the condition $t_c = \mathsf{get}\, p.x\, \mathsf{match}\{\mathsf{some}\, x' \Rightarrow \mathsf{true}\}\{\mathsf{none}\, (B, [s]) \Rightarrow \mathsf{false}\}$ and the branches $t_t = \mathsf{get}\, p.x$ and $t_f = \mathsf{none}\, (B, [s])$. The condition reduces to true if the remote request succeeds and to false otherwise. Depending on the result the if-term either reduces to t_t or to t_f. In the first case, further reduction leads to runtime latency but not in the second one. Since remote communication in λ^{lat} is non-deterministic, we cannot statically determine the exact amount of runtime latency the if-term's reduction causes.

Upper Latency Bounds We can, however, bound the runtime-latency based on the information on both branches. Suppose we can statically compute the exact runtime latencies l_c, l_t, l_f caused by reducing the condition t_c and the branches t_t, t_f, respectively. Then we can prove that the runtime-latency caused during the reduction of $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$ does not exceed the sum $l_c + \max\big(l_t, l_f\big)$.

Throughout the rest of this section we use this approach to extract upper latency bounds. Hence, whenever we assign a type $(B, [s], [l])$ to a term t, the latency annotation l represents an upper bound on the runtime latency a reduction of t entails.

Besides if-terms also match-terms force us to extract latency bounds rather than exact values. Furthermore let-terms allow us to explicitely over-approximate runtime latency. Figure 2.10 presents the respective typing rules.

if-Terms Consider the term $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$ and let $(\mathsf{Boolean}, [s_c], [l_c])$, $(B_t, [s_t], [l_t])$ and $(B_f, [s_f], [l_f])$ be the types assigned to t_c, t_t and t_f, respectively. We can use typing rule T-If to type the if-term.

$$\frac{\begin{array}{c}\Lambda \Vdash B_t \approx B_f \qquad \Lambda \vdash s_t =_0 s_f \\ \Delta;\Gamma;\Lambda;P \vdash t_c : (\textsf{Boolean}, [0], [l_c]) \\ \Delta;\Gamma;\Lambda;P \vdash t_t : (B_t, [s_t], [l_t]) \qquad \Delta;\Gamma;\Lambda;P \vdash t_f : (B_f, [s_f], [l_f])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash \textsf{if}\, t_c\, \{t_t\}\, \{t_f\} : (B_t, [s_t], [l_c + \max\big(l_t, l_f\big)])} \text{(T-If)}$$

$$\frac{\begin{array}{c}\Lambda \Vdash B'_s \approx B'_n \qquad \Lambda \vdash s'_s = s'_n \\ \Lambda \Vdash B \approx \widehat{B} \qquad \Lambda \vdash s_B = s_{\widehat{B}} \\ \Delta;\Gamma;\Lambda;P \vdash t : (\textsf{Option}\,(B, [s_B]), [0], [l_t]) \\ \Delta;\Gamma, x \mapsto (B, [s_B]);\Lambda;P \vdash t_s : (B'_s, [s'_s], [l'_s]) \\ \Delta;\Gamma;\Lambda;P \vdash t_n : (B'_n, [s'_n], [l'_n])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash t\,\textsf{match}\{\textsf{some}\, x \Rightarrow t_s\}\{\textsf{none}\,(\widehat{B}, [s_{\widehat{B}}]) \Rightarrow t_n\} : (B'_n, [s'_n], [l_t + \max\,(l'_s, l'_n)])}$$

(T-MatchOpt)

$$\frac{\begin{array}{c}\Lambda \Vdash B'_s \approx B'_n \qquad \Lambda \vdash s'_s = s'_n \\ \Lambda \Vdash B \approx \widehat{B} \qquad \Lambda \vdash s_B = s_{\widehat{B}} \\ \Delta;\Gamma;\Lambda;P \vdash t : (\textsf{List}(B, [s_B]), [s], [l]) \\ \Delta;\Gamma, x_h \mapsto (B, [s_B]), x_t \mapsto (\textsf{List}(B, [s_B]), [s \dot{-} 1]);\Lambda \cup \{0 <_0 s\};P \vdash t_c : (B'_c, [s'_c], [l'_c]) \\ \Delta;\Gamma;\Lambda \cup \{s = 0\};P \vdash t_n : (B'_n, [s'_n], [l'_n])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash t\,\textsf{match}\{\textsf{cons}\, x_h\, x_t \Rightarrow t_s\}\{\textsf{nil}\,(\widehat{B}, [s_{\widehat{B}}]) \Rightarrow t_n\} : (B'_n, [s'_n], [l + \max\,(l'_c, l'_n)])}$$

(T-MatchList)

$$\frac{\begin{array}{c}\Lambda \Vdash B_t \approx B_x \qquad \Lambda \vdash s_t =_0 s_x \qquad \Lambda \vdash l_t \le_0 l_x \\ \Delta;\Gamma;\Lambda;P \vdash t : (B_t, [s_t], [l_t]) \qquad \Delta;\Gamma, x \mapsto (B_x, [s_x]);\Lambda;P \vdash t' : (B', [s'], [l'])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash \textsf{let}\, x \,:\, (B_x, [s_x], [l_x]) \coloneqq t\, \textsf{in}\, t' : (B', [s'], [l_x + l'])}$$

(T-Let)

Figure 2.10: Typing Rules for Latency Bound Extraction

As proposed before, we extract the latency bound $l_c + \max\big(l_t, l_f\big)$. Note that here l_c, l_t and l_f denote upper bounds. Also, this sum is the least upper bound which we can extract without considering the exact value the condition t_c reduces to. As explained above, inspecting the condition's reduction result is not possible in genaral. However, this approach leads to suboptimal bounds for some terms like $\mathsf{if}\,\mathsf{true}\,\{t_t\}\,\{t_f\}$. There are already enough optimizations like *Conditional Constant Propagation* [19] to eliminate dead branches and obsolete conditionals. We can use these to transform our programs in a way that allows us to extract better latency bounds. Hence, this does not effect the quality of our extracted bounds significantly.

The typing rule contains the premises $\Lambda \Vdash B_t \approx B_f$ and $\Lambda \vdash s_t =_0 s_f$. These ensure that both branches reduce to a value of the same basic type and size. Even though their representations B_t, B_f and s_t, s_f might differ. We chose B_t and s_t to represent the if-term's basic type and size.

That is, by applying typing rule T-If, we can assign the type $(B_t, [s_t], [l_c + \max\big(l_t, l_f\big)])$ to the $\mathsf{if}\,t_c\,\{t_t\}\,\{t_f\}$.

Furthermore, the rule requires the condition t_c to be of basic type $\mathsf{Boolean}$. According to the reduction rules we presented in Section 2.4, we can only reduce a term $\mathsf{if}\,v\,\{t_t\}\,\{t_f\}$ for some value v further if $v \in \{\mathsf{true}, \mathsf{false}\}$. By requiring t_c to be of type $\mathsf{Boolean}$ we ensure that it reduces to either true or false. Thereby, we prevent stuck reductions. Consider the term $\mathsf{if}\,(\mathsf{cons}\,t\,t')\,\{\mathsf{unit}\}\,\{\mathsf{unit}\}$ with an unreduced condition. We can reduce this term further until the condition reaches a value $\mathsf{cons}\,v\,v'$. However, eventually the reduction gets stuck because we cannot apply E-IfTrue nor E-IfFalse. Our typing rule T-If rejects this term because the condition does not match the expected basic type $\mathsf{Boolean}$. Thereby, we guarantee that all terms we assign types to eventually reduce to a value.

Pattern Matching allows us to deconstruct Option- and List-values. Both types offer two term constructors some, none and cons, nil. The reduction rules concerning match-terms are similiar to those for if-terms. Hence, the same holds for their corresponding typing rules T-MatchOpt and T-MatchList. The major difference is that a match-term's first branch (i.e., some-, cons-branch) introduces new local variable bindings.

Option-**Patterns** can have the form $\mathsf{some}\,t'$ or $\mathsf{none}\,(B, [s])$. A corresponding match-term has the form $t\,\mathsf{match}\{\mathsf{some}\,x \Rightarrow t_s\}\{\mathsf{none}\,(\widehat{B}, [s_{\widehat{B}}]) \Rightarrow t_n\}$. We can use rule T-MatchOpt to type this term. The rule requires t to have a type of the form $(\mathsf{Option}\,(B, [s_B]), [0], [l_t])$. Similar to rule T-If this prevents stuck reductions for well-typed match-terms.

The rule also contains a premises $\Lambda \Vdash B \approx \widehat{B}$ and $\Lambda \vdash s_B =_0 s_{\widehat{B}}$. These ensure that B, $\widehat{B}$ and s_B, $s_{\widehat{B}}$ represent the same basic element type and size, respectively. Meanwhile we still allow for different representations.

t_n represents the branch which the reduction rules consider if t reduces to a none-value. We can type it in the same typing context $\Delta; \Gamma; \Lambda; P$ we used for the matched term t. Let $(B'_n, [s'_n], [l'_n])$ be its type.

Similarly, t_s represents the branch considered if the matched term reduces to a some-value. Let $\mathsf{some}\, v$ be this value. The pattern $\mathsf{some}\, x$ binds the contained value v to a local variable x with scope t_s. We have to consider this variable binding when we assign a type to t_s. Therefore, we expand our typing environment for local variables Γ by the binding $x \mapsto (B, [s_B])$. That is, we type the branch t_s under consideration of the extended typing context $\Delta; \Gamma, x \mapsto (B, [s_B]); \Lambda; P$. Let $(B'_s, [s'_s], [l'_s])$ be the type we assign to it.

We do not know which branch is considered during the match-term's reduction. Hence, we do not know whether the reduction result has basic type B'_s or B'_n. The same holds for the reduction result's size. Therefore, our typing rule T-MatchOpt contains the premises $\Lambda \Vdash B'_s \approx B'_n$ and $\Lambda \vdash s'_s =_0 s'_n$. We chose B'_n and s'_n to represent the basic type and size of the match-term.

According to our reduction rules presented in Section 2.4, the matched term t is reduced in any case and depending on the result one of the branches t_s, t_n. l_t is an upper bound on the runtime latency caused during the reduction of t. l'_s and l'_n are upper bounds on the runtime latency caused during the reduction of t_s and t_n, respectively. Both do not include l_t. In particular, the typing of t_s assumes that the value bound to x (i.e., v in $\mathsf{some}\, v$) is available without causing any latency. For this reason, we can bound the match-term's runtime latency by $l_t + \max(l'_s, l'_n)$. Note that as in T-If this is the least upper bound we can compute without statically determining which branch is considered during reduction.

Following these steps, our reduction rule T-MatchOpt assignes the type $(B'_n, [s'_n], [l'_n])$ to the term $t\, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t_s\}\{\mathsf{none}\, (\widehat{B}, [s_{\widehat{B}}]) \Rightarrow t_n\}$.

List-Patterns are similar to Option-patterns and so is the corresponding typing rule T-MatchList almost analagous to T-MatchOpt. There is, however, a difference in typing the first branch.

The some-constructor wraps a single element into a some-term. The pattern $\mathsf{some}\, x$ in T-MatchOpt binds this element to a local variable x and we consider its typing when we type the some-branch.

Consider the term $t\,\mathsf{match}\{\mathsf{cons}\,x_h\,x_t \Rightarrow t_c\}\{\mathsf{nil}\,(\widehat{B},[s_{\widehat{B}}]) \Rightarrow t_n\}$ where t represents a list of type $(\mathsf{List}(B,[s_B]),[s],[l])$. The list constructor $\mathsf{cons}\cdot\cdot$ prepends an element to a list and the respective pattern $\mathsf{cons}\,x_h\,x_t$ introduces two corresponding variables in the cons-branch t_c. We have to consider both variables and the types of their bound values when we type t_c. Variable x_t binds the tail of list t which contains one element less that t. Therefore, we have to consider the size decrease from s to $s \dot{-} 1$. Similar to T-MatchOpt, we extend our local variable environment Γ by the bindings $x_h \mapsto (B,[s_B])$, $x_t \mapsto (\mathsf{List}(B,[s_B]),[s \dot{-} 1])$.

As we see later in Section 2.5.7 we use this size decrease to construct size-decreasing recursive functions and prove their termination. The arithmetic term $s \dot{-} 1$ represents substraction over natural numbers and again yields a natural numbers. In case $s = 0$ the result is 0. That is, in order to prove that the tail contains indeed one element less than the original list, we need to assume $s > 0$. This assumption is fulfilled when the reduction considers the cons-branch. Therefore, additionally to extending our typing environment Γ, we also extend our set of arithmetic assumptions Λ by the assumption $s > 0$. That is, we type the cons-branch t_c under consideration of the extended typing context:

$\Delta;\ \Gamma, x_h \mapsto (B,[s_B]), x_t \mapsto (\mathsf{List}(B,[s_B]),[s \dot{-} 1]);\ \Lambda \cup \{s > 0\};\ P$
Otherwise T-MatchList is analagous to T-MatchOpt.

Over-Approximation So far we saw how we can extract least upper bounds for a term's runtime latency. Sometimes, however, it is desirable to state explicit bounds that are not necessarily optimal and prove their correctness. Consider the term $\mathsf{let}\,x : (B_x,[s_x],[l_x]) \coloneqq t_x\,\mathsf{in}\,t'$. It binds the result of computation t to a variable x and thereby makes it accessible to the computation t'. The type ascription $(B_x,[s_x],[l_x])$ expresses the expectation that t_x reduces to a value of basic type B_x and size s_x with a runtime latency of at most l_x. We later see that the corresponding typing rule T-Let only types the let-term if t meets these expectations. Hence, the computation t' can rely on these assumptions.

In practice implementations frequently change. Suppose we replace t_x by a faster computation t of type $(B_x,[s_x],[l_t])$ with $l_t < l_x$. It excels the latency contraints expressed by l_x. This should not be regarded as an error but be accepted by our type system.

let-Terms Consider the term $\mathsf{let}\,x : (B_x,[s_x],[l_x]) \coloneqq t\,\mathsf{in}\,t'$ and suppose t is of type $(B_t,[s_t],[l_t])$. Typing rule T-Let requires that B_x, B_t and s_x, s_t

represent the same basic types and sizes, respectively, and that l_t does not exceed l_x. This is expressed by the premises $\Lambda \Vdash B_t \approx B_x$, $\Lambda \vdash s_t =_0 s_x$ and $\Lambda \vdash l_t \leq_0 l_x$, respectively.

The variable x is available in t', hence, we also have to consider its typing when we type t'. Similar to the some-branch in rule T-MatchOpt, we type t' in the extended typing context $\Delta; \Gamma, x \mapsto (B_x, [s_x]); \Lambda; P$.

Let $(B', [s'], [l'])$ be the type we assign to t'. The typing of t' relies on the assumption that the local variable x is readily available. Hence, the latency bound l' does not include the latency caused during the reduction of t. Furthermore, note that according to the reduction rules presented in Section 2.4 we reducue t to a value independently of whether variable x is referenced in t' or not. Therefore, we have to correct the bound. The type assigned to the let-term relies on the provably correct type assumption $x : (B_x, [s_x], [l_x])$. Therefore we bound the overall latency by $l_x + l'$. Accordingly, rule T-Let assigns the type $(B', [s'], [l'])$ to the term $\mathsf{let}\, x : (B_x, [s_x], [l_x]) := t \,\mathsf{in}\, t'$.

2.5.7 Functions

In the standard typed λ-calculus a function f transforming an argument of type A into a value of type A' has the type $A \to A'$. Its type system accepts an application $f\,a$ only if a matches the expected type A to prevent stuck reductions.

In λ^{lat} we follow this approach but additionally annotate basic types by sizes and latency bounds. In particular, we assign a type of the form $(B', [s'], [l'])$ to every well-typed function application $f\,a$. It expresses that the application provably reduces to a value of basic type B' and size s'. Furthermore, the typing expresses that we can prove that the runtime-latency arising throughout this reduction does not exceed the upper bound l'.

The type we assign to the function f must reflect that. This leads us to a type of the form $B \to (B', [s'], [l'])$. Let id be an identity function and a_1, a_2 values of non-equal types $(B_1, [s_1], [l_1])$, $(B_2, [s_2], [l_2])$, respectively. Then the applications $id\,a_1$, $id\,a_2$ respectively have type $(B_1, [s_1], [l_1])$, $(B_2, [s_2], [l_2])$, too. This implies the following contradiction: $(B', [s'], [l']) = (B_1, [s_1], [l_1]) \not\approx (B_2, [s_2], [l_2]) = (B', [s'], [l'])$.

We could restore consistency by making the argument size and latency explicit in function types, leading to $(B, [s], [l]) \to (B', [s'], [l'])$. This would restrict functions to accept only arguments of a specific latency. However, for any function application $f\,a$ only the kind value a reduces to should matter, not the way we compute it. Instead we choose a more flexible approach and only make the argument's expected basic type and

$$\frac{\Delta;\Gamma,x\mapsto(B,[s]);\Lambda;P\vdash t:(B',[s'],[l'])}{\Delta;\Gamma;\Lambda;P\vdash \lambda x:(B,[s]).t:((B,[s])\to(B',[s'],[l']),[0],[0])}\ \text{(T-Abs)}$$

$$\frac{\begin{array}{c}\Lambda\Vdash B_f\approx B_a \qquad \Lambda\vdash s_f=_0 s_a\\ \Delta;\Gamma;\Lambda;P\vdash f:((B_f,[s_f])\to(B',[s'],[l']),[0],[l_f]) \qquad \Delta;\Gamma;\Lambda;P\vdash a:(B_a,[s_a],[l_a])\end{array}}{\Delta;\Gamma;\Lambda;P\vdash f\,a:(B',[s'],[l_f+l_a+l'])}\ \text{(T-LocalApp)}$$

$$\frac{\begin{array}{c}\Lambda\Vdash B_f\approx B_a \qquad \Lambda\vdash s_f=_0 s_a\\ \Delta;\Gamma;\Lambda;P\vdash p':(P',[0],[l_{p'}]) \qquad P\neq P' \qquad P\leftrightarrow P'\\ \Delta;\varnothing;\varnothing;P'\vdash f:((B_f,[s_f])\to(B',[s'],[l']),[0],[l_f]) \qquad \Delta;\Gamma;\Lambda;P\vdash a:(B_a,[s_a],[l_a])\end{array}}{\Delta;\Gamma;\Lambda;P\vdash \mathsf{remoteCall}\,p'.f\,a:(B',[s'],[l_{p'}+l_a+\mathcal{L}(P,P')+l_f+l'+\mathcal{L}(P',P)])}\ \text{(T-RemoteApp)}$$

Figure 2.11: Typing Rules for Functions with fixed Argument Sizes

size explicit in function types. Hence, function types in λ^{lat} have the form $(B,[s])\to(B',[s'],[l'])$. This suffices to prevent stuck reductions regarding function applications. Throughout this section we also see that it carries enough information to extract type-level latency bounds.

Figure 2.11 presents the typing rules for λ-abstraction as well as local and remote function application.

λ-Terms in λ^{lat} have the form $\lambda x:(B,[s]).t$ and we can use typing rule T-Abs to type them. The type ascription $(B,[s])$ expresses that the function expects an argument of basic type B and size s. A supplied argument is bound to variable x which is made locally available in the function body t. Hence, we have to consider the variable's typing when we type the body t. Similar to typing rule T-Let, we type t under consideration of the extended typing context $\Delta;\Gamma,x\mapsto(B,[s]);\Lambda;P$.

