DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

Master's thesis advised by Andrzej Filinski.

Formalizing the Metatheory of Metaprogramming in a Metalogical Framework

by Stine Søndergaard

29 August 2008

Abstract

We investigate the feasibility of formalizing the syntaxes, type systems, operational semantics, and especially the metatheories of a collection of multi-stage programming languages within the metalogical framework Twelf, a framework usually employed when dealing with single-stage programming systems.

More specifically, we consider three different modal-logical based systems for staged computation, and we succeed in adequately representing them all within Twelf. Compared to most applications of Twelf, proving correctness of the representations is here a non-trivial task. The problem of distinguishing variables from different stages is solved by combining standard higher-order abstract syntax with an intrinsic encoding of world-based-scopes inspired by Kripke semantics; we show that this approach does not prevent the representation of operational semantics and Twelf verifications of relevant metatheorems. One of the treated metaprogramming systems comprises two languages, and a key result is a successful Twelf formalization of a type preserving translation between these two languages by introducing an intermediate first-order representation of free variables. We also challenge our representation methodology by in one case formulating a small-step operational semantics of the language, proving it equivalent to the big-step one, and showing type soundness by the standard method of progress and preservation – all formalized within Twelf.

The experience gained suggests that even though Twelf does not provide any explicit support for reasoning about metaprogramming features, its general facilities are powerful and flexible enough to allow multi-stage programming systems to be formalized within the framework, with effort mostly comparable to conventional single-stage cases.

Resumé

Vi undersøger muligheden for formalisering af syntakser, typesystemer, operationssemantikker og særligt metateorier for en samling flertrinsprogrammeringssprog i det metalogiske rammeværk Twelf, et rammeværk der sædvanligvis anvendes i forbindelse med ettrinsprogrammeringssystemer.

Mere specifikt betragter vi tre modallogikbaserede systemer til håndtering af trinvis beregning, og vi får dem alle tilfredsstillende repræsenteret i Twelf. Til forskel fra de fleste andre anvendelser af Twelf er det her en ikke-triviel opgave at bevise korrektheden af repræsentationerne. Problemet med at skelne variable, der tilhører forskellige trin, løses ved at kombinere standard højere-ordens abstrakt syntaks med en Kripkesemantisk inspireret intrinsisk indkodning af verdensbaserede virkefelter. Vi viser, at denne metode ikke forhindrer repræsentation af operationssemantikker og maskinverificering af relevante metasætninger i Twelf. Ét af de behandlede metaprogrammeringssystemer omfatter to programmeringssprog, og ét af vores nøgleresultater er en Twelf-formalisering af en typebevarende oversættelse mellem disse to sprog under indførelse af en midlertidig første-ordens repræsentation af frie variable. Vi udfordrer også repræsentationsmetodologien ved i ét tilfælde at formulere en lilleskridtsoperationssemantik, vise denne ækvivalent med den givne storskridtssemantik og vise typesyndhed via fremskridts- og bevaringsegenskaben – alt formaliseret i Twelf.

De gjorte erfaringer indikerer, at selvom Twelf ikke direkte understøtter ræsonnering om konstruktioner til metaprogrammering, så er de generelle faciliteter stærke og fleksible nok til, at flertrinsprogrammeringssystemer i de fleste tilfælde uden videre kan formaliseres i Twelf.

Preface

This is the final project (speciale) in achieving my master's degree in Computer Science (cand.scient.) at the Department of Computer Science, University of Copenhagen (DIKU).

I was introduced to the Edinburgh Logical Framework and Twelf following the graduate course Computation and Deduction, and I came to like the way of flavouring paper maths with practical machine verifications. It thus felt natural to find the subject for my master's thesis within this area.

I would like to express my utmost gratitude to my project advisor Andrzej Filinski. Had it not been for his tremendous expertise, his kindness, and his great, great patience there would for sure have been no project at all. Among many things, Andrzej taught me that you can never ever be too precise. Also I am very grateful that I was offered the opportunity to do the last months of writing behind a student's desk located in the TOPPS area at Southern Campus at Amager. The people there have all been incredibly friendly and obliging. Not least Jakob Grue Simonsen who is a very skilled whipper-in. Finally, I would like to thank my family and friends for their encouragement and support and for inviting me over for healthy meals during this long process. A special thank you goes to my sister Rikke Holten for her patient layout assistance and to Klavs Mulvad for doing proofreadings despite the endless amount of strange characters.

Contents

Chapter 1

Introduction

1.1 Metaprogramming

Metaprogramming covers the discipline of writing programs to be used in generation or manipulation of other programs. *Staged computation* is a large subject within metaprogramming, referring to the field of efficient execution of programs with staggered program input. A common example is partial evaluation, where a program is specialized into a more efficient version through some precomputing, exploiting the availability of specialization-time program input. For instance, consider the Mini-ML program implementing the function raising a natural number x to the power of some other natural number n:

$$\begin{aligned} \mathit{power} \quad \equiv \quad & \mathbf{fix}\ p : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}\,. \\ & \lambda x : \mathbf{nat}\,.\ \lambda n : \mathbf{nat}\,. \\ & \quad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{s}\ \mathbf{z} \\ & \qquad\qquad \mid\ \mathbf{s}\ n' \Rightarrow \mathit{times}\ x\ (p\ x\ n') \end{aligned}$$

It would make sense to have a given exponent n incorporated and obtain a specialized program only taking x as an argument. When $n = 2$ the program could for instance be specialized into

$$\mathit{power}_2 \quad \equiv \quad \lambda x : \mathbf{nat}\,.\ \mathit{times}\ x\ (\mathit{times}\ x\ (\mathbf{s}\ \mathbf{z}))$$

containing no conditional checks. Here and in the following *times* is assumed to be a closed expression mapping two natural numbers to their product.

Several techniques and approaches have been developed in the area of staged computation. Determining which parts of a program depend on which parts of its input is often of central importance, and to this purpose a number of typed languages with binding-time annotations, as for example Nielson and Nielson's two-level programming language, have been developed. Although most of these type systems have been algorithmically motivated, some work has been done in developing systems, which can be motivated logically.

In this project we will investigate three systems extending the simple functional language Mini-ML with some extra language constructs dedicated for code generation. These extra constructs are motivated by an extension of the Curry-Howard isomorphism to different modal logics and, within this setting we can think of each world in the Kripke semantics as a stage of computation.

When writing programs generating other programs both practicality and safety are relevant parameters. Considering for instance the untyped programming language Scheme, the possibility of writing quasiquoted expressions makes this simple language very well-suited for straightforward writing of programs generating other programs. No guarantees are given though, that either the generating or the generated programs can be executed without getting stuck. Considering instead the strongly-typed ML, it will be less fluent to do metaprogramming by use of appropriately defined datatypes within this language, but at least well-typed metaprograms will be guaranteed to execute without any errors. In the cross-stage type systems of the modal-logically based systems to be treated in this project, programs generated by well-typed metaprograms will in fact also be well-typed, while allowing some of the syntactical conveniences of Scheme to be maintained.

1.2 Metatheory of metaprogramming

A *metatheory* can be understood as a theory having another theory as its subject, and the metatheory of a programming language can thus be seen as the properties satisfied by the given programming language. Some of these properties might be described in theorems within the metatheory and are then called *metatheorems* about the programming language.

As for programming languages in general there are certain properties that metaprogramming languages should fulfill in order to be considered healthy. For instance we want the operational semantics of the languages to be executable as well as uniquely specified in order to ensure deterministic execution of programs. Also we want that well-typed program-generating programs can be executed without getting stuck. This can be guaranteed if the operational semantics satisfy progress and type preservation. Informally progress here covers the property that either the program is fully evaluated or else an evaluation step can be taken.

Of course it is also relevant to consider the fitness of code generated by a program-generating program. For those metaprogramming languages facilitating execution of generated code within a metaprogram itself, it is obvious that the properties of the metaprogram, like for instance type preservation, must also go for the generated code.

As it is not possible to find much work attempting to fully formalize and machine-verify the metatheory of metaprogramming languages, there is no real evidence whether this is easily done or not. The purpose of this thesis is to investigate if certain multi-stage programming languages can be formalized and verified within the frameworks normally used for reasoning about languages facilitating single-stage programming. A significant concern will be the impact of constructs binding variables across execution stages.

1.3 Thesis outline

The next chapter, **Chapter 2**, contains a systematic introduction of formalization tools which are normally geared for single-stage programming languages but which we in later chapters will try to adapt to multi-stage programming languages. The tools included are the Edinburgh Logical Framework, the programming language Elf based on this framework, and last but not least the metalogical framework Twelf providing facilities enabling automated verification of metatheorems expressed as Elf programs.

The core language of the metaprogramming languages to be considered in this project is the single-stage functional language Mini-ML. A formalization of Mini-ML within the Edinburgh Logical Framework can be found in for example [10] but, as it is customary, maintenance of variable uniqueness is to some extent left implicit there. In preparation for reasoning about languages allowing multiple computation stages where α-equivalence might not be as obviously defined as in Mini-ML, **Chapter 3** contains a formalization of Mini-ML with an extra careful treatment of bound variables.

The first of the three metaprogramming systems to be scrutinized in this project is introduced in **Chapter 4**. The system is based on modal necessitation and seems to be the earliest example of a logic-based metaprogramming system. Two languages are comprised in the system: the explicit language Mini-ML$^{\Box}_{\text{ex}}$ and the implicit language Mini-ML$^{\Box}$. Within Mini-ML$^{\Box}_{\text{ex}}$ code operations and staging of computation are represented very explicitly, whereas Mini-ML$^{\Box}$ provides a more pragmatic way of programming staged computation. The two languages are related through a type preserving translation from Mini-ML$^{\Box}$ to Mini-ML$^{\Box}_{\text{ex}}$, and one of the main challenges of this project is to have this translation represented within our logical framework and have its correctness Twelf-verified. Before we get to the translation we solve the other important problem of finding adequate representations of the two multi-stage languages themselves. Also the operational semantics for Mini-ML$^{\Box}_{\text{ex}}$ will be represented and its determinacy verified within Twelf.

In **Chapter 5** another metaprogramming system is introduced. This second system is based on an extension of the Curry-Howard isomorphism to include temporal logic, and its programming language is named Mini-ML$^{\bigcirc}$. Whereas code containing free variables is not accepted in the programming languages of the previous chapter, the language Mini-ML$^{\bigcirc}$ does allow for code where variables occur freely. An advantage of this is the possibility of generating code containing fewer residual β-redexes, but unfortunately it also implies that generated code can no longer be safely executed within the metaprogram itself. In the chapter we will see how experience gained in the previous chapter can be exploited in the formalization of Mini-ML$^{\bigcirc}$ within our logical framework. Again operational semantics will be represented and evaluation determinacy Twelf-verified.

The subject of **Chapter 6** is a third and final metaprogramming system. This last system is motivated by a contextual modal logic which is a direct generalization of the modal logic of necessity supporting the metaprogramming system described in **Chapter 4**. The generalized system seems to provide the best of the two previously described systems: code without a lot of superfluous λ-abstractions can be generated, and also code can be both generated and executed within the same cycle of operations. We will see how the formalization techniques of the previous chapters can be naturally extended to the formalization of this system. Also

we will define and represent both big-step and small-step operational semantics for Mini-$\mathrm{ML}_{\mathrm{co}}$, and we will have the equivalence of these verified within Twelf. Finally we enlarge the formulated metatheory of the system by formulating and verifying that well-typed expressions never get stuck.

In **Chapter 7** we conclude on the project, and in **Appendix A** all the Twelf code can be found.

Chapter 2

Formalization Tools

When dealing with deductive systems (logics, type systems, operational semantics, or similar) we are often interested in doing actions like constructing derivations or checking proofs against the given specification. It would therefore be advantageous if we did not have to develop the setup for those kinds of actions from scratch every time we encountered a new deductive system. Instead we should be able to translate the given system into some uniform representation for which the appropriate tools had been developed once and for all. This is exactly the idea of a *logical framework*: it is a metasystem facilitating uniform representation of a wide range of deductive systems.

The Edinburgh Logical Framework (LF) is a logical framework which has been widely used for formalization of single-stage programming languages. In this project we intend to find out how well LF also embraces a class of deductive systems defining multi-stage computation. Do constructs binding variables across execution stages need special treatment?

The rest of this chapter is for later reference and is opened with a description of LF in Section **2.1**. The section that follows, Section **2.2**, contains an introduction of Elf as a logical programming language based on LF, and we will see how this language enables us to do things like proof-search in a Prolog-like style. Twelf extends Elf with some metalogical features valuable in verification of metatheoretical properties of a formalized object language and will be introduced in Section **2.3**.

2.1 The Edinburgh Logical Framework

This section contains a description of LF and its essential metatheoretic properties needed for proving representation adequacy theorems later on. Further details about the logical framework can be found in for instance [10].

A logical framework provides both a concrete representation language as well as a methodology guiding the translation of deductive systems into the metalanguage of the framework.

2.1.1 Representation language of LF

In LF the language being used to represent a given object language can be characterized as a dependent-typed λ-calculus. The list of syntax categories are [1]:

$$\begin{array}{rrcl} \text{Kinds:} & K & ::= & \mathbf{type} \mid \Pi\, x : A\,.\, K \\ \text{Type families:} & A & ::= & a \mid \Pi\, x : A_1\,.\, A_2 \mid A\ M \\ \text{Objects:} & M & ::= & c \mid x \mid \lambda x : A\,.\, M \mid M_1\ M_2 \end{array}$$

$$\begin{array}{rrcl} \text{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma,\ a : K \mid \Sigma,\ c : A \\ \text{Contexts:} & \Lambda & ::= & \cdot \mid \Lambda,\ x : A \end{array}$$

The concatenation of two contexts Λ_1 and Λ_2 will be written $\boxed{\Lambda_1,\ \Lambda_2}$.

A fully instantiated type family is called a *type*. All types have kind **type** and can informally be defined by the following syntax category

$$\text{Types:} \quad A \ ::= \ a\ M_1 \ldots M_k \mid \Pi\, x : A_1\,.\, A_2$$

where M_i, $i \in \{1, \ldots, k\}$, are called *index objects*, and k is the *arity* of the type family a, i.e. the number of applications it takes before the constant is fully applied.

Also, a dependent type $\Pi\, x : A_1\,.\, A_2$ where x does not occur freely in A_2 can be abbreviated using normal function style as $\boxed{A_1 \to A_2}$. Similarly, a dependent kind $\Pi\, x : A\,.\, K$ can be written $\boxed{A \to K}$ when x does not occur freely in K.