Suppose we can prove in this extended context that t has type $(B',[s'],[l'])$. Then we can also prove that the function $\lambda x:(B,[s]).t$ transforms any argument of basic type B and size s into a value of type $(B',[s'],[l'])$. Therefore, we can assign the basic type $(B,[s])\to(B',[s'],[l'])$ to it.

Furthermore, λ^{lat} does not offer means to decompose a λ-term. Hence, we do not view it as a composite like $\mathsf{cons}\,t_1\,t_2$ and augment its basic type with the size 0. This holds for all functions independent of the argument size s or the size its body t might reduce to.

According to Definition 9 λ-terms are values and hence do not need further reduction. In particular, the term $\lambda x : (B, [s]).t$ on itself does not cause any remote communication. Hence, we can bound its latency by 0.

Accordingly, T-Abs assigns the type $((B, [s]) \rightarrow (B', [s'], [l']), [0], [0])$ to the term $\lambda x : (B, [s]).t$.

Local Function Applications have the form $f\,a$. In order to prevent stuck reductions, the responsible tying rule T-LocalApp accepts such an application only if f is a function and a meets its expectations.

That is, f must have a basic type of the form $(B_f, [s_f]) \rightarrow (B', [s'], [l'])$. Let $((B, [s]) \rightarrow (B', [s'], [l']), [0], [l_f])$ be its fully annotated type. Furthermore, the type $(B_a, [s_a], [l_a])$ of a must provably match the basic type and size expected by f, i.e., $\Lambda \Vdash B_f \approx B_a$ and $\Lambda \vdash s_f =_0 s_a$. Then we know, that the application $f\,a$ reduces to a value of basic type B' and size s'.

According to our reduction rules presented in Section 2.4, this reduction involves the reductions of f and a. This holds independently of whether f actually uses the argument in any way or not. The types of f and a present the type-level bounds l_f, l_a for the runtime latency the reduction of f and a to some values v_f and v_a involves, respectively. According to the basic type of f, it maps an argument to a value of type $(B', [s'], [l'])$. That is, the reduction of the application $v_f\,v_a$ causes a runtime latency bounded by l'. Hence, we can bound the runtime latency of the original application $f\,a$ by $l_f + l_a + l'$.

Under these conditions rule T-LocalApp assigns the type $(B', [s'], [l_f + l_a + l'])$ to the application $f\,a$.

Remote Function Applications have the form $\textsf{remoteCall}\, p'.f\,a$ for an instance p' of some remote peer P', a function f placed on P' and a local argument a. Reducing this term means we have to reduce p' and a on our current peer P to values $v_{p'}$ and v_a. Afterwards we can send v_a to $v_{p'}$ and request the result of the remote computation $f\,v_a$.

Hence, the main difference to a local function application $f\,a$ is that we have to reduce the argument and the application on different peers. Suppose we have the typings $\Delta; \varnothing; \varnothing; P' \vdash f : ((B_f, [s_f]) \rightarrow (B', [s'], [l']), [0], [l_f])$, $\Delta; \Gamma; \Lambda; P \vdash a : (B_a, [s_a], [l_a])$ and $\Delta; \Gamma; \Lambda; P \vdash p' : (P', [0], [l_{p'}])$. Then we can apply the typing rule T-RemoteApp to type our remote application. This rule is almost analagous to T-LocalApp. However, it includes the latency caused by sending the request from P to P' and transmitting the result from P' to P. Hence, T-RemoteApp assigns the type $(B', [s'], [l_{p'} + l_a + \mathcal{L}(P, P') + l_f + l' + \mathcal{L}(P', P)])$ to the term $\textsf{remoteCall}\, p'.f\,a$.

Similar to the typing rule T-Get we need to ensure that P' indeed is a remote peer. Otherwise we run into scenarios where we accept a remote request $\mathsf{remoteCall}\, p.f\, a$ from p to itself. Furthermore, we need to ensure that P and P' have a possibility to communicate directly. Typing rule T-RemoteApp ensures both by including the premises $P \not\equiv P'$ and $P \leftrightarrow P'$.

2.5.7.1 Size-Dependent Functions

Above we introduced function types $(B, [s]) \to (B', [s'], [l'])$. This type expresses that arguments of basic type B and size s are transformed into values of basic type B' and size s'. Furthermore this transformations entails a latency bounded by l'.

Consider the function

```
conj_2 =  λx : (List(Boolean, [0]), [2]).
              x match
                {cons x_h x_t ⇒  if x_h  { x_t match
                                            {cons x'_t n ⇒ x'_t}
                                            {nil (Boolean, [0]) ⇒ true}
                                         }
                                         {false}
                }
                {nil (Boolean, [0]) ⇒ true}
```

It takes a pair of $\mathsf{Booleans}$ (i.e., a $\mathsf{Boolean}$ list of size 2) and computes their conjunction (i.e. a $\mathsf{Boolean}$ of size 0) without any remote communication. According to T-Abs, it has the type $(\mathsf{List}(\mathsf{Boolean}, [0]), [2]) \to (\mathsf{Boolean}, [0], [0])$. Similarly, the identity function for boolean pairs $id_{\mathbb{B}2} = \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [2]).x$ has type the type: $(\mathsf{List}(\mathsf{Boolean}, [0]), [2]) \to (\mathsf{List}(\mathsf{Boolean}, [0]), [2], [0])$. Here, the result size equals the fixed input size 2. Both function types only involve arithmetic terms without any variables.

In general, however, the arithmetic terms s', l' describing the result's size and latency are not necessarily variable free. The rules T-Abs, T-LocalApp, T-RemoteApp presented above allow them to contain variables as long as they are contained in the local typing environment Γ.

The size annotation s in $id_{\mathbb{B}2}$ expresses a fixed argument size. Let us replace it by a free variable $\hat{s}$. Then we get $id_{\mathbb{B}s} = \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]).x$. According to our reduction rules presented in Section 2.4 $id_{\mathbb{B}s}$ behaves like an identity function for boolean lists of arbitrary size. However, we want to prevent stuck reduction caused by wrongly named variable references. Therefore, our type system only accepts references to previously introduced variables. That is, we have to introduce $\hat{s}$ explicitly as arithmetic variable

$$\frac{\begin{array}{c} x_s \in FAV(t) \\ \Delta; \Gamma, x_s \mapsto \mathbb{N}; \Lambda; P \vdash t : (B, [s], [l]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \forall (x_s : \mathbb{N}) \,.\, t : (\forall (x_s : \mathbb{N}) \,.\, B, [s], [l])} \ (\text{T-ForAll})$$

$$\frac{\begin{array}{c} x_s \notin FAV(t) \\ \Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \forall (x_s : \mathbb{N}) \,.\, t : (B, [s], [l])} \ (\text{T-ForAllIgnore})$$

$$\frac{\begin{array}{c} x_s \in FAV(t) \\ \Delta; \Gamma, x_s \mapsto \mathbb{N}; \Lambda \cup \{x_s <_0 u\}; P \vdash t : (B, [s], [l]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \forall (x_s : \mathbb{N}) < u \,.\, t : (\forall (x_s : \mathbb{N}) < u \,.\, B, [s], [l])} \ (\text{T-ForAllBounded})$$

$$\frac{\begin{array}{c} x_s \notin FAV(t) \\ \Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l]) \end{array}}{\Delta; \Gamma; \Lambda; P \vdash \forall (x_s : \mathbb{N}) < u \,.\, t : (B, [s], [l])} \ (\text{T-ForAllBoundedIgnore})$$

Figure 2.12: Typing Rules for Size Abstractions

and specify the range of natural numbers we may substitute for $\hat{s}$.

Abstraction over Sizes In λ^{lat} we can use a universal quantifier $\forall (\hat{s} : \mathbb{N})$ to introduce $\hat{s}$ as a variable ranging over natural numbers. Using this, we can correct our definition of $id_{\mathbb{B}s}$ to $id_{\forall\mathbb{B}} = \forall (\hat{s} : \mathbb{N}) \,.\, \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]).x$. The scope of variable $\hat{s}$ is $\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]).x$ and thereby our type system accepts the reference to $\hat{s}$ as the function's argument size.

Suppose we used a relaxed version of T-Abs to type this function and thereby assigned it the following basic type: $(\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]) \rightarrow (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}], [0])$. Then our function type would contain the free variable $\hat{s}$. Our type system forbids free variables in types for the same reason as in terms. That is, we have to make the type and range of $\hat{s}$ explicit in the type of $id_{\forall\mathbb{B}}$. We do this by lifting the quantification $\forall (\hat{s} : \mathbb{N})$ to the type level and thereby get the basic type $\forall (\hat{s} : \mathbb{N}) \,.\, \big((\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]) \rightarrow (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}], [0])\big)$. Here, the quantifier's scope is the complete original basic type $(\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]) \rightarrow (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}], [0])$. Therefore, the references to $\hat{s}$ in the argument and result type are valid.

Typing Universal Quantifications In λ^{lat} quantifications $\forall (\hat{s} : \mathbb{N}) \,.\, t$ are treated as any other placed term and may therefore occur in any position. In particular, we may use quantifiers to introduce arithmetic variables that are never used. According to our definition of values (Def. 9) a term $\forall(\hat{s} : \mathbb{N}) \,.\, t$ is a value if t is one and if t contains $\hat{s}$ as free variable. Our reduction rules presented in Section 2.4 respect that and eliminate unneeded quantifiers. That is, for any term t with $\hat{s} \notin FAV(t)$, they reduce $\forall(\hat{s} : \mathbb{N}) \,.\, t$ to t. The quantifier $\forall(\hat{s} : \mathbb{N})$ does not introduce any relevant information to its scope t. Hence, this reduction step preserves the term's computational content. Since types are approximations of a computation's runtime behaviour, our typing rules should preserve the term's type as well. Figure 2.12 presents the typing rules regarding quantifiers.

We therefore introduce two typing rules T-ForAll and T-ForAllIgnore that differentiate between quantifiers that add relevant information to their scope and such that do not.

Consider the universally quantified term $\forall(\hat{s} : \mathbb{N}) \,.\, t$. In case the quantified variable $\hat{s}$ occurs free in t we can use T-ForAll to type it. Otherwise we have to use T-ForAllIgnore. These requirements are expressed by the rules' premise $\hat{s} \in FAV(t)$ and $\hat{s} \notin FAV(t)$, respectively.

Suppose t contains $\hat{s}$ as free variable, then we have to consider its type when we assign a type to the scope t. That is, we type t in the extended typing context $\Delta; \Gamma, \hat{s} \mapsto \mathbb{N}; \Lambda; P$. Suppose t has type $(B, [s], [l])$ under consideration of this extended context. Our original context $\Delta; \Gamma; \Lambda; P$ does not know about $\hat{s}$. Hence, the basic type B contains $\hat{s}$ as a free variable. We have to make the introduction of $\hat{s}$ explicit on the type level. Thus, rule T-ForAll assigns the type $(\forall(\hat{s} : \mathbb{N}) \,.\, B, [s], [l])$ to the term $\forall(\hat{s} : \mathbb{N}) \,.\, t$.

Note that according to the structure of placed terms in λ^{lat} and the typing rules we presented so far, the introduced variable $\hat{s}$ does not affect the size and latency annotations of t's type. For instance, we saw that the identity function for boolean lists $id_{\forall\mathbb{B}}$ has type $\big(\big(\forall(\hat{s} : \mathbb{N}) \,.\, ((\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]) \rightarrow (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}], [0]))\big), [0], [0]\big)$. The function's definition has the form $id_{\forall\mathbb{B}} = \forall(\hat{s} : \mathbb{N}) \,.\, t$ and we see that the quantified variable only affects the basic type $(\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]) \rightarrow (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}], [0])$.

Suppose t does not conatin $\hat{s}$ as a free variable. Then, we do not have to consider $\hat{s}$ when we type t. Let $(B, [s], [l])$ be the type we assign to t considering our unmodified typing context $\Delta; \Gamma; \Lambda; P$. This type does not contain $\hat{s}$ as a free variable. Hence, typing rule T-ForAllIgnore assigns the type $(B, [s], [l])$ to the term $\forall(\hat{s} : \mathbb{N}) \,.\, t$.

Note that reduction rule E-ForAllElim eliminates unneeded quantifiers. For instances, it reduces $\forall(x : \mathbb{N}) \,.\, \mathsf{unit}$ to unit. Suppose we did not differentiate between needed and unneeded quantifiers in our typing rules. Then this reduction step would reduce a term of type $(\forall(s:\mathbb{N})\,.\,\mathsf{Unit},[0],[0])$ to one of type $(\mathsf{Unit},[0],[0])$. That is, by introducing separate rules T-ForAll and T-ForAllIgnore, we ensure that our reduction rules preserve the reduced term's type.

We call functions with a basic type of the form $\forall(s:\mathbb{N})\,.\,(B,[s]) \to (B',[s'],[l'])$ *size-dependent functions* since the result of an application (potentially) depends on the input size. We later see a restricted form of size-dependent functions, as well. Hence, we postpone our formal definition until this form has been introduced, too. Figure 2.13 presents the typing rules regarding the application of size-dependent functions.

Local Size-Dependent Applications Consider the identity function for boolean lists of arbitrary size $id_{\forall\mathbb{B}}$ from above and let a be a boolean list of size s_a. Then a has type $(\mathsf{List}(\mathsf{Boolean},[0]),[s_a],[l_a])$ and $id_{\forall\mathbb{B}}$ has type $(\forall(\hat{s}:\mathbb{N})\,.\,(\mathsf{List}(\mathsf{Boolean},[0]),[\hat{s}]) \to (\mathsf{List}(\mathsf{Boolean},[0]),[\hat{s}],[0]),[0],[0])$. Let v_a be the value a reduces to. Then the application $id_{\forall\mathbb{B}}\,a$ reduces to v_a with the runtime latency caused during the reduction of a to v_a. According to the type of $id_{\forall\mathbb{B}}$, the function application does not add any latency. Hence, the we must type the application $f\,a$ by $(\mathsf{List}(\mathsf{Boolean},[0]),[s_a],[l_a])$. This resembles the result type of $id_{\forall\mathbb{B}}$ but with the additional latency caused by the supplied argument a. We see that the application's type depends on the input size, which is why we call this application *size-dependent*.

In general, however, not only the result's size but also its basic type and latency can depend on the input's size. Imagine a function *remConj* that accepts boolean lists of arbitrary size and processes one element after the other. For each element it computes the conjunction with some remote value and stores it in the list it finally returns. Later section 2.5.7.2 we see how to define such a function. For now, let l be the runtime latency introduced by the processing of a single element. Then *remConj* has the basic type $\forall(\hat{s}:\mathbb{N})\,.\,(\mathsf{List}(\mathsf{Boolean},[0]),[\hat{s}]) \to (\mathsf{List}(\mathsf{Boolean},[0]),[\hat{s}],[l\cdot\hat{s}])$. Let a be a boolean list of size s_a and consider the application *remConj* a. The term $l\cdot\hat{s}$ describes the latency caused by the function application depending on an input size $\hat{s}$. We can calculate a latency bound for the application by substituting s_a for $\hat{s}$, that is, $(l\cdot\hat{s})[\hat{s}\mapsto s_a] = l\cdot s_a$. We see that size-dependent functions allow us to compute latency bounds depending on the number of steps a function takes. Later we see how we can generalize this principle to extract latency bounds for a recursive functions.

Let us consider the general case of a size-dependent application $f\,a$ and the responsible typing rule T-SizeDepLocalApp. Suppose the func-

$$\frac{\begin{array}{c}\Lambda \Vdash B_f \approx B_a \\ \Delta;\Gamma;\Lambda;P \vdash a : (B_a, [s_a], [l_a]) \\ \Delta;\Gamma;\Lambda;P \vdash f : (\forall(s:\mathbb{N})\,.\,\big((B_f,[s]) \to (B',[s'],[l'])\big), [0], [l_f])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash f\,a : (B'[s \mapsto s_a], [s'[s \mapsto s_a]], [l_f + l_a + l'[s \mapsto s_a]])}\ \text{(T-SizeDepLocalApp)}$$

$$\frac{\begin{array}{c}\Lambda \Vdash B_f \approx B_a \\ \Delta;\Gamma;\Lambda;P \vdash p' : (P', [0], [l_{p'}]) \qquad P \neq P' \\ \Delta;\Gamma;\Lambda;P \vdash a : (B_a, [s_a], [l_a]) \\ \Delta;\Gamma;\Lambda;P' \vdash f : (\forall(s:\mathbb{N})\,.\,\big((B_f,[s]) \to (B',[s'],[l'])\big), [0], [l_f])\end{array}}{\begin{array}{l}\Delta;\Gamma;\Lambda;P \vdash \mathsf{remoteCall}\, p'.f\,a : (B'[s \mapsto s_a], [s'[s \mapsto s_a]], \\ \qquad [l_{p'} + l_a + \mathcal{L}(P,P') + l_f + l'[s \mapsto s_a] + \mathcal{L}(P',P)])\end{array}}\ \text{(T-SizeDepRemoteApp)}$$

$$\frac{\begin{array}{c}\Lambda \Vdash B_f \approx B_a \qquad \Lambda \vdash s_a <_0 u \\ \Delta;\Gamma;\Lambda;P \vdash f : (\forall(s:\mathbb{N}) < u\,.\,\big((B_f,[s]) \to (B',[s'],[l'])\big), [0], [l_f]) \\ \Delta;\Gamma;\Lambda;P \vdash a : (B_a, [s_a], [l_a])\end{array}}{\Delta;\Gamma;\Lambda;P \vdash f\,a : (B'[s \mapsto s_a], [s'[s \mapsto s_a]], [l_f + l_a + l'[s \mapsto s_a]])}\ \text{(T-BoundedSizeDepLocalApp)}$$

$$\frac{\begin{array}{c}\Lambda \Vdash B_f \approx B_a \qquad \Lambda \vdash s_a <_0 u \\ \Delta;\Gamma;\Lambda;P \vdash p' : (P', [0], [l_{p'}]) \qquad P \neq P' \\ \Delta;\Gamma;\Lambda;P \vdash a : (B_a, [s_a], [l_a]) \\ \Delta;\Gamma;\Lambda;P' \vdash f : (\forall(s:\mathbb{N}) < u\,.\,\big((B_f,[s]) \to (B',[s'],[l'])\big), [0], [l_f])\end{array}}{\begin{array}{l}\Delta;\Gamma;\Lambda;P \vdash \mathsf{remoteCall}\, p'.f\,a : (B'[s \mapsto s_a], [s'[s \mapsto s_a]], \\ \qquad [l_{p'} + l_a + \mathcal{L}(P,P') + l_f + l'[s \mapsto s_a] + \mathcal{L}(P',P)])\end{array}}\ \text{(T-BoundedSizeDepRemoteApp)}$$

Figure 2.13: Typing Rules for Size-Dependent Function Applications

tion f and its argument a have types of the form $(\forall(s : \mathbb{N}) . \big((B_f, [s]) \rightarrow (B', [s'], [l'])\big), [0], [l_f])$ and $(B_a, [s_a], [l_a])$, respectively. In order to prevent stuck reductions, the typing T-SizeDepLocalApp accepts the application only if the arguments basic type meets the function's expectations. This is expressed by the premise $\Lambda \Vdash B_f \approx B_a$. Suppose this holds and suppose f and a are values. Then, we can compute the type of the application's result by considering the function's result type $(B', [s'], [l'])$ and subsituting the argument size s_a for the universally quantified size variable s. Thereby we get $(B'[s \mapsto s_a], [s'[s \mapsto s_a]], [l'[s \mapsto s_a]])$. However, in general we also have to consider the runtime latency caused during the reduction of f and a. Hence, rule T-SizeDepLocalApp assigns the type $(B'[s \mapsto s_a], [s'[s \mapsto s_a]], [l_f + l_a + l'[s \mapsto s_a]])$ to the application $f\,a$.