The elements of the kinds category are used to classify type families, and types are used to classify objects. This three-level structure is characterized by judgements of the general form

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

which takes on one of the three specific forms

$$\begin{array}{l} \Lambda \vdash^{\mathrm{LF}}_{\Sigma} K \\ \Lambda \vdash^{\mathrm{LF}}_{\Sigma} A\ :\ K \\ \Lambda \vdash^{\mathrm{LF}}_{\Sigma} M\ :\ A \end{array}$$

and respectively asserts that within the valid signature Σ and the valid context Λ, K is a valid kind, A is a valid type family of kind K, and M is a valid object of type A.

[1] In most presentations of LF the metavariable Γ is typically used to range over contexts. To avoid confusion, we have decided to use Λ instead, and reserve Γ for the multi-stage object languages presented later.

The last two syntax classes, signatures and contexts, are for assumption management. Signatures contain declarations of type families, and object constants, and contexts contain variable declarations. Now, before describing the three judgements introduced above any further we need to define what to understand from a valid signature Σ and a valid context Λ. We will not introduce formal judgement forms for this but just explain that a signature can be seen as an always present environment, and thus, to be valid, the involved kinds and types have to be valid assuming nothing but the constants previously defined in the given signature. Furthermore, a signature containing multiple declarations of the same constant is not considered valid. Concerning contexts, a valid context cannot contain more than one type assumption for each variable either, but the type involved is allowed to depend on both constants declared in the given valid signature as well as the assumptions previously defined in the given context. In the following we will always presume validity of the signatures and contexts mentioned unless otherwise explicitly stated.

The detailed rules for valid kinds and type families will not be given here, but as we will need to make several references to some of the rules for valid LF objects later on, we will indeed recite the main inference rules defining the judgement $\boxed{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} M \ : \ A}$:

$$\frac{\Sigma(c) = A}{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} c \ : \ A} \ \text{TP}^{\mathrm{LF}}\text{-TERM-CONST} \qquad \frac{\Lambda(x) = A}{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} x \ : \ A} \ \text{TP}^{\mathrm{LF}}\text{-TERM-VAR}$$

$$\frac{\Lambda,\ x : A_1 \vdash^{\mathrm{LF}}_{\Sigma} M \ : \ A_2}{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} \lambda x : A_1 \,.\, M \ : \ (\Pi\, x : A_1 \,.\, A_2)} \ \text{TP}^{\mathrm{LF}}\text{-TERM-ABS}$$

$$\frac{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} M_1 \ : \ \Pi\, x : A_1 \,.\, A_2 \qquad \Lambda \vdash^{\mathrm{LF}}_{\Sigma} M_2 \ : \ A_1}{\Lambda \vdash^{\mathrm{LF}}_{\Sigma} M_1\ M_2 \ : \ \{M_2/x\}\, A_2} \ \text{TP}^{\mathrm{LF}}\text{-TERM-APP}$$

where $\boxed{\Sigma(c) = A}$ means that $c : A$ is present in the predefined signature Σ and $\boxed{\Lambda(x) = A}$ that $x : A$ is present in the context Γ. In general, $\boxed{\{X/x\}\, Y}$ denotes the substitution of the X for x in Y.

Taking a closer look at the typing rule TP^{LF}-TERM-ABS, it will appear that the rule does not explicitly ensure that context validity is preserved during the context extension taking place. However, as α-equivalence is well-defined for single-stage λ-calculus, the convention is that unique variable names are preserved by, if necessary, performing variable renaming before application of the rule.

It should also be noted that when dependent types are part of the system, it is appropriate to consider certain typings equivalent, namely the ones where the involved types are of the same shape and their possible index objects are pairwise $\beta\eta$-equivalent. However, all inference rules dealing with definitional equality are left out here.

In the next section on methodology it will become clear how the LF signature varies with the choice of object language.

2.1.2 Representation methodology of LF

Having the metalanguage of LF in place we are now ready to take a look at what the guidelines for translation of a given object language into the LF metalanguage are. This is the question of representation methodology.

The principles are easiest illustrated by application to a concrete deductive system, and we choose our example of an object language to be the untyped λ-calculus endowed with a two-ruled call-by-name step-relation. We define the untyped λ-calculus by a syntax category of expressions:

$$\text{Expressions:} \quad e \quad ::= \quad x \mid \lambda x\,.\,e \mid e_1\ e_2$$

The step-relation $\boxed{e \mapsto^{\lambda} e'}$ is defined by the following β-reduction and context reduction rule:

$$\frac{}{(\lambda x\,.\,e_1)\ e_2 \mapsto^{\lambda} \{e_2/x\}\,e_1}\ \text{ST}^{\lambda}\text{-}\beta \qquad \frac{e_1 \mapsto^{\lambda} e_1'}{e_1\ e_2 \mapsto^{\lambda} e_1'\ e_2}\ \text{ST}^{\lambda}\text{-CTX}$$

Now, letting Σ_λ denote the LF signature containing the constant declarations needed for the formalization of our untyped λ-calculus, we let the single syntax category of expressions produce the following type declaration in Σ_λ:

$$\mathbf{exp} \ : \ \mathbf{type}$$

Each expression of the object language should then be translated into an LF object of type **exp**. Considering function application constructions we need an object constant in Σ_λ which generates an object of type **exp** from two other objects of type **exp**, namely:

$$\mathbf{app} \ : \ \mathbf{exp} \ \to \ \mathbf{exp} \ \to \ \mathbf{exp}$$

Then, if we let $\llbracket e \rrbracket^{\lambda}$ denote the function translating an expression e into an LF object M of type **exp**, the translation of an application construct is given by

$$\llbracket e_1\ e_2 \rrbracket^{\lambda} \ = \ \mathbf{app}\ \ \llbracket e_1 \rrbracket^{\lambda}\ \ \llbracket e_2 \rrbracket^{\lambda}$$

The strategy for variables and variable-binding λ-abstractions is not as obvious. The representation function will be defined for those kinds of expressions in the next section on higher-order abstract syntax.

Higher-order abstract syntax

When it comes to representing variables and variable-binders in LF the choice of λ-calculus as our metalanguage has some advantages over for instance ML. If our metalanguage had been ML we would typically define a data type like

```
datatype exp  =  VAR of string
              |  LAM of string * exp
              |  APP of exp * exp
```

and then represent an object expression like $\lambda x\,.\,\lambda y\,.\,x$ by the term

```
LAM("x", LAM("y", VAR("x")))
```

where variables are translated into text strings, losing their variable status. Now, as the metalanguage of LF is based on λ-calculus, we aim at representing variables of the object language by variables of the metalanguage and make use of so called *higher-order abstract syntax* where variable-bindings of the object language are represented by λ-abstractions of the metalanguage. When higher-order abstract syntax is used, useful variable manipulation operations of the object language can be carried over to equivalent operations of the metalanguage. Particularly variable-renaming and capture-avoiding substitution of the object language can be emulated by respectively α-equivalence and β-reduction in the metalanguage.