Remote Size-Dependent Applications can be typed by the rule T-SizeDepRemoteApp. Just as T-SizeDepLocalApp is a refinement of T-LocalApp, its remote version T-SizeDepRemoteApp is an analogous refinement of T-RemoteApp.

Restricting Universal Qantification Above we saw how to define well-typed functions that can handle arbitrary argument sizes. Sometimes, however, functions can only handle certain range of sizes. Consider the function $last_{\mathbb{B}3}$ returning the last element of a boolean triple:

```
$last_{\mathbb{B}3}$ =  $\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [3]).$
             $x$ match
                $\{\mathsf{cons}\, x_h\, x_t \Rightarrow$   $x_t$ match
                                  $\{\mathsf{cons}\, x'_h\, x'_t \Rightarrow$   $x'_t$ match
                                                    $\{\mathsf{cons}\, x''_h\, n \Rightarrow \mathsf{some}\, x''_h\}$
                                                    $\{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{some}\, x'_h\}$
                                  $\}$
                                  $\{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{some}\, x_h\}$
                $\}$
                $\{\mathsf{nil}\,(\mathsf{Boolean}, [0]) \Rightarrow \mathsf{none}\,(\mathsf{Boolean}, [0])\}$
```

Considering the function's runtime behaviour, we see that it returns a list's last element as long as the list's length does not exceed 3. However, our type system rejects applications to lists of any size other 3. As we saw above, we can relax this restriction by replacing the expected argument size by a universally quantified size variable. Let t_{last} denote the nested match-term forming the body of function $last_{\mathbb{B}3}$. Then $\forall(\hat{s} : \mathbb{N}) . \lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]).t_{last}$ is a relaxed version of $last_{\mathbb{B}3}$ that accepts boolean lists of arbitrary size. Furthermore, we can prove that such an application successfully reduces to a value. However, for any argument of size greater than 3 the function does not meet its original specification. Hence, it makes sense to only allow arguments of any size $\hat{s} < 4$.

In λ^{lat} we can restrict quantifiers $\forall(\hat{s} : \mathbb{N})$ to the range of natural numbers below a strict upper bound u. That is: $\forall(\hat{s} : \mathbb{N}) < u$. We can use this restricted quantification to relax the function $last_{\mathbb{B}3}$ appropriately. The function $\forall(\hat{s} : \mathbb{N}) < 4\,.\,\lambda x : (\mathsf{List}(\mathsf{Boolean}, [0]), [\hat{s}]).t_{last}$ accepts lists with up to 3 elements and adheres to the original specification of $last_{\mathbb{B}3}$.

The typing rules responsible for terms of the form $\forall(s : \mathbb{N}) < u\,.\,t$ are T-ForAllBounded and T-ForAllBoundedIgnore. They are almost analagous to T-ForAll and T-ForAllIgnore. The only difference is the additional upperbound u. When we type the quantifier's scope t, we have to consider that s ranges over natural numbers strictly smaller than u. Hence, we add the arithmetic assumption $s < u$ to our set of arithmetic assumptions Λ. That is, we type the scope t under the extended typing context $\Delta; \Gamma, s \mapsto \mathbb{N}; \Lambda \cup \{s < u\}; P$.

The typing rules responsible for size-dependent local and remote applications $f\,a$ and $\mathsf{remoteCall}\,p'.f\,a$ are T-BoundedSizeDepLocalApp and T-BoundedSizeDepRemoteApp. They are almost analogous to T-SizeDepLocalApp and T-SizeDepRemoteApp, respectively. They only feature an additional premise demanding a proof that the argument size does not exceed the upper bound.

Suppose a and f have the types $(B_a, [s_a], [l_a])$ and $(\forall(s : \mathbb{N}) < u\,.\,\big((B_f, [s]) \rightarrow (B', [s'], [l'])\big), [0], [l_f])$, respectively. Both T-BoundedSizeDepLocalApp and T-BoundedSizeDepRemoteApp contain the premise $\Lambda \vdash s_a <_0 u$. That is, we can only apply them if we can prove that our argument is small enough to meet the expectations of f.

Definition 30 (Size-Dependent Function)**.** *Let $\Delta; \Gamma; \Lambda; P$ be a typing context, B a basic type and s, l, u arithmetic terms. Furthermore, let $\hat{s}$ be a variable and t, t' placed terms with $\Delta; \Gamma; \Lambda; P \vdash t : (\forall(\hat{s} : \mathbb{N})\,.\,B, [s], [l])$ and $\Delta; \Gamma; \Lambda; P \vdash t' : (\forall(\hat{s} : \mathbb{N}) < u\,.\,B, [s], [l])$. Then we call t and t'* size-dependent functions *regarding the typing context $\Delta; \Gamma; \Lambda; P$.*

2.5.7.2 Size-Decreasing Recursion

Recursion is a convenient way to define size-dependent functions. In λ^{lat} we can define recursive functions by means of a fixpoint operator fix. In Section 2.4.3.3 we already saw that allowing unrestricted recursion prevents us from extracting type-level latency bounds. In this section we follow up on the ideas presented there and explain the restrictions our type system puts on the fixpoint operator.

The standard implementation of a fixpoint operator expects a function of the form $f = \lambda g : T.t$ and type $T \to T$ (for some unannotated type T). The function supplied for g acts as continuation. A fixpoint application $\mathsf{fix}\, f$ expands to $f\,(\mathsf{fix}\, f)$. That is, we supply f as its own continuation and thereby define a recursive function.

The problem is that this allows the definition of non-terminating recursive functions. Consider the identity function $id_T = \lambda x : T.x$ and the application $\mathsf{fix}\, id_T$. It expands to $id_T\,(\mathsf{fix}\, id_T)$ and then reduces to $\mathsf{fix}\, id_T$, again. Hence, this application diverges but does not cause any runtime-latency.

Consider the modified version $id_{T,rem} = \lambda x : T.(\mathsf{get}\, p'.x')\, \mathsf{match}\{\mathsf{some} \ldots \Rightarrow x\}\{\mathsf{none}\,(\ldots, [\ldots]) \Rightarrow x\}$. It behaves like the original identity function id_T but includes a remote request to some variable x'. Consider its fixpoint application $\mathsf{fix}\, id_{T,rem}$ and let l be the runtime latency caused by one expansion-reduction cycle. Then, the overall runtime latency caused during the diverging reduction sequence of $\mathsf{fix}\, id_{T,rem}$ is the diverging series $\Sigma_{j=0}^{\infty} l$. Since this series cannot be bounded by any natural number, we cannot extract a type-level bound.

We need to restrict our fixpoint operator such that it prevents the construction of diverging functions. Furthermore, we have to ensure that these functions have a form that allows us to extract latency bounds.

In the previous section we saw how to we can define unbounded and bounded size-dependent functions. Suppose we want to define an unbounded size-dependent function

Consider the function $conj_2$ expecting a pair of booleans and returning their conjunction.

```
conj₂ =  λx : (List(Boolean, [0]), [2]).
                x  match
                   {cons x_l x_t ⇒   if  x_t   {  x_t match
                                                   {cons x_r n ⇒ x_r}
                                                   {nil (Boolean, [0]) ⇒ true}
                                               }
                                               {false}
                   }
                   {nil (Boolean, [0]) ⇒ true}
```

Its type is $(\mathsf{List}(\mathsf{Boolean}, [0]), [2]) \to (\mathsf{Boolean}, [0], [0])$. In general, boolean conjunction is defined for an arbitrary number number of elements. Hence, we would generalize $conj_2$ accordingly. Note that we can relax our definition such that it also accepts lists size 0 and 1 without actually changing its implementation. The result is

```
conj_{<3} =   ∀(s : ℕ) < 3.
                  λx : (List(Boolean, [0]), [s]).
                        x match
                          {cons x_l x_t ⇒  if x_t  { x_t match
                                                         {cons x_r n ⇒ x_r}
                                                         {nil (Boolean, [0]) ⇒ true}
                                                     }
                                                     {false}
                          }
                          {nil (Boolean, [0]) ⇒ true}
```

and has type $\forall(s:\mathbb{N}) < 3\,.\big((\mathsf{List}(\mathsf{Boolean},[0]),[2]) \rightarrow (\mathsf{Boolean},[0],[0])\big)$.

Conjunction is associative. Hence, for any number u and any a list of booleans $b_1, ...b_{u+1}$, we know that $\bigwedge_{j=1}^{u+1} b_j = b_1 \wedge (\bigwedge_{j=2}^{u+1})$. That is, provided we have a function that computes the conjunction for up to u elements (i.e., $\bigwedge_{j=2}^{u+1}$), we can extend it to accept $u+1$ (i.e., $b_1 \wedge (\bigwedge_{j=2}^{u+1})$). Thereby we get

```
conj_{<u → <u+1}
=     ∀(u : ℕ).
          λg : (∀(s : ℕ) < u . (List(Boolean, [0]), [s]) → (Boolean, [0], [0]) , [0]).
                ∀(s : ℕ) < u + 1.
                    λx : (List(Boolean, [0]), [s]).
                          x match
                            {cons x_l x_t ⇒ if x_t {g x_t} {false}}
                            {nil (Boolean, [0]) ⇒ true}
```

Let $u \in \mathbb{N}$. Then $conj_{<u \,\rightarrow\, <u+1}$ expects a function g that can compute the conjunction of a list containing up to $u-1$ booleans. Further, it expects a boolean list of size $s < u$. In case the list is empty, it returns true. Otherwise it decomposes the list into a head x_h and a tail x_t of size $s-1$. In case $x_h =$ false, $conj_{<u \,\rightarrow\, <u+1}$ returns false without invoking g. Otherwise it applies g to the tail x_t and returns its result.

Note that since g has type $\forall(s : \mathbb{N}) < u\,.(\mathsf{List}(\mathsf{Boolean},[0]),[s]) \rightarrow (\mathsf{Boolean},[0],[0])$ and the tail x_t has size $s_t = s-1 < u$. Hence, our type system accepts the application $g\,x_t$.

Let $B_{conj} = (\mathsf{List}(\mathsf{Boolean},[0]),[s]) \rightarrow (\mathsf{Boolean},[0],[0])$ with free variable s. Then $conj_{<u \,\rightarrow\, <u+1}$ has the type $\forall(u:\mathbb{N})\,.(\forall(s:\mathbb{N}) < u\,.B_{conj},[0]) \rightarrow (\forall(s:\mathbb{N}) < u+1\,.B_{conj},[0],[0])$

In other words: For any upper bound u, $conj_{<u \,\rightarrow\, <u+1}$ expects a bounded function capable of handling lists of sizes below u. Then, $conj_{<u \,\rightarrow\, <u+1}$

extends this function to handle lists of sizes up to u.

We call such functions *bound-increasing*. In the rest oft this section we use them to build terminating recursive functions.

Definition 31 (Bound-Increasing Function). *Let $\Delta;\Gamma;\Lambda;P$ be a typing context, B a basic type and $\hat{s}$ a variable and l, u. Furthermore, let w be an arithmeric term with $\Lambda \vdash \forall(u:\mathbb{N}) \,.\, u <_0 w$ and let t be a placed term with*

$$\Delta;\Gamma;\Lambda;P \vdash t : (\forall(u:\mathbb{N})\,.\,\big((\forall(\hat{s}:\mathbb{N}) < u \,.\, B, [0]) \to (\forall(\hat{s}:\mathbb{N}) < w \,.\, B, [0], [0])\big), [0], [l]).$$

Then we call t a bound-increasing function.

According to the above definitions, bound-increasing functions preserve the representation of the basic function type B.

Let us modify the previously presented function $conj_{<u \,\to\, <u+1}$ to include some remote communication. The function $conj^{rem}_{<u \,\to\, <u+1}$defined below, has the same input-output behaviour as $conj_{<u \,\to\, <u+1}$ but additionally requests a remote value x'. Let $l_{x'}$ be the runtime-latency this remote request causes. Then we can define $conj^{rem}_{<u \,\to\, <u+1}$

```
=      ∀(u : ℕ).
           $\lambda g : (\forall(s:\mathbb{N}) < u \,.\, (\mathsf{List}(\mathsf{Boolean},[0]),[s]) \to (\mathsf{Boolean},[0],[s \cdot l_{x'}])\ ,\ [0])$.
                 $\forall(s:\mathbb{N}) < u+1$.
                     $\lambda x : (\mathsf{List}(\mathsf{Boolean},[0]),\ [s])$.
                         $x$ match
                             {cons $x_h\, x_t \Rightarrow$  let $y : T := \mathsf{get}\, p'.x'$ in
                                                  if $x_h \{g\, x_t\}$ {false}
                             }
                             {nil (Boolean, [0]) ⇒ true}
```

To ease the notation, let $\forall(s:\mathbb{N}) < u \,.\, B^{rem}_{conj}$ denote the type of argument g. That is $B^{rem}_{conj} = (\mathsf{List}(\mathsf{Boolean},[0]),[s]) \to (\mathsf{Boolean},[0],[s \cdot l_{x'}])$ with a free variable s. Then $conj^{rem}_{<u \,\to\, <u+1}$has the complex type $\forall(u:\mathbb{N})\,.\,(\forall(s:\mathbb{N}) < u \,.\, B^{rem}_{conj}, [0]) \to (\forall(s:\mathbb{N}) < u+1 \,.\, B^{rem}_{conj}, [0], [0])$. We see that increasing the bound u to $u+1$ preserves the function type B^{rem}_{conj}. In particular it preserves the arithmetic term $s \cdot l_{x'}$ expressing the runtime that depends on the size s of an input list.

Now consider the fixpoint application $\mathsf{fix}\, conj^{rem}_{<u \,\to\, <u+1}$ and the reduction rules presented in Section 2.4. Here, we supply $conj_{<u \,\to\, <u+1}$ as its own continuation. We can expand the application and thereby increase any starting bound indefinetely often. Hence, the result is a recursive function that can handle boolean lists of arbitrary size.

$$\frac{\Lambda \vdash \forall(u:\mathbb{N}) \,.\, u <_0 w \qquad \Delta;\Gamma;\Lambda;P \vdash f : (\forall(u:\mathbb{N}) \,.\, \big((\forall(s:\mathbb{N}) < u \,.\, B, [0]) \to (\forall(s:\mathbb{N}) < w \,.\, B, [0], [0])\big), [0], [l_f])}{\Delta;\Gamma;\Lambda;P \vdash \mathsf{fix}\, f : (\forall(s:\mathbb{N}) \,.\, B, [0], [l_f])} \quad (\text{T-FixApp})$$

Figure 2.14: Typing Rule for Size-Decreasing Recursive Functions

Consider an application $(\mathsf{fix}\, conj^{rem}_{<u \to <u+1})\, a$ to a boolean list a of size s_a. Suppose $s_a = 0$. Then application reduces to true without any recursive call and without causing any latency.

Now suppose $s_a > 0$. The list a is decomposed into a head x_h and a tail x_t of size $s_a - 1$. The following remote request $\mathsf{get}\, p'.x'$ causes a runtime latency of $l_{x'}$ before we proceed to inspect the tail. For $x_h = \mathsf{false}$ the function returns false. Otherwise it takes a recursive step on the tail and returns $(\mathsf{fix}\, conj^{rem}_{<u \to <u+1})\, x_t$.

From this analysis we can make two observations:

- **Preserved Latency Bound** We can bound the runtime latency of the application $(\mathsf{fix}\, conj^{rem}_{<u \to <u+1})\, a$ by $l_{x'} + (s_a - 1) \cdot l_x$ which is provably equivalent to $s_a \cdot l_{x'}$. Hence, the fixpoint operator preserves the latency bound $s \cdot l_x$.

- **Termination** Independent of the argument size s_a, the recursive step is always taken on a smaller argument a' of size $s'_a < s_a$. We call this principle *size-decreasing recursion.* Since sizes are finite we can prove that the application $(\mathsf{fix}\, conj^{rem}_{<u \to <u+1})\, a$ terminates.

These observations imply that it is safe to assign the type $(\forall(s:\mathbb{N}) \,.\, B^{rem}_{conj}, [0], [0]) = (\forall(s:\mathbb{N}) \,.\, (\mathsf{List}(\mathsf{Boolean}, [0]), [s]) \to (\mathsf{Boolean}, [0], [s \cdot l_{x'}]), [0], [0])$ to $\mathsf{fix}\, conj^{rem}_{<u \to <u+1}$.

The above observations do not depend on the concrete implementation of $conj_{<u \to <u+1}$ but on its type. Hence, we can generalize them to all bound-increasing functions. That is, we know that fixpoint applications $\mathsf{fix}\, f$ to bound-increasing functions f result in size-decreasing recursion with the above properties. Therefore, we can restrict λ^{lat} to allow only size-decreasing recursion by restricting the fixpoint operator to bound-increasing functions. This justifies the typing rule T-FixApp presented in Figure 2.14.

Let w be an arithmetic term and let $B^{\to} = \forall(u : \mathbb{N}) \,.\, ((\forall(s : \mathbb{N}) < u \,.\, B, [0]) \to (\forall(s : \mathbb{N}) < w \,.\, B, [0], [0]))$ be the basic type of a bound-increasing function. Consider the fixpoint application $\mathsf{fix}\, f$ and suppose we can assign the type $(B^{\to}, [0], [l_f])$ to f.

In order to apply T-FixApp to $\mathsf{fix}\, f$ we need to prove that f is indeed a bound-increasing function. This is expressed by the rule's premise $\Lambda \vdash \forall(u : \mathbb{N}) \,.\, u <_0 w$. Suppose the premise holds. Then T-FixApp assigns the type $(\forall(s : \mathbb{N}) \,.\, B, [0], [l_f])$ to $\mathsf{fix}\, f$.

2.5.8 Extending the Type System to $\lambda^{\mathrm{lat}'}$

$\lambda^{\mathrm{lat}'}$extends λ^{lat} by peer contexts. These are placed terms of the form $(\langle t\rangle_{\mathcal{I}}, [l])$. Such a context expresses that placed term t is to be reduced on the set of peer instances $\mathcal{I}$. The arithmetic term l represents the exact runtime latency tracked so far during the reduction that led to t.

Since $\lambda^{\mathrm{lat}'}$does only extend the set of placed terms we only have to extend our typing relation for placed terms $\vdash$. That is, we can apply our typing relation for placement terms $\Vdash$ to $\lambda^{\mathrm{lat}'}$without any changes.

Peer contexts are not part of λ^{lat}, the languae in which we formulate programs. They only occur as a result of reduction steps in the extended intermediate representation $\lambda^{\mathrm{lat}'}$. Consider a well-type λ^{lat} program q and let $(\langle t\rangle_{\mathcal{I}}, [l])$ be a peer context that is introduced during the reduction of q. According to the reduction rules we presented in Section 2.4, one of the following proposition holds:

- $\mathcal{I} = \mathcal{I}^P$ for some peer P (i.e., $\mathcal{I}$ contains all instances of peer type P)
- $\mathcal{I} = \{\ p\ \}$ for some peer instance p.

In both cases we can approximate $\mathcal{I}$ by a peer type P.

Since a peer context is a placed term, we type it with regard to some current peer. That is, the typing has the form $\Delta; \Gamma; \Lambda; P \vdash (\langle t\rangle_{\mathcal{I}}, [l_c]) : (B, [s], [l])$. Note that this typing involves two locations:

- P: The location on which the typing takes place.
- $\mathcal{I}$: The location on which t is to be reduced.

Hence, $(\langle t\rangle_{\mathcal{I}}, [l])$ can either be a local or a remote context. For each case we introduce a typing rule T-LocalReductionContext and T-RemoteReductionContext. Both are presented in Figure 2.15.

Local Reductiong Contexts Consider a typing context $\Delta; \Gamma; \Lambda; P$. Suppose $(\langle t\rangle_{\mathcal{I}}, [l_c])$ is a local context and that we can infer the typing $\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l_t])$. The peer context states that t is to be reduced on our current peer P. Meanwhile the typing states that t reduces to a value v of

$$\frac{\forall p \in \mathcal{I} : \mathcal{P}(p) = P \qquad \Delta;\Gamma;\Lambda;P \vdash t : (B,[s],[l_t])}{\Delta;\Gamma;\Lambda;P \vdash (\langle t\rangle_{\mathcal{I}},[l_c]) : (B,[s],[l_c + l_t])}$$
(T-LocalReductionContext)

$$\frac{\forall p' \in \mathcal{I}' : \mathcal{P}(p') = P' \qquad P \neq P' \qquad \Delta;\varnothing;\varnothing;P' \vdash t : (B,[s],[l_t])}{\Delta;\Gamma;\Lambda;P \vdash (\langle t\rangle_{\mathcal{I}'},[l_c]) : (\mathsf{Option}\,(B,[s]),[0],[l_c + l_t + \mathcal{L}(P',P)])}$$
(T-RemoteReductionContext)

Figure 2.15: Typing Rule Extension for $\lambda^{\text{lat}'}$

basic type B and size s. If the context is nested within some other peer context, we can reduce it to v, according to the reduction rules presented in Section 2.4. Otherwise $(\langle v\rangle_{\mathcal{I}},[...])$ is a value. Hence, we can assign it basic type B and size s.

l_t is an upper bound for the runtime latency the reduction of t causes. Hence, we can assign the peer context l_t as latency bound. However, as a result reduction would not preserve our extracted latency bounds. Consider the term $\mathsf{get}\,p.x$ which introduces remote communication. Suppose its type is $(\mathsf{Option}\,(\mathsf{Unit},[0]),[0],[2])$ and that we can reduce it to $\mathsf{some}\,\mathsf{unit}$. This reduction result has type $(\mathsf{Option}\,(\mathsf{Unit},[0]),[0],[0])$. That is, the latency bound decreased from 2 to 0 though the reduction step entailed a latency of 2. This is clearly not desirable. However, $\lambda^{\text{lat}'}$uses peer contexts to track runtime latency. Consider the example again with augmented peer contexts. The term $(\langle \mathsf{get}\,p.x\rangle_{\mathcal{I}},[0])$ reduces to $(\langle \mathsf{some}\,\mathsf{unit}\rangle_{\mathcal{I}},[2])$.

Therefore typing rule T-LocalReductionContext assigns the type $(B,[s],[l_c+l_t])$ to the context $(\langle t\rangle_{\mathcal{I}},[l_c])$. Thereby we preserve the extracted latency bound during reduction.

To ensure that we only apply T-LocalReductionContext to local peer contexts, it requires all peer instances contained in $\mathcal{I}$ to belong to our current peer P. This is expressed by the rule's premise $\forall p \in \mathcal{I} : \mathcal{P}(p) = P$.

Remote Peer Contexts $(\langle t\rangle_{\mathcal{I}'},[l_c])$ expresses that term t is to be reduced on the set of remote peer instances $\mathcal{I}'$. Hence, we have to type t on the remote peer P' which the peer instances in $\mathcal{I}'$ belong to. Suppose we can infer $\Delta;\varnothing;\varnothing;P' \vdash t : (B,[s],[l_t])$. This typing states that we can reduce t on P to a value v of basic type B and size s with a latency of at most l_t. Since, v lives on the remote peer P' we have to transfer it to our current peer P. We can do this by applying one of the reduction rules E-RemoteResultSuccess and E-RemoteResultFail. Both lead to a value of basic type $\mathsf{Option}\,(B,[s])$

and both introduce the additional runtime latency $\mathcal{L}(P', P)$.

Hence, typing rule T-RemoteReductionContext assigns the type $(\mathsf{Option}\,(B, [s]), [0], [l_c + l_t + \mathcal{L}(P', P)])$ to our term $(\langle t \rangle_{\mathcal{I}'}, [l_c])$.

In order to ensure that we apply the typing rule only to remote peer contexts, it contains the premises $\forall p' \in \mathcal{I}' : \mathcal{P}(p') = P'$ and $P \not\equiv P'$.

2.5.9 Typing Placement Terms

Over the previous sections we explained how we assign types to placed terms. The typing relation we presented considers a typing context $\Delta; \Gamma; \Lambda; P$ and a placed term t and infers a type $(B, [s], [l])$ for t.

A typing of the form $\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l])$ is a proof that placed term t reduces successfully on peer P to a value of basic type B and size s. By infering the type we extract an arithmetic term l. The typing $\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l])$ proves that l is an upper bound for the runtime latency caused during the reduction of t on P.

In the following we define a typing relation $\Vdash$ for placement terms. The purpose of placement terms is to place a computation on a specific peer and make its result available to other peers. Hence, our typing relation $\Vdash$ does not assign types to placement terms. Intead, a typing of the form $\ldots \Vdash q$ expresses that every computation placed by q on P reduces on P to a value of its expected type.

For the typing relation for placed terms we considered typing contexts of the form $\Delta; \Gamma; \Lambda; P$. Here, Δ and Γ are the typing environments for placed and local variables. Λ contains arithmetic assumptions regarding local size variables and P decribes a computation's location.

The life span of local variables is limited to a placed computation. Therefore, we do not consider Γ and Λ in our typing relation for placement terms. In constrast to placed terms, placement terms do not live on a specific peer but have a global view. Hence, do not consider the current peer P, either.

Accordingly, our typing context only consists of a typing environment for placed variables Δ as defined in Definition 26. The typing of a placement term q has the form $\Delta \Vdash q$. We call such terms *well-typed*. The typing rules for placement terms are presented in Figure 2.16

Placement terms can have the form end or the form $\mathsf{place}\, x : T := t\, \mathsf{on}\, P\, \mathsf{in}\, q$. While latter forms a sequence of placement terms, the former represents the end of such a sequence.

end does not place any computation and only works as an end marker. We can use typing rule T-PlaceEnd to type it. Independent of the typing context Δ, the term end is always well-typed.

$$\frac{}{\Delta \Vdash \mathsf{end}} \quad \text{(T-PlaceEnd)}$$

$$\frac{\begin{array}{ccc}\Lambda \Vvdash B_t \approx B_x & \Lambda \vdash s_t =_0 s_x & \Lambda \vdash l_t \leq_0 l_x \\ \multicolumn{3}{c}{\Delta; \varnothing; \varnothing; P \vdash t : (B_t, [s_t], [l_t]) \qquad \Delta, x \mapsto (B_x, [s_x])\,\mathsf{on}\,P \Vdash q}\end{array}}{\Delta \Vdash \mathsf{place}\,x : (B_x, [s_x], [l_x]) \coloneqq t\,\mathsf{on}\,P\,\mathsf{in}\,q} \quad \text{(T-Place)}$$

Figure 2.16: Typing Rules for Placement Terms

A Placement term of the form $\mathsf{place}\,x : (B_x, [s_x], [l_x]) \coloneqq t\,\mathsf{on}\,P\,\mathsf{in}\,q$ places the computation t on peer P. It binds the result to a variable x and makes the binding accessible in its scope q. We can use the typing rule T-Place to type it.

The above placement term places t on peer P. Hence, when we type t we have to consider this location. Furthermore, the typing context for t must not contain information from any previous local variables. Hence, we type it under consideration of the local context $\Delta; \varnothing; \varnothing; P$. Suppose our typing relation for placed terms infers the typing $\Delta; \varnothing; \varnothing; P \vdash t : (B_t, [s_t], [l_t])$. The type asciption $x : (B, [s], [l])$ expresses the expectation that t reduces to a value of basict type B and size s. Rule T-Place contains the premises $\Lambda \Vvdash B_t \approx B_x$ and $\Lambda \vdash s_t =_0 s_x$. These ensure that we can only apply the rule if B_t, B_x and s_t, s_x are provably equivalent.

Similar to let-terms, we allow explicit over-approximation in placement terms. Therefore, we do not require the inferred latency bound l_t and the latency ascription l_x to be equivalent. It suffices if we can prove that l_t does not exceed l_x. This is expressed by the rule's premise $\Lambda \vdash l_t \leq_0 l_x$.

When we type the nested placement term q, we have to consider the newly introduced variable x. This variable is bound to a value of basic type B_x and size s_x which lives on peer P. Hence, we type q under the extended context $\Delta, x \mapsto (B_x, [s_x])\,\mathsf{on}\,P$.

Suppose q is well-typed regarding this context. Then according to typing rule T-Place, the placement term $\mathsf{place}\,x : (B_x, [s_x], [l_x]) \coloneqq t\,\mathsf{on}\,P\,\mathsf{in}\,q$ is well-typed regarding the original context Δ.

2.6 Properties

Over the previous sections we defined our languages λ^{lat} and $\lambda^{\text{lat}'}$. The goal we pursue with λ^{lat} is to extract static bounds on a program's runtime latency. We used placement types to make a computation's location explicit in its type. Our type system uses this information to statically reason about the remote communication a computation entails and about the thereby caused latency. In the following we take a closer look at why this approach succeeds.

A crucial property of our language is its totality. That is, every well-typed computation terminates and successfully reduces to a value of its assigned type. This has two parts which we analyze separately:

- **Progress**
  Every well-typed term is either a value or we can reduce it further. Hence, well-typed computations do not get stuck during reduction.

- **Type Preservation**
  Reduction steps do not change the reduced term's type. This is necessary to guarantee that terms reduce to values of their assigned type.

We start by proving *Progress* in Section 2.6.1 before we prove *Preservation* in Section 2.6.2.

In Sections 2.4 and 2.5 we defined separate reduction and typing relations for placement terms and for placed terms. However, provided the named properties hold for placed terms, they are trivial for placement terms. Hence, we only investigate said properties regarding placed terms.

2.6.1 Progress

λ^{lat} is an extension of the typed λ-calculus for which progress and preservation hold. Hence, we have to analyze the extensions we added and their respective reduction rules.

In Section 2.4 we defined our reduction relation for the extended intermediate language $\lambda^{\text{lat}'}$. Hence, we refer to placed terms in $\lambda^{\text{lat}'}$throughout this analysis. All reduction steps consider terms wrapped in a peer context $(\langle t \rangle_{\mathcal{I}}, [l])$. For our analysis we consider a well-typed term $(\langle t \rangle_{\mathcal{I}}, [l])$ and show that it either is a value or that we can take a reduction step. The only typing rules we can apply to contexts of this form are T-LocalReductionContext and T-RemoteReductionContext. Both require that all peer instances $p \in \mathcal{I}$ belong to the same peer type. We only consider well-typed terms and therefore well-typed peer contexts. Hence, we can neglect all premises of the form $\forall p \in \mathcal{I} : \mathcal{P}(p) = P$ in our reduction rules. Furthermore, we also neglect the outermost peer context whenever it is not essential. That is, whenever we analyze a term t that is not a peer context, we implicitly consider a term of the form $(\langle t \rangle_{\mathcal{I}}, [l])$.

We can group the extensions we added to the type λ-calculus in five groups:

- branching terms, e.g., $\mathsf{if}\, t_c \,\{t_t\}\, \{t_f\}$
- remote communication, e.g., $\mathsf{get}\, p.t$
- size abstractions, e.g., $\forall(s : \mathbb{N}) \,.\, t$
- fixpoint application $\mathsf{fix}\, f$
- peer contexts $(\langle t \rangle_{\mathcal{I}}, [l])$

Branching Terms allow us to deconstruct values and depending on the result to reduce one of two branches. There are three forms in λ^{lat}: if-terms for $\mathsf{Boolean}$ values as well as match-terms for Option- and List values. Their reduction and typing rules are very similar. Hence, it suffices to analyze if-terms.

Consider a well-typed term $\mathsf{if}\, t_c \,\{t_t\}\, \{t_f\}$. The only typing rule we can use for this is T-If. The rule implies that t_c has basic type $\mathsf{Boolean}$. Suppose t_c is a value. Then we havae $t_c \in \{\mathsf{true}, \mathsf{false}\}$. Hence, we can either use reduction rule E-IfTrue or E-IfFalse to reduce it further. In case t_c is not a value, we can use reduction rule E-LocalContext to reduce it further. In any case we can take a reduction step on our if-term.

Remote Communication can have the form $\mathsf{get}\, p.t$, $\mathsf{remoteCall}\, p.t$ or the form $\mathsf{eval}\, t\, \mathsf{on}\, p$. The reduction rules for all three are very similar. Hence, we only consider $\mathsf{get}\, p.t$ and suppose that it is well-typed. Suppose p and t are values. Then we can apply reduction rule E-Get. Otherwise we can apply reduction rule E-LocalContext.

Size Abstractions can have the form $\forall(s : \mathbb{N}) \,.\, t$ and $\forall(s : \mathbb{N}) < u \,.\, t$. Since the reduction rules for both are very similar, we only consider the former one.

Suppose t is not a value. Then, we can apply reduction rule E-LocalContext to reduce the quantification further.

Suppose t is a value. If t contains the quantified variable s as a free arithmetic variable, the whole quantification $\forall(s : \mathbb{N}) \,.\, t$ is a value. In this case nothing is to do. Otherwise, we can eliminate the quantifier by applying reduction rule E-ForAllElim.

Fix Point Applications have the form $\mathsf{fix}\, f$. If f is a value, then so is $\mathsf{fix}\, f$. Otherwise we can reduce the term by applying rule E-LocalContext.

However, we still have to consider applications of the form $\mathsf{fix}\, f\, a$ since our reduction semantic treats them differently from normal function applications. If a is not a value we can reduce it with rule E-LocalContext, too

Hence, suppose both f and a are values and assume that $\mathsf{fix}\, f\, a$ is well-typed. We can only type it by applying the rules T-SizeDepLocalApp and T-FixApp. These rules imply that f is a bound-increasing function. Since it is also a value, this implies the form $f = \forall (u : \mathbb{N}) \,.\, \lambda x : (\forall (s : \mathbb{N}) < u \,.\, B, [0]).t$. Hence, we can apply reduction rule E-FixApp.

Peer Contexts have the form $(\langle t \rangle_{\mathcal{I}}, [l])$. Here, we assume that this is the outermost context our reduction relation considers.

If t is also a peer context, then the complete term represents an unfinished remote request. Hence we can either use E-RemoteContext to reduce the request or one of E-RemoteResultSuccess and E-RemoteResultFail to transmit the request's result.

Suppose that t is not a peer context. If it is a value, so is $(\langle t \rangle_{\mathcal{I}}, [l])$. Otherwise we can reduce t further.

We see that every well-typed term is either a value or we can reduce it further. Hence, we can prove the following lemma.

Theorem 1 (Progress). *Let Δ and Γ be typing environments for placed and local variables, Λ a set of arithmetic assumptions and P a peer type. Furthermore, let t be a placed term, $\mathcal{I} \subseteq \mathbb{I}$ a set of peer instances and T a type. Suppose $\Delta; \Gamma; \Lambda; P \vdash (\langle t \rangle_{\mathcal{I}}, [l]) : T$. and that $(\langle t \rangle_{\mathcal{I}}, [l])$ is not a value. Then there exists a reduction step*

$$(\langle t \rangle_{\mathcal{I}}, [l]) \overset{\mathcal{I}'}{\rightsquigarrow} \big(\langle t' \rangle_{\mathcal{I}}, [l']\big)$$

for a placed term t', a latency l' and a set of peer instances $\mathcal{I}' \subseteq \mathbb{I}$.

Proof. Proof by induction over the structure of placed term t. □

2.6.2 Preservation

In general type preservation means that reduction steps preserve the reduced term's type. In the standard typed λ-calculus it holds that $\vdash t : B$ and $t \rightsquigarrow t'$ imply $\vdash t' : B$. That is, a reduction step also preserves the type's representation B which is clear since all types in the standard λ-calculus have a unique representation. In λ^{lat}, however, we annotate basic types by arithmetic terms denoting sizes and latencies. These are not unique and so are our types' representations. Hence, we consider a relaxed notion of preservation.

Consider a placed term t with $\Delta;\Gamma;\Lambda;P \vdash t : (B,[s],[l])$ and a reduction step $(\langle t\rangle_{\mathcal{I}},[l]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}},[l'])$. We say that the reduction step preserves t's type $(B,[s],[l])$ if the reduction result t' has a type $(B',[s'],[l'])$ where B, B' and s, s' are provably equal and l' does not exceed l.

Definition 32 (Type Preservation). *Let Δ and Γ be typing environments for placed and local variables, Λ a set of arithmetic assumptions and P a peer type. Furthermore, let t, t' be placed terms, B a basic type, s a size, l, l_c, l'_c latencies and $\mathcal{I} \subseteq \mathbb{I}$ a set of peer instances. Moreover, let $\Delta;\Gamma;\Lambda;P \vdash (\langle t\rangle_{\mathcal{I}},[l_c]) : (B,[s],[l])$.*

A reduction step $(\langle t\rangle_{\mathcal{I}},[l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t'\rangle_{\mathcal{I}},[l'_c])$ is type-preserving *if there exists a basic type B', a size s' and a latency l' with*

$$\begin{aligned}
&\Delta;\Gamma;\Lambda;P \vdash \big(\langle t'\rangle_{\mathcal{I}},[l'_c]\big) : (B',[s'],[l'])\\
&\Lambda \Vdash B \approx B'\\
&\Lambda \vdash s =_0 s'\\
&\Lambda \vdash l' \leq_0 l
\end{aligned}$$

Most reduction steps clearly preserve the type. However, there are four groups of reduction rules we have to consider:

- reduction rules for branching terms, e.g., E-IfTrue and E-IfFalse.
- function application, e.g., E-LocalApp
- quantifier elimination, e.g.,, E-ForAllElim
- recursive function application, i.e., E-FixApp

Branching Terms As mentioned in the previous section, the reduction rules for the different kinds of branching terms are quite similar. Hence, we only consider if-terms. Furthermore, reduction rule E-LocalContext clearly preserves the type, provided that the other reduction rules do.

Let $\Delta;\Gamma;\Lambda;P \vdash \mathsf{if}\, v_c\,\{t_t\}\,\{t_f\} : (B,[s],[l])$ and suppose v_c is a value. The only typing rule we can apply is T-If. The only reduction rules we can apply are E-IfTrue and E-IfFalse which reduce the term either to t_t or t_f. Typing rule T-If ensures that both have provably equal basic types and sizes and that $\mathsf{if}\, v_c\, \{t_t\}\, \{t_f\}$ has the same basic type and size as t_t. That is, $\Delta; \Gamma; \Lambda; P \vdash t_t : (B, [s], [l_t])$ and $\Delta; \Gamma; \Lambda; P \vdash t_f : (B_t, [s_f], [l_f])$ with $\Lambda \Vdash B \approx B_f$ and $\Lambda \vdash s =_0 s_f$. Furthermore, together with these typings and with rule T-If we get $l = ... + \max\big(l_t, l_f\big)$. Our reduction rules lead to a term with one of the two types $(B, [s], [l_t])$ and $(B_f, [s_f], [l_f])$. We can prove $\Lambda \vdash l_t \leq_0 \max\big(l_t, l_f\big)$ and $\Lambda \vdash l_f \leq_0 \max\big(l_t, l_f\big)$. Hence, in any case our reduction rules preserve the type of $\mathsf{if}\, t_c\, \{t_t\}\, \{t_f\}$. Analogous results hold regarding match-terms for List and Option.