In the case of our simple λ-calculus the methodology of higher-order-abstract syntax works just fine. Adding the constant declaration

$$\mathbf{lam} \;:\; (\mathbf{exp} \;\rightarrow\; \mathbf{exp}) \;\rightarrow\; \mathbf{exp}$$

to our LF signature Σ_λ and extending the definition of $[\![e]\!]^\lambda$ with

$$\begin{aligned} [\![x]\!]^\lambda &= x \\ [\![\lambda x\,.\,e]\!]^\lambda &= \mathbf{lam}\ (\lambda x\,.\,\ [\![e]\!]^\lambda) \end{aligned}$$

the object expression $\lambda x\,.\,\lambda y\,.\,x$ is adequately represented by the LF object

$$\mathbf{lam}\ (\lambda x\,.\,\mathbf{lam}\ (\lambda y\,.\,x)).$$

As we will investigate further in the coming chapters, the methodology of higher-order abstract syntax unfortunately might not be quite as applicable in the case of multi-staged computation where the scope of variables is possibly within different stages. In situations where we cannot exploit higher-order abstract syntax we will not get variable renaming and substitution for free, but will have to formulate those operations explicitly from scratch.

Judgements-as-types

Having defined the syntax representation for our untyped λ-calculus we now proceed to the representation of its operational semantics, namely the step relation given by the two inference rules ST^{λ}-β and ST^{λ}-CTX. For this we introduce another main representation methodology of LF, namely the so called *judgements-as-types* principle.

The judgements-as-types principle can be summarized by the following diagram:

Object language:	Judgements	Derivations	Inference rules
	$\downarrow$	$\downarrow$	$\downarrow$
Metalanguage:	Types	Objects	Constants

As also suggested by the name, the judgements-as-types principle dictates that judgements in the object language should be translated into types in the metalanguage. Furthermore, derivations become objects, and inference rules are represented by object constants in the LF signature. In this way, building a derivation of a judgement from rules in the object language becomes building an object of a given type from constants in the metalanguage.

A straightforward yet naive way to represent our step judgements according to the judgements-as-types principle would be to add the following three constant declarations to the signature Σ_{λ}:

$$\begin{aligned} \mathbf{step} &: \mathbf{type} \\ \mathbf{step\_beta} &: \mathbf{step} \\ \mathbf{step\_ctx} &: \mathbf{step} \to \mathbf{step} \end{aligned}$$

Here we have a type **step** which should be the type of all translated step judgements and an object constant for each of the two inference rules. Although this seems to follow the rules marked out by the judgements-as-types principle, a lot of important information is left out. For instance consider the well-formed LF object of type **step** given by

$$\mathbf{step\_ctx}\ \mathbf{step\_beta}.$$

Which expressions are related by this step? And is it the representation of a valid step? We have no way to answer these questions from the information present. Luckily the facility of type families and dependent types comes in handy here. We simply replace the three step constant declarations just given by these other three:

$$\begin{aligned} \mathbf{step} &: \mathbf{exp} \to \mathbf{exp} \to \mathbf{type} \\ \mathbf{step\_beta} &: \Pi\, M_1 : (\mathbf{exp} \to \mathbf{exp})\,.\ \Pi\, M_2 : \mathbf{exp}\,. \\ &\qquad \mathbf{step}\ (\mathbf{app}\ (\mathbf{lam}\ M_1)\ M_2)\ (M_1\ M_2) \\ \mathbf{step\_ctx} &: \Pi\, M_1 : \mathbf{exp}\,.\ \Pi\, M_1' : \mathbf{exp}\,.\ \Pi\, M_2 : \mathbf{exp}\,. \\ &\qquad \mathbf{step}\ M_1\ M_1' \to \mathbf{step}\ (\mathbf{app}\ M_1\ M_2)\ (\mathbf{app}\ M_1'\ M_2) \end{aligned}$$

With these declarations only valid steps will have an LF representation and the representation will be given by the following definitions:

$$\left[\!\!\left[\frac{}{(\lambda x\,.\,e_1)\ e_2 \mapsto^{\lambda} \{e_2/x\}\,e_1} \right]\!\!\right]^{\lambda} = \textbf{step\_beta}\ (\lambda x\,.\,[\![e_1]\!]^{\lambda})\ [\![e_2]\!]^{\lambda}$$

$$\left[\!\!\left[\frac{\mathcal{D} :: e_1 \mapsto^{\lambda} e_1'}{e_1\ e_2 \mapsto^{\lambda} e_1'\ e_2} \right]\!\!\right]^{\lambda} = \textbf{step\_ctx}\ [\![e_1]\!]^{\lambda}\ [\![e_1']\!]^{\lambda}\ [\![e_2]\!]^{\lambda}\ [\![\mathcal{D}]\!]^{\lambda}$$

The presence of type families and dependent types in LF also makes it easy to do intrinsic encoding of certain properties. If we for instance imagine simply-typed λ-calculus as our object language it could be convenient to have the type of an expression attached intrinsically on translation to LF instead of handling the typing judgement separately. Intrinsically typed syntax like this could be implemented by having declarations in our LF signature like

$$\begin{aligned} \textbf{tp} &: \textbf{type} \\ \textbf{exp} &: \textbf{tp} \rightarrow \textbf{type} \end{aligned}$$

plus declarations of object constants of dependent types reflecting the typing rules of the object language.

2.1.3 Adequacy of LF representation

When we decide on a representation of a given object language within the metalanguage LF we have to consider the *adequacy* of that representation in order for our reasoning about metatheories of the object language within the logical framework to be reliable. We should be able to verify that each of the entities of the object language we are trying to represent indeed has a valid LF representation. Also we should be able to prove that any *canonical* LF object with respect to the given signature is actually the representation of some appropriate entity from the object language.

In short a canonical LF object can be characterized as an object containing no β-redexes and where all involved variables and object constants are fully applied, that is an object which is in β-normal and η-long form. According to Lemma 2.10 in [10] we can characterize the canonical LF objects in the following way:

Lemma 2.1 (Canonical objects)**.** An object M is canonical with respect to Σ and Λ iff M is of the form

$$\lambda x_1 : A_1 \ldots . \lambda\, x_n : A_n\,.\,\xi\ M_1\ \ldots M_m$$

where m is the arity of the variable or the object constant ξ and where each M_i, $1 \leq i \leq m$, is canonical with respect to Σ and Λ, $x_1 : A_1, \ldots, x_n : A_n$.

It can be proved that all LF objects have a $\beta\eta$-equivalent object in canonical form.

If we turn to our example of untyped λ-calculus, we should be able to prove that for any expression e with free variables among $x_1, \ldots, x_n$ it will be the case that

$$\cdot,\ x_1 : \mathbf{exp}, \ldots,\ x_n : \mathbf{exp} \vdash^{\mathrm{LF}}_{\Sigma_\lambda} [\![e]\!]^\lambda\ :\ \mathbf{exp}.$$

Also it should be provable that for any canonical LF object M with free variables among $x_1, \ldots, x_n$ and where

$$\cdot,\ x_1 : \mathbf{exp}, \ldots,\ x_n : \mathbf{exp} \vdash^{\mathrm{LF}}_{\Sigma_\lambda} M\ :\ \mathbf{exp}$$

we will be able to find a well-formed expression e with free variables among $x_1, \ldots, x_n$ such that[2] $[\![e]\!]^\lambda = M$. These adequacy properties can be proved by induction on the structure of e and M respectively.