In general we cannot statically decide which branch of an if- or match-term is considered during reduction. Our typing rules extract a static upper bound on a branching term's runtime latency by considering both branches' latency and taking their maximum. Consider a term r of type $(\mathsf{Unit}, [0], [2])$ and the function $\lambda x : (\mathsf{Boolean}, [0]).\mathsf{if}\, x\, \{r\}\, \{\mathsf{false}\}$. If we supply true for x we cause a runtime latency up to 2. If we supply false we do not cause any latency at all. When we reduce $\mathsf{if}\, x\, \{r\}\, \{\mathsf{false}\}$ to false, we gain more information. Meanwhile, our type-level latency bound decreases from $\max(2, 0)$ to 0. This improvement is not a mistake and justifies our relaxed definition of type preservation above.

Function Applications have the forms $f\, a$ and $\mathsf{remoteCall}\, p.f\, a$. In regard to their type preserving behaviour local and remote application are analogous. Hence, it suffices to consider the local application $f\, a$.

If one of f and a is not a value, we can apply reduction rule E-LocalContext which preserves the application's type. Suppose f and a both are values. Then f has the form $f = \overline{Q}[\lambda x : (B, [s]).t]$ where $\overline{Q}$ is a syntactic quantifier context. If f is a normal function this context is empty and f simply is a λ-abstraction. However, if f is an unbounded size-dependent function, the context is an unrestricted quantifier $\forall(s : \mathbb{N})$. Similarly, if f is a bounded size-dependent function, the quantifier is a restricted one.

In any case, the only reduction rule we can apply is E-LocalApp which reduces our term to $\overline{Q}[t[x \mapsto a]]$. In order to analyze whether this reduction preserves the application's type, we must consider the concrete typing rule we used to type $f\, a$, i.e., T-LocalApp, T-SizeDepLocalApp or T-BoundedSizeDepLocalApp. This case distinction and a following detailed analysis is quite tedious. Hence, we omit it here. However, to ensure correctness we give it in the appendix as part of the proof for our type preservation theorem (Theorem 2). Furthermore, we also have to consider whether f results from a fixpoint application $\mathsf{fix}\, f'$. We analyze this case later and for now assume that f is not a recursive function.

In the standard λ-calculus, a function application's reduction preserves the application's type. λ^{lat} adds size-dependent functions whose typing depends on a quantified size variable. Consider a size-dependent application $(\forall(s:\mathbb{N}).\lambda x':(B',[s]).t')\,a'$. The quantified size variable s is bound in t' and it is possible that the substituted term $t'[x' \mapsto a']$ contains s as a free arithmetic variable. In this case the substitution result is not well-typed unless we add a quantification for s. The crucial detail to note here is that our reduction step preserves the quantifier context $\overline{Q}$. Hence, our reduction step preserves quantified size variables. Therefore, our reduction step also preserves the application's type.

Quantifier Elimination The reduction and typing rules for unrestricted and restricted quantications are analogous. Hence, we only consider unrestricted ones. Let $\forall(s:\mathbb{N}).t$ be well-typed.

Suppose that t does contain s as a free arithmetic variable. Then we cannot eliminate the quantifier $\forall(s:\mathbb{N})$. If t is a value, the whole quantification $\forall(s:\mathbb{N}).t$ is one. Otherwise we can apply E-LocalContext and t which preserves the quantification's type (if all other rules do).

Suppose term t does not contain s as a free arithmetic variable. Then we do not have to consider s when we type t. Hence, we can apply rule T-ForAllIgnore to type the term $\forall(s:\mathbb{N}).t$. Suppose $\Delta;\Gamma;\Lambda;P \vdash t : T$. Then we also get $\Delta;\Gamma;\Lambda;P \vdash \forall(s:\mathbb{N}).t : T$. Since s is not a free arithmetic variable of t, we can apply reduction rule E-ForAllElim. This reduces $\forall(s:\mathbb{N}).t$ to t. By assumption both terms have the same type. Hence, the reduction step preserves the type.

Fixpoint application has the form $\mathsf{fix}\,f$ for a bound-increasing function f. If f is a value, so is the application $\mathsf{fix}\,f$. Otherwise we can reduce it by rule E-FixApp which preserves the application's type.

However, we still have to consider application of the size-dependent recursive function $\mathsf{fix}\,f$ to some argument a. Our reduction rules handle such applications different from non-recursive function applications.

Let $\mathsf{fix}\,f\,a$ be well-typed. If one of f or a is not a value we can reduce it in a type preserving way by E-LocalContext. Suppose f and a are values. The only typing rule we can apply to $\mathsf{fix}\,f\,a$ is T-FixApp. Hence, f is a bound-increasing function and we get the typing $\Delta;\Gamma;\Lambda;P \vdash f : (\forall(u:\mathbb{N}).\big((\forall(s:\mathbb{N}) < u.B^{\rightarrow},[0]) \rightarrow (\forall(s:\mathbb{N}) < w.B^{\rightarrow},[0],[0])\big),[0],[l_f])$ as well as a proof $\Lambda \vdash \forall(u:\mathbb{N}).u <_0 w$. Since f is a value its type implies the form $f = \forall(u:\mathbb{N}).\lambda x:(\forall(s:\mathbb{N}) < u.B^{\rightarrow},[0]).t$. Furthermore, for the fixpoint application we get the typing $\Delta;\Gamma;\Lambda;P \vdash \mathsf{fix}\, f : (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0], [l_f])$. Our typing rules do not allow a direct application $f\,(\mathsf{fix}\, f)$. In Section 2.4.3.3 we presented a function BE (cf. Def23) which we use to eliminate the upper bound w in our function body t. The type of f implies that $\Delta;\Gamma, u \mapsto \mathbb{N}, x \mapsto (\forall(s : \mathbb{N}) < u \,.\, B^{\rightarrow}, [0]);\Lambda;P \vdash t : (\forall(s : \mathbb{N}) < w \,.\, B^{\rightarrow}, [0], [l_f])$.

This typing implies $\Delta;\Gamma, u \mapsto \mathbb{N}, x \mapsto (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0]);\Lambda;P \vdash t : (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0], [l_f])$. According to the definition of BE and our typing rules presented in Section 2.5, we get $\Delta;\Gamma;\Lambda;P \vdash \forall(u : \mathbb{N}) \,.\, \lambda x : (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0]).BE(t) : ((\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0]) \rightarrow (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0], [0]), [0], [l_f])$.

Note that we need to keep the universal quantifier $\forall(u : \mathbb{N})$ surrounding our λ-abstraction. The modified body $BE(t)$ might contain u as a free variable (though not in a position that is reflected in its type). We can even proof the following more general Lemma 1.

Let $f' = \forall(u : \mathbb{N}) \,.\, \lambda x : (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0]).BE(t)$. Then our type system accepts the application $f'\,(\mathsf{fix}\, f)$. In fact, according to typing rule T-SizeDepLocalApp we get $\Delta;\Gamma;\Lambda;P \vdash f'\,(\mathsf{fix}\, f) : (\forall(s : \mathbb{N}) \,.\, B^{\rightarrow}, [0], [l_f])$. Hence, the expanded application $f'\,(\mathsf{fix}\, f)\, a$ has the same type as our original recursive function application $\mathsf{fix}\, f\, a$.

The only reduction rule we can apply to $\mathsf{fix}\, f\, a$ is E-FixApp. It reduces the application to $f'\,(\mathsf{fix}\, f)\, a$. As shown above, this reduction step preserves the application's type.

Lemma 1 (Type Bound Elimination). *Let t be a placed term, B a basic type, $\hat{x}$ a variable, u an arithmetic term, s a size and l a latency. Let Δ and Γ be typing environments for placed and local variables, respectively. Furthermore, let Λ be set of arithmetic assumptions and P be a peer type. Suppose*

$$\Delta;\Gamma;\Lambda;P \vdash t : (\forall(\hat{x} : \mathbb{N}) < u \,.\, B, [s], [l]).$$

Then

$$\begin{aligned}
&\Delta;\Gamma;\Lambda;P \vdash BE(t) : (\forall(\hat{x}' : \mathbb{N}) \,.\, B', [s'], [l'])\\
&\Lambda \Vdash \forall(\hat{x} : \mathbb{N}) \,.\, B \approx \forall(\hat{x}' : \mathbb{N}) \,.\, B'\\
&\Lambda \vdash s =_0 s'\\
&\Lambda \vdash l =_0 l'.
\end{aligned}$$

Particularly, for $s = 0$ we get $s' = 0$ and for $l = 0$ we get $l' = 0$.

Proof. Proof by induction over structure of t. □

We can prove the following theorem which states that reduction in λ^{lat} and $\lambda^{\text{lat}'}$ preserves the reduced term's type.

Theorem 2 (Type Preservation)**.** *Let Δ and Γ be typing environments for placed and local variables, Λ a set of arithmetic assumptions and P a peer type. Furthermore, let t, t' be placed terms, l_c, l'_c latencies, T a type and $\mathcal{I} \subseteq \mathcal{I}^P$ a set of peer instances of type P. Assume that*

$$\Delta; \Gamma; \Lambda; P \vdash t : T$$
$$(\langle t \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle t' \rangle_{\mathcal{I}}, [l'_c]\big)$$

Then there exists a type T' such that

$$\Delta; \Gamma; \Lambda; P \vdash t' : T'$$
$$\Lambda \Vdash T \mathrel{\hat{\approx}} T'$$

Proof. Proof by induction over the structure of placed term t (cf. appendix). □

Consider a typing $\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l])$. *Progress* and *Preservation* together imply that we can reduce t further until we reach a value of type $(B', [s'], [l'])$. According to our preservation theorem it holds $\Lambda \Vdash B \approx B'$, $\Lambda \vdash s =_0 s'$ and $\Lambda \vdash l' \leq_0 l$. Furthermore, the definition of our reduction relation implies that this value has the form $(\langle v \rangle_{\mathcal{I}}, [l_v])$ where v is a value and not a peer context. Hence, v has a latency of 0 and the only typing rule we can apply to $(\langle v \rangle_{\mathcal{I}}, [l_v])$ is T-LocalReductionContext. Thereby, we get $\Delta; \Gamma; \Lambda; P \vdash v : (B', [s'], [0])$ and $l_v = l'$. That is, our reduction of t with type $(B, [s], [l])$ results in a value $(\langle v \rangle_{\mathcal{I}}, [l'])$ with $\Lambda \vdash l' \leq_0 l$. This proves that our type system extracts correct static latency bounds.

To make our observation precise we formulate the following corollary.

Corollary 1 (Correctness of Static Latency Bounds)**.** *Let Δ and Γ be typing environments for placed and local variables, Λ a set of arithmetic assumptions and P a peer type. Furthermore, let t be a placed term, B a basic type, s a size, l a latency. Moreover let $\Delta; \Gamma; \Lambda; P \vdash t : (B, [s], [l_B])$.*

Then there exists a finite reduction sequence towards a value $(\langle v \rangle_{\mathcal{I}}, [l_v])$ which cannot be reduced any further. It holds that $\Lambda \vdash l_v \leq_0 l_B$.

Chapter 3

Scala$^{\text{Lat}}$: A Prototype Implementation of λ^{lat}

In this chapter we present a prototype implementation Scala$^{\text{Lat}}$ for the language design proposed in Chapter 2. Scala$^{\text{Lat}}$ is a Scala DSL and a simplification of λ^{lat} where we do not consider latency weights $\mathcal{L}(P, P')$ but instead track the number of sent remote messages.

In Section 3.1 we explain how we represent runtime and type-level locations in Scala$^{\text{Lat}}$. In Section 3.2 we describe how Scala$^{\text{Lat}}$ makes a computation's location and the latency it causes explicit in the computation's type. Similar to λ^{lat} we use arithmetic type-level computations to represent type-level sizes and latency bounds. In Section 3.3 we define arithmetic type-level operations. Furthermore, we develop a proof system that allows the programmer to statically reason about arithmetic type-level computations.

In Section 3.4 we describe the primitives Scala$^{\text{Lat}}$ offers to express remote communication. Their type signatures make the involved locations as well as the arising latency transparent to the programmer. Afterwards, in Section 3.5 we use sized types to present two approaches to express size-decreasing recursion in Scala$^{\text{Lat}}$ and to extract corresponding latency bounds. In particular, we present a fixpoint operator and describe how recursive function applications expressed via this operator are statically unrolled. This allows us to guarantee termination of fixpoint applications.

3.1 Locations

Anagous to λ^{lat}, our implementation uses peer types to represent type-level locations and instances of these types for runtime locations.

```
trait Peer[Self <: Peer[Self]] {
    this: Self =>

    def get ...

    def remoteCall ...

    def placedCall ...
}

trait Server extends Peer[Server] {
    val data = ...
}

class Client (server: Server) extends Peer[Cient] {
    val remoteData = get(server.data)
}
```

Figure 3.1: Runtime and Type-Level Locations

Our implementation offers a trait Peer that implements methods get, remoteCall, placedCall to perform remote computations. We implement type-level locations by types derived from Peer. Runtime locations are instantiated peer types. That is, while λ^{lat} assumes fixed sets of peer types and peer instances, our implementation allows the programmer to define them herself.

Figure 3.1 presents the definition of two peer types Server and Client. While Server is implemented as trait, Client is defined as a class. Both extend Peer and thereby inherit its methods get, remoteCall and placedCall. The server contains some local data data and the client stores a reference to a server instance server. Since Client inherits from Peer, it can use the server instance to access the placed data via get(server.data)

The client uses the inherited method get to request data placed on Server from peer instance server, i.e., get(server.data). This is analogous to a remote request $\mathsf{get}\, p.x$ in λ^{lat}.

3.2 Type Annotations

3.2.1 Latency Types

In λ^{lat} types have the form $(B, [s], [l])$ where B is a basic type, s a size and l a latency annotation. However, sizes only matter for lists since all values of all other basic types have size 0. This uniform structure simplifies analyses

```
class HasLatency[+C, L <: Nat](_comp: => C) {
    protected[latencyAnnotation] def comp: C = _comp

    def map[B](f: C => B): HasLatency[B, L]
        = new HasLatency(f(comp))

    def flatMap[B, Lb <: Nat](f: C => HasLatency[B, Lb])
        : HasLatency[B, L# + [Lb]]
        = new HasLatency(f(comp).comp)

        def rewriteLatency[L2 <: Nat]
            (implicit lEqL2: EqualityNat[L, L2])
            : HasLatency[C, L2]
                = new HasLatency(comp)
}


object HasLatency {
    def apply[C](localComp: => C): HasLatency[C, _0] =
        new HasLatency(localComp)
}
```

Figure 3.2: Latency Monad

of the language. For instance, it reduces the number of case distinctions in proofs by induction over the type structure.

In our implementation usability is more important than simplifying induction proofs. Hence, we only annotate sizes were necessary which allows the programmer to focus on using the type system to extract latency bounds.

In Scala$^{\text{Lat}}$ we express latency annotations by wrapping expressions into an annotated monad HasLatency. Figure 3.2 shows its definition. The monad expects an expression comp of type C and a type-level computation L expressing the computation's latency. An instantiation HasLatency[C, L](comp) does not execute the computation comp. Arithmetic type-level computations are subtypes of Nat which we explain in Section 3.3.

Tracking Latency Latency-annotated expressions cannot be accessed directly. Thereby, we prevent the programmer from accidentally circumventing our latency tracking mechanism. Instead the annotated expression can be accessed via the monad's map and flatMap methods. Both make latency changes explicit in the result's type. Furthermore, both lead to an evaluation of the annotated expression. The implementation of map is standard.

Consider an annotated expression `i: HasLatency[Int, L]`. We can double the integer by `i.map(x =>x*x)`. The wrapped expression is only evaluated once before it is bound to variable `x`. Hence, the latency did not change and the result has type `HasLatency[Int, L]`. However, `flatMap` is not standard. It does not only eliminate a nested `HasLatency` but also statically sums up the latency annotations.

Let `Lc <: Nat`, `Lb <: Nat` be two type-level latency values. Furthermore, consider an annotated expression `c : HasLatency[C, Lc]` and a function `f : C =>HasLatency[B, Lb]`. An application `c.map(f)` produces a result of type `HasLatency[HasLatency[C, Lc], Lb]`. In constrast, `flatMap` returns a value of type `HasLatency[C, Lc#+ [Lb]]`. Here, `L1#+ [L2]` is a type-level computation as we later see in Section 3.3. That is, the result type of `c.map(f)` represents the additional latency `Lb` introduced by the application of `f`.

Latency Representations In λ^{lat} we treat arithmetic terms as representatives of equivalence classes. Consider two syntactically different latency annotations l_1, l_2 in λ^{lat} for which we can prove $l_1 = l_2$. Then, our type systems treats any two types $(B, [s], [l_1])$, $(B, [s], [l_2])$ as equal. Now consider Scala type-level computations `L1 <: Nat`, `L2 <: Nat` corresponding to l_1, l_2, respectively. Scala's type system considers the types `HasLatency[C, L1]` and `HasLatency[C, L2]` to be different. In practice the distinction between different representations of the same latency renders our type system quite useless. To overcome this restriction, we allow to replace latency annotations by provably equal annotations. The `HasLatency`-monad therefore offers a method `rewriteLatency`.

In Section 3.3 we explain how we prove properties concerning arithmetic type-level computations. For now, suppose we have an equality proof `l1EqL2 : EqualityNat[L1, L2]` which proves that `L1` and `L2` are equal and let `c : HasLatency[C, L1]`. The application `c.rewriteLatency(l1EqL2)` rewrites the annotation into `HasLatency[C, L2]`. The `rewriteLatency` proof argument is implicit. Hence, Scala$^{\text{Lat}}$ is able to construct simple proofs like `EqualityNat[_3#+ [_5], _8]` or `EqualityNat[L1#+ [L2], L2#+ [L1]]`. However, complicated proofs have to be created and supplied by the programmer.

3.2.2 Placement Types

In Scala$^{\text{Lat}}$ we use placement types to track the location of data and computations in their type. As in λ^{lat} this allows us to statically extract latency bounds from computations that involve messages sent between remote peers. Our implementation is a simplification of λ^{lat} that only focuses on the ex-

```
class Placement[C, P <: Peer[_], L <: Nat](_comp: => C, location: P)
    extends HasLatency[C, L](_comp) {

    def evalAndRelocate[Pt <: Peer[_]](target: Pt)
        : Placement[C, Pt, L# + [_1]]
            = ...

    def requestEvalAndRetrieval[Pt <: Peer[_]](target: Pt)
        : Placement[C, Pt, L# + [_2]] = ...

    override def map[B](f: C => B): Placement[B, P, L] = ...

    override def flatMap[B, Lb <: Nat](f: C => HasLatency[B, Lb])
        : Placement[B, P, L# +[Lb]] = ...
}

object Placement {
    def apply[C, P <: Peer[_]](localComp: => C, location: P)
        : Placement[C, P, _0]
            = ...
}
```

Figure 3.3: Signature of the Placement Monad

traction of latency bounds. For instance, Scala$^{\text{Lat}}$ does not consider ties and assumes that all peers are connected if they have the possibility to reference an instance of each other. Furthermore, we do not consider weighted connections. Instead, our implementation tracks the number of remote messages a computation involves.

Since we only use placement types as a means to extract latency bounds, we implement them as a refinement of latency types. That is, a placement type is a monad Placement that extends the latency monad HasLatency and features an additional peer annotation.

Like HasLatency, the placement monad expects an expression of type C and a type-level computation L <: Nat representing a latency annotation. It further expects a runtime location, that is, an instance of some peer type P <: Peer[_]. Then, a placement type has the form Placement[C, P, L]. Figure 3.3 presents its signature.