Similar adequacy theorems can be formulated for the representation of judgements. Taking for instance the step-relation of our untyped λ-calculus, the derivation of $e \mapsto^\lambda e'$ is represented by an LF object of type $\mathbf{step}\ [\![e]\!]^\lambda\ [\![e']\!]^\lambda$ and for any canonical LF object of type $\mathbf{step}\ M\ M'$ we can find e and e' such that $e \mapsto^\lambda e'$ can be derived and such that $[\![e]\!]^\lambda = M$ and $[\![e']\!]^\lambda = M'$.

2.1.4 Representation of metatheory in LF

Having the syntax and semantics of a given object language adequately represented in LF is just the beginning. At the end, what we really want to reason about is actually the metatheories of the object language. It is therefore necessary to consider also how metatheorems and their proofs can be appropriately represented in LF for later verification by some metalogical tool.

In Section **2.1.2** we saw how judgements-as-types worked as a methodology for representing judgements about syntax in LF. In this section we will see how this methodology also becomes applicable in the case of metatheoretical properties, when these are formulated as *higher-level judgements*. A higher-level judgement can be characterized as a judgement relating derivations of other judgements.

To illustrate the approach of higher-order judgements we again turn to our untyped λ-calculus and consider the representation of the following determinacy property of the step-relation:

Theorem 2.1 (Small-step determinacy)**.** If $e \mapsto^\lambda e'$ and $e \mapsto^\lambda e''$ then $e' = e''$.

Proof. The theorem is proved by induction on the derivation of $e \mapsto^\lambda e'$ and below each of the two step-rules is in turn considered the last rule applied in the derivation.

1. $\textsc{st}^\lambda$-β. In this case e must be of the form $(\lambda x\,.\ e_1)\ e_2$ and, as there is no step-rule for λ-abstractions, $\textsc{st}^\lambda$-β must also be the last rule applied in the derivation of $e \mapsto^\lambda e''$. Thus both e' and e'' are $\{e_2/x\}\, e_1$ and thus they are equal.

[2] Within LF we will take $=$ to include α-equivalence. In cases where explicit equality is needed we will use the symbol $\equiv$.

2. ST$^\lambda$-CTX. In this case e is of the form $e_1\ e_2$ and we can find e_1' such that $e_1 \mapsto^\lambda e_1'$. As there is no step-rule for λ-abstractions, e_1 cannot be a λ-abstraction and thus ST$^\lambda$-CTX must also be the last rule applied in the derivation of $e \mapsto^\lambda e''$. Then we must be able to find e_1'' such that $e_1 \mapsto^\lambda e_1''$ and now applying the induction hypothesis to e_1 we get that $e_1' = e_1''$. When $e_1' = e_1''$ we also have that $e_1'\ e_2 = e_1''\ e_2$ and, as $e' = e_1'\ e_2$ and $e'' = e_1''\ e_2$, this finishes the case. □

We see that the proof is constructive in the sense that a derivation of $e' = e''$ is constructed from the derivations of $e \mapsto^\lambda e'$ and $e \mapsto^\lambda e''$. We can write this as a higher-level judgement

$$\boxed{\begin{array}{c}\mathcal{D}\\ e \mapsto^\lambda e'\end{array} \ + \ \begin{array}{c}\mathcal{D}'\\ e \mapsto^\lambda e''\end{array} \implies \begin{array}{c}\mathcal{E}\\ e' \doteq e''\end{array}}$$

with the following two inference rules:

$$\frac{}{(\lambda x\,.\,e_1)\ e_2 \mapsto^\lambda \{e_2/x\}\,e_1} \ + \ \frac{}{(\lambda x\,.\,e_1)\ e_2 \mapsto^\lambda \{e_2/x\}\,e_1} \implies \frac{}{\{e_2/x\}\,e_1 \doteq \{e_2/x\}\,e_1}\ \text{STEP}^\lambda\text{-DET-}\beta$$

$$\frac{\begin{array}{c}\mathcal{D}\\ e_1 \mapsto^\lambda e_1'\end{array} \ + \ \begin{array}{c}\mathcal{D}'\\ e_1 \mapsto^\lambda e_1''\end{array} \implies \begin{array}{c}\mathcal{E}\\ e_1' \doteq e_1''\end{array}}{\dfrac{\mathcal{D}}{e_1 \mapsto^\lambda e_1'}\ e_2 \ + \ \dfrac{\mathcal{D}'}{e_1 \mapsto^\lambda e_1''}\ e_2 \implies \dfrac{\mathcal{E}}{e_1' \doteq e_1''}\ e_2}\ \text{STEP}^\lambda\text{-DET-CTX}$$

where the equality of two expressions is defined by the axiom

$$\frac{}{e \doteq e}\ \text{EQ-REFL}$$

and where

$$\frac{e_1 \doteq e_1'}{e_1\ e_2 \doteq e_1'\ e_2}$$

is not an inference rule but actually a small lemma. Again, this lemma can be written as a higher-level judgement

$$\boxed{\begin{array}{c}\mathcal{E}\\ e_1 \doteq e_1'\end{array} \implies \begin{array}{c}\mathcal{E}'\\ e_1\ e_2 \doteq e_1'\ e_2\end{array}}$$

and this time the judgement is defined by a single axiom:

$$\frac{}{\dfrac{}{e_1 \doteq e_1} \Longrightarrow \dfrac{}{e_1\ e_2 \doteq e_1\ e_2}}\ \textsc{eq-app-refl}$$

That is, since $e_1 \doteq e_1'$ can only be shown using EQ-REFL, we must have $e_1 = e_1'$, hence $e_1\ e_2 = e_1\ e_2$ and thus $e_1\ e_2 \doteq e_1\ e_2$.

Applying the judgements-as-types principle, all this can be represented in LF by addition of the following declarations to our LF signature Σ_λ:

$$\begin{aligned}
\mathbf{eq} &: \mathbf{exp} \to \mathbf{exp} \to \mathbf{type} \\
\mathbf{eq_refl} &: \Pi M : \mathbf{exp}\,.\ \mathbf{eq}\ M\ M \\
\\
\mathbf{eq_app} &: \Pi M_1 : \mathbf{exp}\,.\ \Pi M_1' : \mathbf{exp}\,.\ \Pi M_2 : \mathbf{exp}\,. \\
&\qquad \mathbf{eq}\ M_1\ M_1' \to \mathbf{eq}\ (\mathbf{app}\ M_1\ M_2)\ (\mathbf{app}\ M_1'\ M_2) \to \mathbf{type} \\
\mathbf{eq_app_refl} &: \Pi M_1 : \mathbf{exp}\,.\ \Pi M_2 : \mathbf{exp}\,. \\
&\qquad \mathbf{eq_app}\ M_1\ M_1\ M_2\ (\mathbf{eq_refl}\ M_1)\ (\mathbf{eq_refl}\ (\mathbf{app}\ M_1\ M_2)) \\
\\
\mathbf{step_det} &: \Pi M : \mathbf{exp}\,.\ \Pi M' : \mathbf{exp}\,.\ \Pi M'' : \mathbf{exp}\,. \\
&\qquad \mathbf{step}\ M\ M' \to \mathbf{step}\ M\ M'' \to \mathbf{eq}\ M'\ M'' \to \mathbf{type} \\
\mathbf{step_det_beta} &: \Pi M_1 : (\mathbf{exp} \to \mathbf{exp})\,.\ \Pi M_2 : \mathbf{exp}\,. \\
&\qquad \mathbf{step_det}\ (\mathbf{app}\ (\mathbf{lam}\ M_1)\ M_2)\ \{M_2/x\}\,M_1\ \{M_2/x\}\,M_1 \\
&\qquad\qquad (\mathbf{step_beta}\ M_1\ M_2)\ (\mathbf{step_beta}\ M_1\ M_2) \\
&\qquad\qquad (\mathbf{eq_refl}\ \{M_2/x\}\,M_1) \\
\mathbf{step_det_ctx} &: \Pi M_1 : \mathbf{exp}\,.\ \Pi M_1' : \mathbf{exp}\,.\ \Pi M_1'' : \mathbf{exp}\,.\ \Pi M_2 : \mathbf{exp}\,. \\
&\qquad \Pi M : (\mathbf{step}\ M_1\ M_1')\,.\ \Pi M' : (\mathbf{step}\ M_1\ M_1'')\,. \\
&\qquad\quad \Pi M'' : (\mathbf{eq}\ M_1'\ M_1'')\,.\ \Pi M''' : (\mathbf{eq}\ (\mathbf{app}\ M_1'\ M_2)\ (\mathbf{app}\ M_1''\ M_2))\,. \\
&\qquad\qquad \mathbf{step_det}\ M_1\ M_1'\ M_1''\ M\ M'\ M'' \to \\
&\qquad\qquad \mathbf{eq_app}\ M_1'\ M_1''\ M_2\ M''\ M''' \to \\
&\qquad\qquad \mathbf{step_det}\ (\mathbf{app}\ M_1\ M_2)\ (\mathbf{app}\ M_1'\ M_2)\ (\mathbf{app}\ M_1''\ M_2) \\
&\qquad\qquad\qquad (\mathbf{step_ctx}\ M_1\ M_1'\ M_2\ M)\ (\mathbf{step_ctx}\ M_1\ M_1''\ M_2\ M') \\
&\qquad\qquad\qquad M'''
\end{aligned}$$

Now, these are nothing but rules representing the judgements of our object language relating derivations, just like we made declarations defining the step-relation earlier. The representation does not in itself deliver any information on whether the inference rules for $\Rightarrow$ cover all

possible input structures or whether the relation $\Rightarrow$ has a well-founded recursive definition. All we have accomplished here is a way to represent meta-theories within the LF framework. An actual verification of validity of the metatheories takes some metalogical tools which we will return to in Section **2.3**.

2.1.5 Some metatheory about LF

As the above description of validity makes clear, signatures and contexts cannot be treated as sets but should rather be seen as ordered lists which cannot be rearranged arbitrarily. In order to be able to perform some sound context manipulations later on we here recall Theorem 2.3 from [10] about weakening, strengthening, substitution, and permutation as four theorems, leaving out the proofs:

Theorem 2.2 (Weakening)**.** If we have that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

and if $\Lambda,\ \Lambda'$ is valid in Σ then also

$$\Lambda,\ \Lambda' \vdash^{\mathrm{LF}}_{\Sigma} \rho\,.$$

Theorem 2.3 (Strengthening)**.** If we have that

$$\Lambda,\ x : A,\ \Lambda' \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

then also

$$\Lambda,\ \Lambda' \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

when $x \notin \mathrm{FV}(\Lambda')$ and $x \notin \mathrm{FV}(\rho)$.

Theorem 2.4 (Substitution)**.** If we have that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma} M \ :\ A$$

and that

$$\Lambda,\ x : A,\ \Lambda' \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

then also

$$\Lambda,\ \{M/x\}\,\Lambda' \vdash^{\mathrm{LF}}_{\Sigma} \{M/x\}\,\rho\,.$$

Theorem 2.5 (Permutation)**.** If we have that

$$\Lambda,\ x_1 : A_1,\ \Lambda',\ x_2 : A_2,\ \Lambda'' \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

then also

$$\Lambda,\ x_2 : A_2,\ \Lambda',\ x_1 : A_1,\ \Lambda'' \vdash^{\mathrm{LF}}_{\Sigma} \rho$$

when $x_1 \notin \mathrm{FV}(\Lambda')$ and $x_1 \notin \mathrm{FV}(A_2)$ and when A_2 is valid in Λ.

$\boxed{\mathrm{FV}(X)}$ in general denotes the set of variables occurring freely in X (including type tags of bound variables).

Informally we can say that weakening is about adding extra assumptions, strengthening is about removing unused assumptions, substitution is about replacing an assumption with its proof, and permutation is about exchanging assumptions within the context.

2.2 The logic programming language Elf

In the previous section we were introduced to the logical framework LF and saw how the syntax, semantics and metatheories of an object language could be uniformly represented in the metalanguage of LF. In this section we take a step further and look into the programming language Elf, which is an implementation of LF, giving the clauses of an LF signature an operational interpretation. For further details about Elf than given here see for instance [8].

The concrete syntax of Elf can be thought of as an ASCII translated version of the syntax of LF endowed with a few extra constructs:

Terms:	*term*	::=	*id*	Constant (c or a) or variable (x)
		|	`{`*id* `:` *term*$_1$`}` *term*$_2$	Π-abstraction ($\Pi\, x : A_1\,.\, A_2$ or $\Pi\, x : A\,.\, K$)
		|	`[`*id* `:` *term*$_1$`]` *term*$_2$	λ-abstraction ($\lambda x : A\,.\, M$)
		|	*term*$_1$ *term*$_2$	Application ($A\ M$ or $M_1\ M_2$)
		|	`type`	**type**
		|	*term*$_1$ `->` *term*$_2$	Simple function type ($A_1 \to A_2$)
		|	*term*$_1$ `<-` *term*$_2$	*term*$_2$ `->` *term*$_1$
		|	`{`*id*`}` *term*	Omitted type ascription
		|	`[`*id*`]` *term*	Omitted type ascription
		|	`_`	Omitted term
		|	*term*$_1$ `:` *term*$_2$	Type ascription
		|	`(`*term*`)`	Grouping
Declarations:	*decl*	::=	*id* `:` *term*`.`	Constant declaration ($a : K$ or $c : A$)
		|	*id* `:` *term*$_1$ `=` *term*$_2$`.`	Alias declaration ($c : A = M$)

Here the LF categories of objects, types, and kinds are all joined into one single category of terms. Joining the three categories simplifies type checking and parsing.

A term *id* standing for a bound variable or constant in Elf can be either upper- or lowercase, whereas a free variable has to be uppercase (an initial `_` is allowed).

2.2.1 Type reconstruction in Elf

In Elf a type reconstruction algorithm is implemented. We will not describe how the reconstruction works in details here but just mention that its presence implies that we can leave out a lot of the Π-quantifications when writing declarations of an LF signature as declarations in Elf. The complete type will then be statically reconstructed.