Placement inherits and refines the map and flatMap from HasLatency. As with the latency monad we forbid direct access to the annotated expression. This prevents the programmer from accidentally circumventing the location and latency tracking mechanism. Placement offers two additional methods `evalAndRelocate` and `requestEvalAndRetrieval`. Both methods model different types of remote communication primitives but do not actually perform any network communication. Their type signatures, however, allow us to track placement and latency changes in our type system. As we see later in Section 3.4 we can use these to implement the remote communication methods `get`, `remoteCall`, `placedCall` in our base trait `Peer`.

Remote Communication Consider a peer type `P <: Peer[_]`, a latency annotation `L <: Nat` and placed expression `c: Placement[C, P, L]`. That is, we place an expression on an instance `location` of peer `P` and its evaluation on `P` involves up to `L` remote messages. Furthermore, let `Pt <: Peer[_]` be a remote peer and `target: Pt` a remote peer instance.

A call `c.evalAndRelocate(target)` models the wrapped expression's evaluation on `location` and the result's transmission from `location` to `target`. Speaking on the type level, the evaluation takes place on peer `P` and the result is sent to remote peer `Pt`. Hence, the result is located on `Pt`. Since we do not consider weighted connections as in λ^{lat} but only track the number of remote messages, this increases the latency by `1`. Therefore, the result type of `c.evalAndRelocate(target)` is `Placement[C, Pt, L#+ [_1]]`.

A call `c.requestEvalAndRetrieval(target)` models a request sent by remote peer instance `target` requesting the wrapped expression's result. Hence, its invokation involves (i) the request sent from `target` to `location`, (ii) the expression's evaluation on `location` and (iii) the result's transmission from `location` to `target`. This involves two remote messages for sending the request and transmitting the result as well as up to `L` messages sent during the evaluation on `location`. Hence, the result type is `Placement[C, Pt, L#+ [_2]]`.

3.3 Arithmetic

In λ^{lat} we use arithmetic terms to specify arithmetic type-level computations and Heyting Arithmetic as an arithmetic theory to prove properties about them. This allows us to abstract over different representations of the same natural number like $S\,S\,0$ and $S\,0 + S\,0$. In λ^{lat} we view arithmetic terms as representatives of the equivalence class of provably equal terms. For instance, $S\,S\,0$ and $S\,0 + S\,0$ belong to the same equivalence class [2].

Let h_1, h_2 be arithmetic terms for which we can prove in Heyting Arithmetic that $h_1 = h_2$. Then our type system considers the types $(B, [h_1], [l])$, $(B, [h_2], [l])$ and $(B, [s], [h_1])$, $(B, [s], [h_2])$ to be equal.

We follow this approach in Scala$^{\text{Lat}}$. That is, (i) we represent natural numbers as Heyting numbers (ii) implement basic arithmetic type-level operators like + and (iii) define intuitionistic predicates to reason about them.

Type-Level Natural Numbers We implement natural numbers and arithmetic type-level computations as phantom types. That is, as types that cannot be instantiated and hence do not introduce any runtime overhead. Figure 3.4 presents our implementation.

Natural numbers are subtraits of the sealed base trait Nat. In Heyting Arithmetic we build natural numbers using the constant 0 and a successor function S. In general, we represent type-level computations in Scala$^{\text{Lat}}$ by sealed traits and model arguments as type parameters. Therefore, we derive two sealed subtraits trait Zero extens Nat and trait Suc[N <:Nat] extends Nat. Note that the successor function $S\,\cdot$ expects a natural number as argument and so does our trait Suc[N <:Nat]. The upper type-level bound <: Nat ensures that Scala's type system only accepts applications of Suc to natural numbers. This approach allows us to define the domain of type-level functions.

Analogous to Heyting Arithmetic we can represent natural numbers as Zero, Suc[Zero], Suc[Suc[Zero]] and so on. To reduce the syntactic overhead, we define static abbreviations _0, _1, _2, ... for small natural numbers Zero, Suc[Zero], Suc[Suc[Zero]], ...

Arithmetic Type-Level Computations In Heyting Arithmetic we define arithmetic operators by an axiomatic description. For instance, we can define addition by the two axioms $0 + y = y$ and $(S\,x) + y = S\,(x + y)$. In Scala, there are two possibilities how we can model arithmetic computations. (i) We can define separate traits for every operator. For instance, Sum[A <:Nat, B <:Nat] to express addition. (ii) We can also express them in terms of our traits Zero and Suc. In case we chose, approach (i), computations and their intended result would not be related. For instance, the addition Sum[_3, _2] and its intended result _5 are different types. Hence, we chose approach (ii) and map the addition of _3 and _2 = Suc[Suc[Zero]] to _5 = Suc[Suc[_3]] .

We declare arithmetic operators as abstract type members of our base trait Nat and implement them in the subtraits Zero and Suc. Consider an addition $x + y$ and its defining axioms from above. If $x = 0$ the result is the second argument y. Otherwise x has the form $x = S\,x'$ and the result is the successor of $x'+y$. We can view these axioms as pattern matching on the left argument and we follow this perspective throughout our implementation.

As Figure 3.4 shows, our base trait Nat contains an abstract type member

```
sealed trait Nat {
  type Match[SucBranch <: Up, ZeroBranch <: Up, Up] <: Up
  type Pred <: Nat
  type +[B <: Nat] <: Nat
  type -[B <: Nat] <: Nat
  type Max[B <: Nat] <: Nat
  type *[B <: Nat] <: Nat
}

sealed trait Zero extends Nat {
  type Match[SucBranch <: Up, ZeroBranch <: Up, Up] = ZeroBranch
  type Pred = Zero
  type +[B <: Nat] = B
  type -[B <: Nat] = Zero
  type Max[B <: Nat] = B
  type *[B <: Nat] = Zero
}

sealed trait Suc[N <: Nat] extends Nat {
  type Match[SucBranch <: Up, ZeroBranch <: Up, Up] = SucBranch
  type Pred = N
  type +[B <: Nat] = Suc[N# +[B]]
  type -[B <: Nat] = B#Match[N# -[B#Pred], Suc[N], Nat]

  type Max[B <: Nat] = (this.type # -[B])# + [B]

  type *[B <: Nat] = (N# * [B])# + [B]
}


object Nat {
  type _0 = Zero
  type _1 = Suc[_0]
  ...
}
```

Figure 3.4: Natural Numbers and Type-Level Computations

type +[B <:Nat] <: Nat. Here, + is a type-level function that expects a natural number B <:Nat and returns a natural number. Let N <:Nat, M <:Nat be natural numbers. Then we can express their sum by N#+ [M]. This accesses the definition of type +[B <:Nat] in N. The axioms for addition we presented above pattern matched on the left argument. We follow these axioms and define the computation in Zero as type +[B <:Nat] =B and in Suc[N <:Nat] as type +[B <:Nat] =Suc[N + [B]]. Note that this complies with the presented axioms since _0#+ [Y] = Y and Suc[X]#+ [Y] = Suc[X#+ Y].

We implement the other basic arithmetic operators in a similar way. Figure 3.4 shows their implementations. Note that according to this definition the literal _5 and the addition _3#+ [_2] are the same type. Hence, we do not need an external equality proof to identify those. Consider a method f : HasLatency[A, _5] =>Z and an argument a : HasLatency[A, 3#+ [_2]]. Since _5 and 3#+ [_2] are the same type, Scala's type system accepts the application f(a).

However, this is only possible since the left argument is a concrete type. Consider two type variables X <:Nat, Y <:Nat as well as a method g : HasLatency[A, X#+ [Y]] =>Z and an argument a : HasLatency[A, Y#+ [X]]. From the implementation of + in Zero and Suc it follows that, independent of the concrete choice of X and Y, both X#+ [Y] and Y#+ [X] evaluate to the same result. However, Scala's type system cannot prove this since it only knows that X and Y are subtypes of Nat which does not contain an implementation. We solve this by defining additional logical predicates and corresponding axioms. These allow the programmer to prove propositions regarding arithmetic type-level computations, like the equality of X#+ [Y] and Y#+ [X].

Propositions and Proofs The Curry-Howard Correspondence states that we can view logical propositions as types and proofs as programs [8]. We follow this perspective to build a simple proof system that is sufficient to proof arithmetic properties regarding our type-level computations.

As mentioned above, we want to prove that certain type-level computations like X#+ [Y] and Y#+ [X] yield the same result. Hence, we define a sealed trait Equality[A <:Nat, B <:Nat] to represent the proposition. We accept any instance of Equality[X,Y] for natural numbers X, Y as a proof that A equals B. Sealing the trait prohibits any instantiations outside of the file containing the trait's definition. This allows us to control the construction of Equality proofs. Note that without this restriction the programmer could accidentally proof anything. For instance, new Equality[_0, _1]{}.

Furthermore, we can view the trait Equality without its type parameters as a relation between natural numbers. We can axiomatize this relation by defining static constants and methods to instatiate Equality within its source file. Throughout our implementation, we use the relation's companion object to store its axioms. Since Equality should be an equality relation, we must include the corresponding axioms: (i) reflexivity, symmetry, transitivity. Furthermore, we also want to be able to reason about type-level computations like A#+ [B]. Therefore, we have to include axioms for every relevant operator. Figure 3.5 presents a simplified definition of Equality that only includes (i) general equality relation axioms, (ii) axioms regarding the successor trait Suc[N <:Nat] and (iii) axioms regarding addition.

Reflexivity Every natural number equals itself. Hence, for any A <:Nat we should be able to instantiate Equality[A, A]. We can express this axiom by a constant polymorphic method that takes A as a type parameter and instantiates Equality[A, A]. When we develop programs that involve rewriting latency bounds, it is inconvenient to write every proof by hand. Defining our axioms as implicit allows us to leverage the compiler to construct simple proofs. Hence, we get:

```
implicit def reflexity[A <: Nat]: Equality[A, A]
        = new Equality[A, A] {}
```

Symmetry states that whenever A <:Nat is equal to B <:Nat, the opposite direction holds, too. In general, we can view any implication $P_1 \to P_2$ as procedure that transforms a proof for proposition P_1 into one for P_2. Hence, we implement our symmetry axiom as a method that takes an instance of Equality[A, B] and returns an instance of Equality[B, A]. Therefore, our implementation has the form:

```
implicit def symmetry[A <: Nat, B <: Nat]
    (implicit aEqB: Equality[A, B]): Equality[B, A]
        = new Equality[B, A] {}
```

The other axioms' definitions are analogous.

Resolution Presedence When we request a proof the compiler considers all implicit definitions that are in scope. A common problem is that the resolution diverges. We can, however, give the compiler hints about the precedence of our axioms by moving ones with a lower precedence further up in the scope's inheritance hierarchy. Consider the structure presented in Figure 3.5. We distributed our axioms over the companion object and two traits to model decreasing precedence. The Equality companion object contains the general equivalence class axioms. Trait EqualityLowPriority1 contains the axioms regarding Suc and trait EqualityLowPriority2 contains the axioms regarding addition. Then we have the inheritance hierarchy trait EqualityLowPriority2 ← trait EqualityLowPriority1 ← object Equality. Suppose we import the contents of the Equality companion object. This tells the compiler to first consider the equality axioms in the companion object. If those are not sufficient to build the requested proof, it includes the successor axioms in trait EqualityLowPriority1. In case those are not sufficient either, the compiler moves further up the inheritance hierarchy and includes the axioms for addition in trait EqualityLowPriority2 in its search.

3.4 Remote Requests

λ^{lat} offers three kinds of terms to perform remote communication: $\mathsf{get}\,p.x$, $\mathsf{remoteCall}\,p.f$ and $\mathsf{eval}\,t\,\mathsf{on}\,p$. Hence, Scala$^{\text{Lat}}$ contains implementations of those remote communication operators. All three are defined in our base trait Peer. Their implementation for their implementation we use the remote communication primitives evalAndRelocate requestEvalAndRetrieval offered by the Placement monad and explained in Section 3.2.2. Hence, we now focus on their signatures. Those are presented in Figure 3.6.

In the following let Ps <: Peer[Ps] and Pr <: Peer[Pr] be peer types with instances sen : Ps, rec. In all examples below, sen sends a request to rec. In λ^{lat} remote communication can fail and is hence not deterministic. However, Scala$^{\text{Lat}}$ is assumes a simplified scenario in which all remote communication succeeds. Hence, in contrast to λ^{lat} a remote request's result type is not an Option but simply the type of the sent response.

Remote Value Requests Suppose peer instance sen requests some data data : Placement[A, Pr, La] from peer instance rec. We can express this by a call sen.get(rec.data). This is similar to a request $\mathsf{get}\,\mathit{rec.data}$ in λ^{lat} reduced on peer instance sen. The remote request must be sent from sen to rec. Afterwards, rec.data is evaluated on Pr and rec sends the result to sen. This involves two remote messages and those sent during the evaluation of rec.data. Furthermore, the request's result is placed on peer Ps who sent the request. The request's result type Placement[A, Ps, La#+ [_2]] expresses the latency La#+ [_2] the remote request causes and the peer Ps the result lives on.

```
sealed trait Equality[A <: Nat, B <: Nat]


trait EqualityLowPriority2 {
    implicit def plusLeftNeutral[A <: Nat]: Equality[_0# + [A], A]
        = new Equality[_0# + [A], A] {}

    implicit def plusCommutativity[A <: Nat, B <: Nat]
        : Equality[A# + [B], B# + [A]]
        = new Equality[A# + [B], B# + [A]] {}

    implicit def plusAssociativity[A <: Nat, B <: Nat, C <: Nat]
        : Equality[A# + [B# + [C]], A# + [B]# + [C]]
        = new Equality[A# + [B# + [C]], A# + [B]# + [C]] {}
}

trait EqualityLowPriority1 extends EqualityLowPriority2 {
    implicit def reflexitySuc[A <: Nat, B <: Nat]
        (implicit aEqB: Equality[A, B]): Equality[Suc[A], Suc[B]]
        = new Equality[Suc[A], Suc[B]] {}

    implicit def sucDestruct[A <: Nat, B <: Nat]
        (implicit saEqSb: Equality[Suc[A], Suc[B]]): Equality[A, B]
        = new Equality[A, B] {}
}

object Equality extends EqualityLowPriority1 {
    implicit def reflexity[A <: Nat]: Equality[A, A]
        = new Equality[A, A] {}

    implicit def symmetry[A <: Nat, B <: Nat]
        (implicit aEqB: Equality[A, B]): Equality[B, A]
        = new Equality[B, A] {}

    implicit def transitivity[A <: Nat, B <: Nat, C <: Nat]
        (implicit aEqB: Equality[A,B], bEqC: Equality[B, C])
        : Equality[A, C] = new Equality[A, C] {}
}
```

Figure 3.5: Axiomatized Equality Relation (Simplified)

```
trait Peer[Self <: Peer[Self]] {
    this: Self =>

    def get[A, Pa <: Peer[_], La <: Nat](a: Placement[A, Pa, La])
        : Placement[A, Self, La# + [_2]] = ...

    def remoteCall[A, B, Pf <: Peer[_], Lf <: Nat, Lb <: Nat, La <: Nat]
        (remotePeer: Pf,
         remoteF: Placement[A => HasLatency[B, Lb], Pf, Lf],
         localArg: Placement[A, Self, La])
        : Placement[B, Self, Lf# +[La# +[_1]# +[Lb]]# + [_1]] = ...

    def placedCall[A, B, Prem <: Peer[_], Lf <: Nat, Lb <: Nat, La <: Nat]
        (target: Prem,
         localF: Placement[A => HasLatency[B, Lb], Self, Lf],
         remoteArg: Placement[A, Prem, La])
        : Placement[B, Self, Lf# + [_1]# + [La# + [Lb]]# + [_1]] = ...

}
```

Figure 3.6: Signature of base trait *Peer*

Remote Method Calls have a form sen.remoteCall(rec, rec.f, sen.arg) for a remote method f : Placement[A =>HasLatency[B, Lb], Pr, Lf] and a local argument arg : Placement[A, Ps, La]. Here, sen requests rec to apply its method f to an argument arg that sen provides. We evaluate the argument locally on sen before we transmit the result together with the request to remote peer instance rec. This causes a latency of La#+ [_1]. Then, sen evaluates f and applies the result to the received argument. According to f's type, this causes a latency of Lf#+ Lb. The result is located on peer rec : Pr. Hence, we transmit it to sen : Ps which causes a latency increase of _1. The result type Placement[B, Ps, Lf#+[La#+[_1]#+[Lb]]#+ [_1]] keeps track of the latency the remote request causes as well as of the peer the result is located on. We see that remote method calls in Scala$^{\text{Lat}}$ are analogous to remote applications in λ^{lat}.

Placed Calls are the pendant to remote method calls. They have the form sen.placedCall(rec, sen.f, sen.arg) for a remote argument arg : Placement[A, Pr, La] and local function f : Placement[A =>HasLatency[B, Lb], Ps, Lf]. In contrast to remote method calls, they request to apply a local function f to a remote argument arg on the remote peer instance rec. The result type is Placement[B, Ps, Lf#+ [_1]#+ [La#+ [Lb]]#+ [_1]]. It expresses the caused la-

```
trait ProofSized[+A, S <: Nat] {
    def rewriteSize[S2 <: Nat](implicit s2EqS: EqualitySize[S2, S])
        : ProofSized[A, S2]

    def map[B, L <: Nat](f : A => HasLatency[B, L])
        : HasLatency[Sized[B, S], S# * [L]]
}
```

Figure 3.7: Base Trait for Sized Collections

tency as well as that the result lives on the peer `Ps` sending the request.

3.5 Recursion

In order to extract latency bounds for realistic programs we need to support either loops or recursion. Our implementation primarily focuses on a functional programming style. Hence, we support recursion and neglect loops. As in λ^{lat} we use *sized types*[1] and *size-decreasing recursion* to prove termination and extract latency bounds. However, in contrast to λ^{lat} not all Scala$^{\text{Lat}}$-programs are total, since we cannot forbid the usage of non-terminating Scala expressions like `{def g(x: Int):Int =g(x); g(3)}`.

As the above expression demonstrates, recursion does not terminate in general. Hence, we cannot offer a general approach to extract latency bounds from recursive definitions. Instead we require the programmer to write recursive programs in one of the provably terminating patterns we support.

Sized Collections allow us to store data and to let the type system keep track of their size. In Scala$^{\text{Lat}}$ all sized collections inherit from a common base trait `trait Sized[+A, S <:Nat]`. A subtype of `Sized[A, S]` represents a collection containing exactly `S` elements of type `A`. Figure 3.7 presents its definition

Note that we use the same type `Nat` to represent sizes that we use to express latency and arithmetic type-level computations in general. Reusing `Nat` for sizes allows us to reuse the proof system presented in Section 3.3 to reason about them. This is analogous to λ^{lat} where we use arithmetic terms to represent sizes and latency and where we use Heyting Arithmetic to reason about both.

The trait contains an abstract method `rewriteSize`. This allows the programmer to replace the type-level computation `S` denoting the number of elements by any other provably equal size S2. An application requires an equality proof Equality[S2, S] and returns a collection of type Sized[A, S2]. This ensures that, analogous to latency annotations, we do not force the programmer to differentiate between different representations of the same size.

Figure 3.8 presents the definition of a type for sized lists. Analogous to the standard Scala implementation we define a sealed base trait sealed trait SizedList[+A, S <: Nat] extends Sized[A, S] and derive case classes SizedCons[+A, Sall <:Nat, Stail <: Nat] and SizedNil[+A, Sz <: Nat]. Since the latter represents an empty list, an instantiation of SizedNil[A, Sz] requires a proof that Sz equals _0. The flexibility to represent its size by a more verbose type like S#− [S] is convenient in size-decreasing recursive definitions.