As an example we use our untyped λ-calculus from before. A concise Elf-version of the signature Σ_λ would be:

```
          exp  :  type.
          abs  :  (exp -> exp) -> exp.
          app  :  exp -> exp -> exp.

         step  :  exp -> exp -> type.
    step_beta  :  step (app (abs E1) E2) (E1 E2).
     step_ctx  :  step E1 E1' -> step (app E1 E2) (app E1' E2).

           eq  :  exp -> exp -> type.
      eq_refl  :  eq E E.

       eq_app  :  eq E1 E1' -> eq (app E1 E2) (app E1' E2) -> type.
  eq_app_refl  :  eq_app eq_refl eq_refl.

      step_det  :  step E E' -> step E E'' -> eq E' E'' -> type.
step_det_beta  :  step_det step_beta step_beta eq_refl.
 step_det_ctx  :  step_det S S' Q ->
                  eq_app Q Q' ->
                  step_det (step_ctx S) (step_ctx S') Q'.
```

Typically, as it also appears from our example, little typing is saved by leaving out the Π-quantifications on representation of object language syntax and operational semantics, whereas a significant amount of keystrokes can often be saved when doing metatheory declarations.

2.2.2 Logic programming interpretation in Elf

In the programming language Elf the declarations of a given LF signature are given a Prolog-like operational interpretation, and execution here becomes a search for closed terms of certain types.

It is appropriate to mention that the logic programming interpretation provided by Elf is not the only possible approach to programming with LF objects as data. Delphin is an example of a programming language making it possible to do functional programming over systems represented in LF [12].

Back-arrows in Elf

Using back-arrow style when declaring LF object constants in Elf the declarations will look more like Prolog clauses, and the logical interpretation will come more natural. Taking for instance our inference rule ST$^{\lambda}$-CTX its representation in Prolog would look something like

```
step(app (E1, E2), app(E1', E2)) :- step(E1, E1').
```

which looks very similar to the back-arrow version of our Elf declaration of `step_ctx`:

```
step_ctx : step (app E1 E2) (app E1' E2)
             <- step E1 E1'.
```

The interpretation of this declaration is: in order to get a step from `app E1 E2` to `app E1' E2` you first have to find a step from `E1` to `E1'`. The readability advantage of the back-arrow style is more prevailing in the case of `step_det_ctx`, where we have more than one subgoal:

```
step_det_ctx : step_det (step_ctx S) (step_ctx S') Q'
                 <- step_det S S' Q
                 <- eq_app Q Q'.
```

This declaration is easily read as: to prove equality of `E'` and `E''` you will first have to exploit determinism of sub-expression evaluation, and then you will have to exploit equality of application constructs.

Term search in Elf

Now let us assume that we are interested in knowing whether the expression $((\lambda x\,.\,x)\ (\lambda x\,.\,x))\ (\lambda x\,.\,x)$ steps to $(\lambda x\,.\,x)\ (\lambda x\,.\,x)$, that is if a derivation of the judgement

$$((\lambda x\,.\,x)\ (\lambda x\,.\,x))\ (\lambda x\,.\,x) \mapsto^{\lambda} (\lambda x\,.\,x)\ (\lambda x\,.\,x)$$

can be found. In Elf this wonder can be posed as the query

```
%query 1 *
D : step (app (app (abs [x] x) (abs [x] x)) (abs [x] x)).
         (app (abs [x] x) (abs [x] x)).
```

telling Elf to search for a closed term `D` of the type

```
step (app (app (abs [x] x) (abs [x] x)) (abs [x] x))
     (app (abs [x] x) (abs [x] x)).
```

In general the prefix of a query declaration is of the form `%query` n k stating that exactly n solutions can be found in no more than k tries. A `*` means that the number is unbounded. The above query thus states that Elf should be able to find a valid instantiation of `D` using an unbound number of tries.

How does Elf perform such searches then? Seeing the object constants declared in Elf as term constructors, Elf will try to construct a canonical term of the appropriate type by in turn (depth-first) considering each of the term constructors and when an applicable one is reached Elf will try to construct the necessary closed argument terms. As we saw above, the argument term types will be listed in the order in which the arguments should be constructed, when constructors are declared in back-arrow style.

Turning to our step verification example the first applicable constructor is obviously the one named `step_ctx`. With the instantiation `E1 = app (abs [x] x) (abs [x] x)` and `E1' = abs [x] x` Elf then starts to look for a closed term of type `step E1 E1'` to be passed as an argument to `step_ctx`. The term `step_beta` is a perfect candidate with the instantiation `E1 = E2 = [x] x` and Elf will return `D = step_ctx step_beta` as the requested solution to our query. As no further solutions will materialize doing more searches, the declared query is valid.

Logical variables in Elf

Let us now consider an example of a query involving a not fully instantiated type, namely the query requesting if an expression e can be found such that $(\lambda x\,.\,x)\ (\lambda x\,.\,x)$ steps to e:

```
%query 1 *
D : step (app (abs [x] x) (abs [x] x)) E.
```

Again Elf will try to construct a canonical term of the correct type and along the way it will become clear that a solution can only be found if `E = abs [x] x`. This will be reported together with the valid instantiation of `D` and we will know that $(\lambda x\,.\,x)\ (\lambda x\,.\,x)$ steps to $\lambda x\,.\,x$. Free variables like `E` can be compared with logical variables in Prolog.

In the same way we can make Elf look for an instantiation of `Q` which will make the query

```
%query 1 *
D : step_det step_beta step_beta Q.
```

valid. Obviously `Q = eq_refl` (and `D = step_det_beta`) will work, and the presence of such instantiation proves that determinism is true when the two involved steps are β-reductions. We could repeat the experiment for context-steps and then feel somehow convinced that the step-relation does indeed fulfill determinism. What we really would like, though, was to have the system systematically check whether the type `step_det S S' Q` was inhabited for all relevant combinations of steps `S` and `S'`. This is not part of Elf but is provided by the metalogical framework Twelf introduced below.

2.3 The metalogical framework Twelf

Twelf is a framework extending the programming language Elf with some extra metalogical features making machine verification of metatheories possible. For further details on Twelf than mentioned here see for instance [11].

At the end of the last section on Elf we saw how it made sense to leave the index object `Q` of the type `step_det step_beta step_beta Q` uninstantiated. If Elf could then find a valid instantiation of `Q` we could see it as a proof of determinism in the case of β-reductions. In Twelf such informal metatheory verification can be replaced by a full-blown machine verification. Again taking the determinism property of our step-relation as an example, the property can be considered fully proved if the following code is accepted by the Twelf compiler:

```
step_det : step E E' -> step E E'' -> eq E' E'' -> type.
%mode step_det +S +S' -Q.
step_det_beta : step_det step_beta step_beta eq_refl.
step_det_ctx : step_det (step_ctx S) (step_ctx S') Q'
                <- step_det S S' Q
                <- eq_app Q Q'.
%total S (step_det S _ _).
```

2.3.1 Mode declaration in Twelf

The mode declaration

```
%mode step_det +S +S' -Q.
```

declares that the two step index objects of `step_det` are input arguments (`+`), and the equality proof is an output argument (`-`). With such a mode declaration the type family can be interpreted as a runnable logic program.

2.3.2 Total declaration in Twelf

Further down the above code we find the total declaration

```
%total S (step_det S _ _).
```

which makes Twelf check that for each appropriate instantiation of the input arguments `S` and `S'`, a term `D` and an output argument `Q` can be found such that `D` is of the type `step_det S S' Q`. Here the `S` in the declaration tells Twelf that induction is on the first step-derivation and thus, for the program to terminate, all recursive calls to `step_det` must be on strict subderivations of the original step-derivation.