SizedCons[A, Sall, Stail] represents a list of size Sall formed by prepending an element to a tail of size Stail. To guarantee consistency, an instantiation requires a proof that Sall equals Suc[Stail].

A common pattern in Scala programs is to transform a list of type A to a list of type B by applying a function f: A =>B to each element. Suppose a single applicaiton of f causes a latency of L and suppose the transformed list has S elements. Then we get an overall latency of S ∗ L

Then in Scala$^{\text{Lat}}$ we get f : A =>HasLatency[B, Lb]. Hence, our implementation of sized lists defines a method map that accepts a function f : A =>HasLatency[B, Lb] and returns a result of the following type: HasLatency[SizedList[A, S], S#∗ [Lb]].

This allows the programmer to manipulates sized lists in a convenient way while keeping track of the caused latency. In particular, she can use the map method as black-box without worrying about size-decreasing recursion and proofs about type-level computations.

Fixpoint Operator The recursive implementation of map in SizedList uses the concrete structure of lists as compounds of head and tail. We also provide a more general approach to define size-decreasing recursive functions that follows our approach for λ^{lat} presented in Section 2.5.7.2. In λ^{lat} our fixpoint operator fix transforms a bound-increasing function (cf., Def. 31) into an unbounded size-dependent function (cf., Def 30). Let f be a function of basic type $\forall(u:\mathbb{N}).\big((\forall(s:\mathbb{N})<u.B^{\rightarrow},[0])\rightarrow(\forall(s:\mathbb{N})<w.B^{\rightarrow},[0],[0])\big)$ and suppose we can prove that $u<w$ holds for all bounds u. Then, $\mathsf{fix}\,f$ is a recursive function of type $\forall(s:\mathbb{N}).B^{\rightarrow}$.

We follow this approach and implement a similar fixpoint operator in Scala$^{\text{Lat}}$. However, we restrict the increased bound w to depend linearly on u. That is, we require functions where a single application increases the

```
sealed trait SizedList[+A, S <: Nat] extends Sized[A, S]{
    override def rewriteSize[S2 <: Nat]
        (implicit s2EqS: Equality[S2, S])
        : SizedList[A, S2]

    def map[B, L <: Nat](f : A => HasLatency[B, L])
        : HasLatency[SizedList[B, S], S# * [L]]
}


case class SizedCons[+A, Sall <: Nat, Stail <: Nat]
    (head: A, tail: SizedList[A, Stail])
    (implicit val sallEqSucStail: Equality[Sall, Suc[Stail]])
    extends SizedList[A, Sall] {

    override def map[B, L <: Nat](f: A => HasLatency[B, L])
        : HasLatency[SizedList[B, Suc[ST]], Suc[ST] # * [L]]
            = ...

    override def rewriteSize[S2 <: Nat]
        (implicit s2EqSall: Equality[S2, Sall]): SizedList[A, S2] = {
        val s2EqSucStail: Equality[S2, Suc[Stail]]
            = Equality.transitivity(s2EqSall, sallEqSucStail)
        SizedCons(head, tail)(s2EqSucStail)
    }
}


case class SizedNil[+A, Sz <: Nat]()
    (implicit val szEqZero: Equality[Sz, _0]) extends SizedList[A, Sz] {

    override def map[B, L <: Nat](f: A => HasLatency[B, L])
        : HasLatency[SizedList[B, _0], _0 # * [L]]
            = HasLatency(SizedNil[B]())

    override def rewriteSize[S2 <: Nat](implicit s2EqSz: Equality[S2, Sz])
        : SizedList[A, S2] = {
        val s2EqZero: Equality[S2, _0] = Equality.transitivity(s2EqSz, szEqZero)
        SizedNil[A, S2]()(s2EqZero)
    }

    def toZeroSize: ProofSizedList[A, _0] = rewriteSize(zeroEqSz)
}
```

Figure 3.8: Sized Lists

bound by a fixed natural number.

As explained in Section 2.5.7.2, a recursive definition that builds on bound-increasing functions reduces the argument size in every recursive step. Consider the function f from above, some argument a of size s_a and let $w = u+i$ for some fixed number i. Then an application $f\,a$ leads to a recursive step on a smaller argument a' of size $s_{a'}$. Since, $w = u + i$, an application of f to a bounded function increases the bound by i. This implies that the recursive step can only be taken on an argument whose size is reduced by i. Hence, we know that $s_{a'} = s_a \dot{-} i$.

The types of f and a statically tell us about the size decrease i and the argument size s_a. We use this information to statically unroll recursive function applications in Scala$^{\text{Lat}}$.

Before we define our fixpoint operator's type, however, we need a representation for bounded size-dependent and bound-increasing function types. In general, the definition of function types that abstract over input sizes is inconvenient in Scala since it does not support partial application on the type-level. We implement bounded size-dependent function types as derivatives of a base trait with signature:

```
trait BoundedFun[A, B[X <: Nat], PS[X, S <: Nat] <: Sized[X, S],
                 UpB <: Nat]
```

Here, A is the function type's domain, B[X <:Nat] is its size-dependent result type. PS[X, S <:Nat] determines the type of collection and UpB <:Nat is the upper bound restricting the input sizes. In analogy to λ^{lat} we can express the intuition behind this signature by the following informal notation: ∀(S:Size) <=UpB. (PS[A, S] =>B[S]). Furthermore, the trait contains an abstract apply method which requires a proof that the argument size does not exceed the upper bound UpB.

Similarly, we can represent linearly bound-increasing functions by a trait with signature:

```
trait LinearBoundIncreasingFun
              [A, B[S <: Nat], PS[X, S <: Nat] <: Sized[X, S],
               BoundInc <: Suc[_]]
```

In analogy to λ^{lat} we can express the intuation behind this signature by the following informal notation:

$\Big(\forall(\text{S:Size}) <= \text{UpB}.\big(\text{PS[A, S]} => \text{B[S]}\big)\Big)$

$\quad => \quad \forall(\text{S:Size}) <= (\text{UpB} + \text{BoundInc}).\big(\text{PS[A, S]} => \text{B[S]}\big)\Big)$

Using these traits Figure 3.9 presents the definition of our fix point operator linearFix. The fixpoint operator expects a bound increasing function f : LinearBoundIncreasingFun[A, B, PS, BoundInc]) and a desired bound DesiredBound and returns a function of the following type:

```
BoundedFun[A, B, PS, _0] => BoundedFun[A, B, ProofSizedList, PS]
```

Our fixpoint operator statically unrolls the recursive application of f until we get an application sequence of the form f (... f(_)...) that can handle argument up to the desired bound. The result is a function that requires us to specify how to handle arguments of size _0. Being able to provide this start value gives the fixpoint operator additional flexibility.

```
def linearListFix[A, B[S <: Nat], PS[X, S <: Nat] <: Sized[X, S],
                  DesiredBound <: Nat, BoundInc <: Suc[_]]
    (f: LinearBoundIncreasingFun[A, B, PS, BoundInc])
    (implicit cont: LinearBoundIncreasingContinuation[DesiredBound, BoundInc])
    : BoundedFun[A, B, PS, _0] => BoundedFun[A, B, PS, DesiredBound]
    = { cont(f) }
```

Figure 3.9: Fixpoint Operator for Linearly Size-Decreasing Recursion

Recursion Unrolling Consider a valid application linearFix(f). We can statically determine the bound increase BoundInc each application of f causes and the bound DesiredBound we want to reach by unrolling the recursion. Our fixpoint operator returns a function that requests information how to handle argument of size _0. Hence, it suffices to apply f exactly n times for the smallest number n and its corresponding type N <:Nat for which DesiredBound#- [N#* [BoundInc]] equals _0.

Additional to the equality relation Equality[A <:Nat, B <:Nat] presented in Section 3.3, we define a relation LT[A <:Nat, B <:Nat] to express that A is strictly smaller than B. For each concrete number N <:Nat the Scala compiler can either infer a proof nEq0 : Equality[N, _0] or a proof zeroLtN : LT[_0, N].

Suppose each application of f increases the bound by _1 and suppose we want to unroll f until we obtain a function that (provided a start value) can handle arguments up to size 10. Using our proof system, the Scala compiler can statically infer a proof for LT[_0, _10]. Hence, we know that we have to apply f (...) . Suppose we had a continuation g that could handle arguments up to size _10#- [_1], then f(g) was a function that (provided a start value) could handle arguments up to our desired bound _10. However, it remains to construct g.

Again, the Scala compiler can infer a proof for LT[_0, _10#- [_1]] . Hence, we need to insert another application of f and thereby obtain f(f (...)) . We can proceed until the compiler infers that we reached `_0` and are finished.

During this process we statically and recursively build a continuation. Hence, we can also define a recursive continuation that applies `f` a certain number of times and let the Scala compiler statically construct it for us.

Figure 3.10 shows its definition. We define a trait
`trait LinearBoundIncreasingContinuation[DesiredBound <:Nat, BoundInc <:Suc[_]]` and two methods as ways to instantiate it:

- The following method asks the complier to infer a proof that our desired bound equals `_0`.

  ```
  implicit def stopAtZero[DesiredBound <: Nat, BoundInc <: Suc[_]]
    (implicit zeroEqDb: EqualitySize[_0, DesiredBound])
    : LinearBoundIncreasingContinuation[DesiredBound, BoundInc]
  ```

  If the compiler succeeds we are finished and hence instantiate a continuation that does nothing.

- The following method asks the compiler to infer a proof that our desired bound is greater than 0. Furthermore, it asks the compiler to instantiate a continuation `cont` which builds a function that can handle arguments up to size `DesiredBound#- [BoundInc]`.

  ```
  implicit def nextStep[DesiredBound <: Nat, BoundInc <: Suc[_]]
    (implicit bLtDb: LTSize[_0, DesiredBound],
              cont:  LinearBoundIncreasingContinuation
                       [DesiredBound# - [BoundInc], BoundInc])
    : LinearBoundIncreasingContinuation[DesiredBound, BoundInc]
  ```

  If the compiler succeeds this method uses the continuation `cont` and inserts an additional application of `f`. Thereby we obtain a continuation that builds a function which can handle arguments up to our desired bound.

The presented fixpoint operator does not compute latency bounds itself. However, we can apply it to a bound-increasing function that in its return type encodes some latency. Then, the same holds for the fixpoint operator's return type.

```
trait LinearBoundIncreasingContinuation
    [DesiredBound <: Nat, BoundInc <: Suc[_]] {

    def apply[A, B[S <: Nat], PS[X, S <: Nat] <: Sized[X, S]]
    (f: LinearBoundIncreasingFun[A, B, PS, BoundInc])
    : BoundedFun[A, B, PS, _0] => BoundedFun[A, B, PS, DesiredBound]
}

trait LowPriorityLBIC {
    implicit def stopAtZero[DesiredBound <: Nat, BoundInc <: Suc[_]]
        (implicit zeroEqDb: Equality[_0, DesiredBound])
        : LinearBoundIncreasingContinuation[DesiredBound, BoundInc]
        = new LinearBoundIncreasingContinuation[DesiredBound, BoundInc] {
            override def apply[A, B[S <: Nat],
                                  PS[X, S <: Nat] <: Sized[X, S]]
                (f: LinearBoundIncreasingFun[A, B, PS, BoundInc])
                : BoundedFun[A, B, PS, _0]
                      => BoundedFun[A, B, PS, DesiredBound]
                = (pbf: BoundedFun[A, B, PS, _0]) => {
                    pbf.rewriteBound(Equality.symmetry(zeroEqDb))
                }
        }
}

object LinearBoundIncreasingContinuation extends LowPriorityLBIC {
    implicit def nextStep[DesiredBound <: Nat, BoundInc <: Suc[_]]
        (implicit bLtDb: LT[_0, DesiredBound],
         cont: LinearBoundIncreasingContinuation
                  [DesiredBound# - [BoundInc], BoundInc])
    : LinearBoundIncreasingContinuation[DesiredBound, BoundInc]
    = new LinearBoundIncreasingContinuation[DesiredBound, BoundInc] {
        override def apply[A, B[S <: Nat],
                              PS[X, S <: Nat] <: ProofSized[X, S]]
            (f: LinearBoundIncreasingFun[A, B, PS, BoundInc])
            : BoundedFun[A, B, PS, _0] => BoundedFun[A, B, PS, DesiredBound]
            = (pbf: BoundedFun[A, B, PS, _0]) => {
                val res: BoundedFun[A, B, PS,
                              DesiredBound# - [BoundInc]# + [BoundInc]]
                        = f(cont(f)(pbf))

                val leq: LEQ[DesiredBound,
                               DesiredBound# -[BoundInc]# +[BoundInc]]
                         = LEQ.minPlusRight
                res.restrict[DesiredBound](leq)
            }
    }
}
```

Figure 3.10: Continuation for Linear Recursion Unrolling

Chapter 4

Related Work

We previously outlined the ideas behind the presented language design, however, without a full formalization and without any implementation [22]. To the best of our knowledge, no previous work explores type-level latency tracking to promote low-latency computations.

[10] augment a type system with cost values to extract upper bounds on the worst-case execution time and heap space usage. Their approach, however, targets embedded systems where both time and space bounds are important. [6] propose an incremental, model-based approach to analyze the validity of latency requirements in cyber-physical systems. [5] present a type system raising the developer's awareness for inefficient code in terms of energie consumption. Their approach augments types by energy consumption patterns and uses type inference to track a program's energy consumption. Session types (e.g., [9]) has been successfully applied to distributed programming to check distributed protocols, but focus on protocol correctness rather than communication cost.

[12] and [13] annotate data with their location and infer whether data and references are shared or private. [16] use a type system to verify constant-resource-usage properties for programs, including constant-time requirements.

[1] use sized types statically reason about termination of recursive and productivity of corecursive in higher-kinded polymorphic programs. In [2] they combine their approach with a dependent type theory to allow termination checking for proof assistants such as Agda. Our approach only uses a restricted form of sized types where sizes are natural numbers instead of ordinals. Furthermore, we only implement size-dependent functions instead of general dependent types. However, while their approaches use size types

mereley as upper bounds, our size indices represent exact sizes. This allows us to extract better static bounds on the number of recursive steps and hence extract better static latency bounds than the usage of sizes as upper bounds would.

The Scala library *Shapeless*[18] implements sized types (restricted to natural numbers) and uses them to statically track the sizes of collections. However, they represent arithmetic type-level computations as traits implicitly instantiated by the compiler. In particular they use different types like `trait Sum` and `trait Diff` to represent arithmetic operations. Our approach uses a uniform representation that simplifies static reasoning about arithmetic type-level computations.

Chapter 5

Conclusion

In this work we proposed a language design that makes latency and locations transparent to the programmer. We presented a full formalization λ^{lat} including a type system that extracts static upper bounds on a program's runtime latency. λ^{lat} is an extension of the typed λ-calculus where we augment types by size and latency annotations. Furthermore, we make locations explicit in our type system. The latency caused by a function application may depend on the argument size or on the number of recursive steps a function takes until it terminates. In λ^{lat} we are able to extract upper latency bounds for such cases. Furthermore, we proved that all exatracted bounds are correct.

Moreover, we presented a prototype implementation Scala$^{\text{Lat}}$ in form of a Scala DSL. Scala$^{\text{Lat}}$ exposes locations and latency to the programmer as part of a computation's type. In Scala$^{\text{Lat}}$ we implemented sized types and a fixpoint operator for size-decreasing recursion. This allows the programmer to prove termination of recursive functions and to extract correspoding latency bounds. Furthermore, Scala$^{\text{Lat}}$ includes a proof system that allows the programmer to statically reason about sizes and latency bounds. We can use Scala$^{\text{Lat}}$ to asses the practicality of our proposed language design.

Chapter 6

Outlook

As message delay in distributed systems is non-deterministic, we plan to refine our approach by using probability distributions for the latency weights $\mathcal{L}(P, P')$ instead of natural numbers.

An important aspect to consider is to complement the (static) analysis provided by the type system with actual latency measurements collected via monitoring. We believe that the combination of both can provide correct feedback to the developers. To this end, we are going to work on a monitoring system that provides realistic estimations for latency and retrofits them in the type system using methods from continuous integration.

We are currently extending our prototype implementation Scala$^{\text{Lat}}$ to allow the specification of latency weights $\mathcal{L}(P, P')$. Eventually, we are going to implement type-based latency tracking in ScalaLoci [21], a multitier language whose type system keeps track of a computation's location similar to λ^{lat} and Scala$^{\text{Lat}}$.

Using ScalaLoci's extended type system, we are going to explore latency-saving refactorings. High-latency inducing computations often contain unnecessary remote communication. Relocating parts of the computation and only transmitting as few data as necessary helps to reduce latency. We believe that the combination of static location and latency information is sufficient to implement such refactorings.

We plan to evaluate the type system's usability with controlled experiments and case studies on applications involving multiple geo-distributed data centers. Using platforms like Amazon AWS, we plan to use real locations for the data centers [7] and to specify realistic latency weights $\mathcal{L}(P, P')$ for the connections.

Appendix A

Proof of Theorem 2. We prove the proposition by induction over the structure of term t. Let B, s, l be a basic type, a size and a latency, respectively, such that $T = (B, s, l)$.

Base case: The theorem holds for $\mathit{value}(t)$, since no reduction step $(\langle t \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t' \rangle_{\mathcal{I}}, [l'_c])$ can be taken. The same holds for $t = x \in \mathbb{X}$.
Inductive step: In the following we assume that t is not a value.

1. $t = \mathsf{some}\, t^*$ for a term t^*.

   The only typing rule typing terms of the form $\mathsf{some}\, t^*$ is T-Some. By this, we get $B = \mathsf{Option}\,(B^*, [s^*]), s = 0$ and $\Delta; \Gamma; \Lambda; P \vdash t^* : (B^*, [s^*], [l])$ for some basic type B^* and a size s^*.

   The only applicable reduction rule is E-LocalContext. Hence, $t' = \mathsf{some}\, t^{*\prime}$ and $(\langle t^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^{*\prime} \rangle_{\mathcal{I}}, [l'_c])$. By induction hypothesis $\Delta; \Gamma; \Lambda; P \vdash t^{*\prime} : (B^*, [s^*], [l_c])$ holds. By typing rule T-Some we get $\Delta; \Gamma; \Lambda; P \vdash \mathsf{some}\, t^* : (\mathsf{Option}\,(B^*, [s^*]), [0], [l])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

2. $t = \mathsf{cons}\, t_h^*\, t_t^*$

   The only typing rule typing terms of the form $\mathsf{cons}\, t_h^*\, t_t^*$ is T-Cons. By this, we get $B = \mathsf{List}(B^*, [s^*])$, $\Delta; \Gamma; \Lambda; P \vdash t_h^* : (B^*, [s^*], [l_h])$, $\Delta; \Gamma; \Lambda; P \vdash t_t^* : (\mathsf{List}(B^*, [s^*]), [s_t], [l_t])$ and $l = l_h + l_t$, $s = s_t + 1$ for some basic type B^*, some sizes s^*, s_t and latencies l_h, l_t.

   The only applicable reduction rule is E-LocalContext. Suppose t_h^* is not a value. Then by E-LocalContext and the induction hypothesis we get $\left(\langle t_h^* \rangle_{\mathcal{I}}, [l_h]\right) \overset{\mathcal{I}}{\rightsquigarrow} \left(\langle t_h^{*\prime} \rangle_{\mathcal{I}}, [l'_h]\right)$, $t' = \mathsf{cons}\, t_h^{*\prime}\, t_t^*$ and $\Delta; \Gamma; \Lambda; P \vdash t_h^{*\prime} : (B^*, [s^*], [l_h])$. Rule T-Cons implies $\Delta; \Gamma; \Lambda; P \vdash \mathsf{cons}\, t_h^{*\prime}\, t_t^* : (\mathsf{List}(B^*, [s^*]), [s_t + 1], [l_h + l_t])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t_t^{*\prime} : T$.

Suppose t_h^* is not a value. Then by E-LocalContext and the induction hypothesis we get $(\langle t_t^* \rangle_{\mathcal{I}}, [l_t]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t_t^{*\prime} \rangle_{\mathcal{I}}, [l_t'])$ and $\Delta; \Gamma; \Lambda; P \vdash t_t^{*\prime} : (\mathsf{List}(B^*, [s^*]), [s_t], [l_t])$. By T-Cons this implies $\Delta; \Gamma; \Lambda; P \vdash \mathsf{cons}\, t_h^*\, t_t^{*\prime} : (\mathsf{List}(B^*, [s^*]), [s_t + 1], [l_h + l_t])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

3. $t = f^*\, a^*$

   Depending on the type of f^*, the term $f^*\, a^*$ can be typed using one of the rules T-LocalApp, T-SizeDepLocalApp, T-BoundedSizeDepLocalApp.

   (a) Suppose the typing $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^* : T$ stems from application of T-LocalApp. Then $\Delta; \Gamma; \Lambda; P \vdash f^* : ((B_{f^*}, [s_{f^*}]) \to (B, [s], [l']), [0], [l_{f^*}])$, $\Delta; \Gamma; \Lambda; P \vdash a^* : (B_{a^*}, [s_{a^*}], [l_{a^*}])$ and $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^* : (B, [s], [l_{f^*} + l_{a^*} + l'])$ for basic types B_{f^*}, B_{a^*}, sizes s_{f^*}, s_{a^*} and latencies l_{f^*}, l', l_{a^*}. We further get $\Lambda \Vdash B_{f^*} \approx B_{a^*}$ and $\Lambda \vdash s_{f^*} =_0 s_{a^*}$.

   Depending on whether f^* and a^* are values, applicable reduction rules are E-LocalContext and E-LocalApp.

   Suppose f^* is no value. Then the only applicable reduction rule is E-LocalContext. By this and the induction hypothesis we get $(\langle f^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} \left(\langle f^{*\prime} \rangle_{\mathcal{I}}, [l_c']\right)$ and $\Delta; \Gamma; \Lambda; P \vdash f^{*\prime} : ((B_{f^*}, [s_{f^*}]) \to (B, [s], [l']), [0], [l_{f^*}])$. By typing rule T-LocalApp, we get $\Delta; \Gamma; \Lambda; P \vdash f^{*\prime}\, a^* : (B, [s], [l_{f^*} + l_{a^*} + l'])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

   Suppose f^* is a value but a^* is not. Then the only applicable reduction rule is E-LocalContext. By this and the induction hypothesis we get $(\langle a^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} \left(\langle a^{*\prime} \rangle_{\mathcal{I}}, [l_c']\right)$ and $\Delta; \Gamma; \Lambda; P \vdash a^{*\prime} : (B_{a^*}, [s_{a^*}], [l_{a^*}])$. Hence, by typing rule T-LocalApp we get $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^{*\prime} : (B, [s], [l_{f^*} + l_{a^*} + l'])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

   Suppose f^* and a^* both are values. Then the typing information $\Delta; \Gamma; \Lambda; P \vdash f^* : ((B_{f^*}, [s_{f^*}]) \to (B, [s], [l']), [0], [l_{f^*}])$ implies that f^* has the form $f^* = \lambda x : (B_{f^*}, [s_{f^*}]).t^*$ with $\Delta; \Gamma; \Lambda; P \vdash t^* : (B, [s], [l'])$ and the only applicable reduction rule is E-LocalApp. By this and the induction hypothesis we get $(\langle f^*\, a^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^*[x \mapsto a^*] \rangle_{\mathcal{I}}, [l_c])$ and $\Delta; \Gamma; \Lambda; P \vdash t^*[x \mapsto a^*] : (B, [s], [l'])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

   (b) Suppose the typing $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^* : T$ stems from application of T-SizeDepLocalApp. Then $\Delta; \Gamma; \Lambda; P \vdash f^* : (\forall (\hat{s} : \mathbb{N}) \,.\, (B_{f^*}, [\hat{s}]) \to (B', [s'], [l']), [0], [l_{f^*}])$ and $\Delta; \Gamma; \Lambda; P \vdash a^* : (B_{a^*}, [s_{a^*}], [l_{a^*}])$ with $\Lambda \Vdash B_{f^*} \approx B_{a^*}$, $B = B'[s \mapsto s_{a^*}]$, $s =$

$s'[\hat{s} \mapsto s_{a^*}]$ and $l = l'[\hat{s} \mapsto s_{a^*}]$ for basic types B_{f^*}, B', B_{a^*}, sizes s', s_{a^*} and latencies l_{f^*}, l', l_{a^*}.

Depending on whether f^* and a^* are values, applicable reduction rules are E-LocalContext and E-LocalApp.

i. Suppose f^* is not a value. Then the only applicable reduction rule is E-LocalContext and together with the induction hypothesis we get $(\langle f^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle f^{*\prime} \rangle_{\mathcal{I}}, [l_c']\big)$ and $\Delta; \Gamma; \Lambda; P \vdash f^{*\prime} : (\forall(\hat{s} : \mathbb{N}) \,.\, (B_{f^*}, [\hat{s}]) \to (B', [s'], [l']), [0], [l_{f^*}])$ By typing rule T-LocalApp we get $\Delta; \Gamma; \Lambda; P \vdash f^{*\prime}\, a^* : (B'[\hat{s} \mapsto s_{a^*}], [s'[\hat{s} \mapsto s_{a^*}]], [l_{f^*} + l_{a^*} + l'[\hat{s} \mapsto s_{a^*}]])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

ii. Suppose f^* is a value but a^* is not. Then the only applicable reduction rule is E-LocalContext. Together with the induction hypothesis the reduction rule leads to $(\langle a^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} \big(\langle a^{*\prime} \rangle_{\mathcal{I}}, [l_c']\big)$ and $\Delta; \Gamma; \Lambda; P \vdash a^{*\prime} : (B_{a^*}, [s_{a^*}], [l_{a^*}])$. By applying typing rule T-SizeDepLocalAppName we get $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^* : (B'[\hat{s} \mapsto s_{a^*}], [s'[\hat{s} \mapsto s_{a^*}]], [l'[\hat{s} \mapsto s_{a^*}]])$ i.e., $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

iii. Suppose both f^* and a^* are values. Then the typing $\Delta; \Gamma; \Lambda; P \vdash f^* : (\forall(\hat{s} : \mathbb{N}) \,.\, (B_{f^*}, [\hat{s}]) \to (B', [s'], [l']), [0], [l_{f^*}])$ can stem from application of T-ForAll or T-FixApp.
Suppose f^* is typed by T-ForAll. Then it has the form $f^* = \forall(\hat{s} : \mathbb{N}) \,.\, \lambda x : (B_{f^*}, [\hat{s}]).t^*$ for a term t^* with $\Delta; \Gamma, \hat{s} \mapsto \mathbb{N}; \Lambda; P \vdash t^* : (B', [s'], [l'])$. The only applicable reduction rule is E-LocalApp which together with the induction hypothesis leads to
$\big(\langle \forall(\hat{s} : \mathbb{N}) \,.\, \lambda x : (B_{f^*}, [\hat{s}]).t^*\, a^* \rangle_{\mathcal{I}}, [l_c]\big) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^*[\hat{s} \mapsto a^*] \rangle_{\mathcal{I}}, [l_c])$ and $\Delta; \Gamma; \Lambda; P \vdash t^*[\hat{s} \mapsto a^*] : (i, [.], [e]).$, $t' = t^*[\hat{s} \mapsto a^*]$ and $\Delta; \Gamma; \Lambda; P \vdash t' : T$.
Suppose f^* is typed by T-FixApp. Then it has the form $f^* = \mathsf{fix}\, f^{**}$ for some value f^{**} with

$$\begin{array}{lll} \Delta; \Gamma; \Lambda; P & \vdash & f^{**} \\ & : & (\forall(u : \mathbb{N}) \,.\, ((\forall(\hat{s} : \mathbb{N}) < u \,.\, B^{\to}, [0]) \\ & & \qquad \to (\forall(\hat{s} : \mathbb{N}) < w(\hat{s}) \,.\, B^{\to}, [0], [0]), \\ , & & [0], [l_{f^*}]) \end{array}$$

where $B^{\to} = (B_{f^*}, [\hat{s}]) \to (B', [s'], [l'])$ and $\Lambda \vdash \forall(u : \mathbb{N}) \,.\, u <_0 w(u)$. Since f^{**} is a value, its typing implies that it has the form $f^{**} = \forall(u : \mathbb{N}) \,.\, \lambda x : (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\to}, [0]).t^*$ with

$\Delta; \Gamma, u \mapsto \mathbb{N}; \Lambda; P \vdash t^* : (\forall(\hat{s} : \mathbb{N}) < w(u) \,.\, B^{\rightarrow}, [0], [0])$.
The only applicable reduction rule is E-FixApp. Thus, we get $(\langle \mathsf{fix}\, f^{**}\, a^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle f^{**\prime}\, \mathsf{fix}\, f^{**} \rangle_{\mathcal{I}}, [l_c])$ with $f^{**\prime} = \forall(u : \mathbb{N}) \,.\, \lambda x : (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow}, [0]).BE(t^*)$. By lemma 1 we get $\Delta; \Gamma, u \mapsto \mathbb{N}, x \mapsto (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow}, [0]); \Lambda; P \vdash BE(t^*) : (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow\prime}, [0], [0])$ with $\Lambda \Vdash B^{\rightarrow} \approx B^{\rightarrow\prime}$
By application of the typing rules T-Abs and T-ForAll, we find that $\Delta; \Gamma; \Lambda; P \vdash f^{**\prime} : (\forall(u : \mathbb{N}) \,.\, (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow}, [0]) \rightarrow (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow\prime}, [0], [0]), [0], [0])$ Thus, by typing rule T-SizeDepLocalApp we get $\Delta; \Gamma; \Lambda; P \vdash f^{**\prime}\, (\mathsf{fix}\, f^{**}) : (\forall(\hat{s} : \mathbb{N}) \,.\, B^{\rightarrow\prime}, [0], [l_{f^*}])$ Let $t' = f^{**\prime}\, (\mathsf{fix}\, f^{**})\, a^*$ then by typing rule T-SizeDepLocalApp, we get $\Delta; \Gamma; \Lambda; P \vdash t' : (\hat{B}', [s'], [l'])$ for a basic type $\hat{B}'$, a size s' and a latency l' such that $\Lambda \Vdash B \approx \hat{B}'$, $\Lambda \vdash s =_0 s'$, $\Lambda \vdash l =_0 l'$.

(c) Suppose the typing $\Delta; \Gamma; \Lambda; P \vdash f^*\, a^* : T$ stems from application of the typing rule T-BoundedSizeDepLocalApp. Then $\Delta; \Gamma; \Lambda; P \vdash f^* : (\forall(\hat{s} : \mathbb{N}) < u \,.\, (B_{f^*}, [\hat{s}]) \rightarrow (B', [s'], [l']), [0], [l_{f^*}])$ and $\Delta; \Gamma; \Lambda; P \vdash a^* : (B_{a^*}, [s_{a^*}], [l_{a^*}])$ with $\Lambda \Vdash B_{f^*} \approx B_{a^*}$ and $\Lambda \vdash s_{a^*} <_0 u$ for basic types B_{f^*}, B', size s_{a^*} and latencies l_{f^*}, l', l_{a^*}. Furthermore, we know that $B = B'[\hat{s} \mapsto s_{a^*}]$, $s = s'[\hat{s} \mapsto s_{a^*}]$ and $l = l_{f^*} + l_{a^*} + l'[\hat{s} \mapsto s_{a^*}]$.
Depending on whether f^* and a^* are values, the reduction rules E-LocalContext and E-LocalApp are applicable.
The cases where f^* is not a value and where f^* is a value but a^* is not are analogous to the cases 3(b)i and 3(b)ii, respectively. Only unbounded quantification must be replaced by bounded quantification.
Suppose f^* and a^* are values. Then f^* has the form $f^* = \forall(\hat{s} : \mathbb{N}) < u \,.\, \lambda x : (B_{f^*}, [\hat{s}]).t^*$ for some term t^* and the only applicable reduction rule is E-LocalApp. By this we get $(\langle f^*\, a^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^*[x \mapsto a^*] \rangle_{\mathcal{I}}, [l_c])$. The typing $\Delta; \Gamma; \Lambda; P \vdash \forall(\hat{s} : \mathbb{N}) < u \,.\, \lambda x : (B_{f^*}, [\hat{s}]).t^* : (\forall(\hat{s} : \mathbb{N}) < u \,.\, (B_{f^*}, [\hat{s}]) \rightarrow (B', [s'], [l']), [0], [l_{f^*}])$ must stem from the seqeuntially applied rules T-ForAllBounded and T-Abs. Therefore, we know that $\Delta; \Gamma; \Lambda; P \vdash t^*[x \mapsto a^*] : (B'[\hat{s} \mapsto s_{a^*}], [s'[\hat{s} \mapsto s_{a^*}]], [l'[\hat{s} \mapsto s_{a^*}]])$ i.e., $t' = t^*[x \mapsto a^*]$ and $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

4. $t = \mathsf{let}\, x : (B_x, [s], [l_x]) \coloneqq t_x^*\, \mathsf{in}\, t_s^*$
Trivial.

5. $t = \mathsf{if}\, t_c^* \, \{t_t^*\} \, \{t_f^*\}$ Already proven in Section 2.6.2.

6. $t = t^* \, \mathsf{match}\{\mathsf{some}\, x \Rightarrow t_c^*\}\{\mathsf{none}\, (B', [s']) \Rightarrow t_n^*\}$
   Analogous to the above case.

7. $t = t^* \, \mathsf{match}\{\mathsf{cons}\, x\, y \Rightarrow t_c^*\}\{\mathsf{nil}\, (B', [s']) \Rightarrow t_n^*\}$ Analogous to the above case.

8. $t = \mathsf{get}\, p^*.t^*$ Trivial.

9. $t = \mathsf{remoteCall}\, p^*.f^*\, a^*$
   Analogous to case $t = f^*\, a^*$.

10. $t = \mathsf{eval}\, t^* \, \mathsf{on}\, p^*$ Analogous to case $t = \mathsf{get}\, p^*.t^*$.

11. $t = \mathsf{fix}\, f^*\, a^*$
    Already proven in Section 2.6.2.

12. $t = \forall(\hat{s} : \mathbb{N})\,.\,t^*$

    Depending on whether $\hat{s} \in FAV(t^*)$ holds or not, the term $\forall(\hat{s} : \mathbb{N})\,.\,t^*$ can be typed by T-ForAll or T-ForAllIgnore.

    Suppose $\hat{s} \in FAV(t^*)$ holds. Then the only applicable typing rule is T-ForAll, by which we get $B = \forall(\hat{s} : \mathbb{N})\,.\,B^*$ for some basic type B^* and $\Delta; \Gamma; \Lambda; P \vdash t^* : (B^*, [s], [l])$. The only applicable reduction rule is E-LocalContext. Applying this rule and using the induction hypothesis leads to $(\langle t^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^{*\prime} \rangle_{\mathcal{I}}, [l_c'])$ and $\Delta; \Gamma; \Lambda; P \vdash t^{*\prime} : (B^*, [s], [l])$. By applying T-ForAll we get $\Delta; \Gamma; \Lambda; P \vdash \forall(\hat{s} : \mathbb{N})\,.\,t^* : (\forall(\hat{s} : \mathbb{N})\,.\,B^*, [s], [l])$. Hence $t' = \forall(\hat{s} : \mathbb{N})\,.\,t^*$ and $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

    Suppose $\hat{s} \notin FAV(t^*)$ holds. Then the only applicable typing rule is T-ForAllIgnore, by which we get $\Delta; \Gamma; \Lambda; P \vdash t^* : (B, [s], [l])$. The only applicable reduction rule is E-ForAllElim which leads to $(\langle \forall(\hat{s} : \mathbb{N})\,.\,t^* \rangle_{\mathcal{I}}, [l_c]) \overset{\mathcal{I}}{\rightsquigarrow} (\langle t^* \rangle_{\mathcal{I}}, [l_c'])$. Hence $t' = t^*$ and $\Delta; \Gamma; \Lambda; P \vdash t' : T$.

13. $t = \forall(s : \mathbb{N}) < u\,.\,t^*$

    This case is analogous to the one above except that the typing rules T-ForAll and T-ForAllIgnore are replaced by T-ForAllBounded and T-ForAllBoundedIgnore, respectively.

14. $t = (\langle t^* \rangle_{\mathcal{I}'}, [l_c^*])$
    Trivial.

$\square$

Bibliography

[1] Andreas Abel. Type-based termination: a polymorphic lambda-calculus with sized higher-order types. 2007.

[2] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1:33:1–33:30, 2017.

[3] Henk Barendregt, J Barwise, D Kaplan, HJ Keisler, P Suppes, and AS Troelstra. Studies in logic and the foundations of mathematics, 1984.

[4] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. In *Proceedings of PACMPL*, OOPSLA '17, New York, NY, USA, 2017. ACM.

[5] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *Proceedings of OOPSLA '12*, New York, NY, USA, 2012. ACM.

[6] Julien Delange and Peter H. Feiler. Incremental latency analysis of heterogeneous cyber-physical systems. In *Proceedings of REACTION '14*, 2014.

[7] Google Data Center FAQ. `http://www.datacenterknowledge.com/archives/2012/05/15/google-data-center-faq/`, 2012. Accessed 2018-04-16.

[8] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[9] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_22. URL `http://dx.doi.org/10.1007/978-3-540-70592-5_22`.

[10] Steffen Jost, Hans-Wolfgang Loidl, Norman Scaife, Kevin Hammond, Greg Michaelson, and Martin Hofmann. Worst-case execution time analysis through types. In *Proceedings of ECRTS '09*, 2009.

[11] Ulrich Kohlenbach. Applied proof theory - proof interpretations and their use in mathematics. In *Springer Monographs in Mathematics*, 2008.

[12] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *POPL*, 2000.

[13] Ben Liblit, Alexander Aiken, and Katherine A. Yelick. Type systems for distributed data sharing. In *SAS*, 2003.

[14] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of DEBS '18*, New York, NY, USA, 2018. ACM.

[15] A. Margara and G. Salvaneschi. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering*, 2018.

[16] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, May 2017. doi: 10.1109/SP.2017.53.

[17] Benjamin C. Pierce. Types and programming languages. 2002.

[18] Miles Sabin. Shapeless. `https://github.com/milessabin/shapeless`. Accessed: 18.12.2018.

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13:181–210, 1991.

[20] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. Quality-aware runtime adaptation in complex event processing. In *Proceedings of SEAMS '17*, Piscataway, NJ, USA, 2017. IEEE Press.

[21] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, October 2018. ISSN 2475-1421. doi: 10.1145/3276499. URL `http://doi.acm.org/10.1145/3276499`.

[22] Pascal Weisenburger, Tobias Reinhard, and Guido Salvaneschi. Static latency tracking with placement types ftfjp. In *Proceedings of the 20th Workshop on Formal Techniques for Java-like Programs*, FTfJP'20, 2018.

[23] Mike P. Wittie, Veljko Pejovic, Lara Deek, Kevin C. Almeroth, and Ben Y. Zhao. Exploiting locality of interest in online social networks. In *Proceedings of CoNEXT '10*, New York, NY, USA, 2010. ACM.