Chapter 3

Unstaged Computation

All the metaprogramming languages to be considered in this project are extensions of the functional language Mini-ML which is a simply-typed λ-calculus augmented with the central ML constructions: products, conditionals, and recursion. In preparation for reasoning about the languages allowing multiple computation stages we find it appropriate to do a thorough study of how pure Mini-ML is adequately formalized within the logical framework LF. The occurrence of cross-stage variables in staged computation makes it relevant to pay extra close attention to the treatment of variables in the single-stage case. This is the subject of the current chapter.

3.1 Syntax of Mini-ML

The syntax categories of Mini-ML are defined in the following common way:

$$
\begin{array}{rrcl}
\text{Types:} & \tau & ::= & \mathbf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
\text{Expressions:} & e & ::= & x \mid \lambda x : \tau \,.\, e \mid e_1\ e_2 \mid \mathbf{fix}\ x : \tau \,.\, e \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \\
 & & & \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma,\ x : \tau
\end{array}
$$

We will use $\boxed{\Gamma_1,\ \Gamma_2}$ to mean the concatenation of the contexts Γ_1 and Γ_2.

3.2 Context updating in Mini-ML

The syntax definition given above does not explicitly disallow variables to appear more than once in a context Γ. Nor does it provide any suggestions on how to deal with or possibly avoid repeated variable occurrences. Should the lastly added assumption be the one ruling? Or should a new assumption of some variable overwrite any already existing assumption about that variable? As α-equivalence is well-defined in the single-stage language Mini-ML, the common way of handling this context issue is to simply avoid colliding variable names by

doing appropriate variable renaming before any context extensions, just like described with LF in Section **2.1**.

When considering multi-staged computation, α-equivalence might not be as obviously defined as in the case of single-staged computation, and thus the convention of exploiting variable renaming to avoid addition of non-fresh variables to a context is not a tenable approach. Now, as we will indeed need variable uniqueness on translation into LF, allowing multiple entrances for each variable is not a good circumvention. Instead we decide to adopt variable overwriting, in the way described above, as the context update operation of our object language. Furthermore, only contexts with unique variable names will be considered valid, and to simplify wording, such validity will be presumed for any context mentioned unless otherwise explicitly stated.

We define the operation $\boxed{\Gamma\{x \mapsto \tau\}}$ updating the type of x to τ in the context Γ in the following way:

$$\begin{aligned} \cdot\{x \mapsto \tau\} &= \cdot,\ x : \tau \\ (\Gamma,\ x : \tau')\{x \mapsto \tau\} &= \Gamma,\ x : \tau \\ (\Gamma,\ x' : \tau')\{x \mapsto \tau\} &= \Gamma\{x \mapsto \tau\},\ x' : \tau',\ \text{when } x \neq x' \end{aligned}$$

Lemma 3.1. The operation $\Gamma\{x \mapsto \tau\}$ preserves context validity.

Proof. The lemma follows by induction on the structure of Γ. □

3.3 Typing rules of Mini-ML

Incorporating the just defined overwriting methodology into the rule for typing of lambda abstractions as well as **fix** and **case** constructions, the judgement $\boxed{\Gamma \vdash e : \tau}$ assigning the type τ to the expression e will be defined in the following way:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{TP-VAR}$$

$$\frac{\Gamma\{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \,.\, e : \tau_1 \to \tau_2}\ \text{TP-ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{TP-APP}$$

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau}{\Gamma \vdash \mathbf{fix}\ x : \tau \,.\, e : \tau}\ \text{TP-FIX}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \text{TP-PAIR}$$

$$\frac{\Gamma \vdash e \,:\, \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}\ e \,:\, \tau_1}\ \text{TP-FST} \qquad \frac{\Gamma \vdash e \,:\, \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}\ e \,:\, \tau_2}\ \text{TP-SND}$$

$$\frac{}{\Gamma \vdash \mathbf{z} \,:\, \mathbf{nat}}\ \text{TP-ZERO} \qquad \frac{\Gamma \vdash e \,:\, \mathbf{nat}}{\Gamma \vdash \mathbf{s}\ e \,:\, \mathbf{nat}}\ \text{TP-SUCC}$$

$$\frac{\Gamma \vdash e \,:\, \mathbf{nat} \qquad \Gamma \vdash e_1 \,:\, \tau \qquad \Gamma\{x \mapsto \mathbf{nat}\} \vdash e_2 \,:\, \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \,:\, \tau}\ \text{TP-CASE}$$

where $\boxed{\Gamma(x) = \tau}$ means that $x : \tau$ is present in the context Γ.

3.4 Representing well-typed Mini-ML expressions in LF

Deviating from normal practice of variable renaming in the object language Mini-ML by use of assumption overwriting on context updates, we have to verify that Mini-ML can still be adequately represented in the metalanguage LF.

As all lambda abstractions and **fix** constructions are type-annotated, and as the typing rules given above offer no ambiguity, each well-typed expression will have a uniquely given type. Instead of treating the typing judgement separately we are thus able to make use of a handy intrinsic LF syntax where the type of a given well-typed expression is attached intrinsically on translation.

Having decided on using intrinsic syntax the above definition of Mini-ML leads to the following contents of the LF signature Σ_{si} to be used in the representation of Mini-ML:

$$\begin{aligned}
\mathbf{tp} &: \mathbf{type} \\
\mathbf{nat} &: \mathbf{tp} \\
\mathbf{arrow} &: \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp} \\
\mathbf{product} &: \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp}
\end{aligned}$$

$$\begin{aligned}
\mathbf{exp} &: \mathbf{tp} \to \mathbf{type} \\
\mathbf{lam} &: \Pi T_1 : \mathbf{tp}\,.\, \Pi T_2 : \mathbf{tp}\,.\, (\mathbf{exp}\ T_1 \to \mathbf{exp}\ T_2) \to \mathbf{exp}\ (\mathbf{arrow}\ T_1\ T_2) \\
\mathbf{app} &: \Pi T_1 : \mathbf{tp}\,.\, \Pi T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ (\mathbf{arrow}\ T_1\ T_2) \to \mathbf{exp}\ T_1 \to \mathbf{exp}\ T_2 \\
\mathbf{fix} &: \Pi T : \mathbf{tp}\,.\, (\mathbf{exp}\ T \to \mathbf{exp}\ T) \to \mathbf{exp}\ T \\
\mathbf{pair} &: \Pi T_1 : \mathbf{tp}\,.\, \Pi T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ T_1 \to \mathbf{exp}\ T_2 \to \mathbf{exp}\ (\mathbf{product}\ T_1\ T_2) \\
\mathbf{fst} &: \Pi T_1 : \mathbf{tp}\,.\, \Pi T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ T_1 \\
\mathbf{snd} &: \Pi T_1 : \mathbf{tp}\,.\, \Pi T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ T_2 \\
\mathbf{z} &: \mathbf{exp\ nat}
\end{aligned}$$

$$\begin{aligned} \mathbf{s} &: \mathbf{exp\ nat} \to \mathbf{exp\ nat} \\ \mathbf{case} &: \Pi T : \mathbf{tp}\,.\, \mathbf{exp\ nat} \to \mathbf{exp}\ T \to (\mathbf{exp\ nat} \to \mathbf{exp}\ T) \to \mathbf{exp}\ T \end{aligned}$$

Below we accordingly define the functions translating types, contexts and typed expressions into LF objects.

The function $\boxed{[\![\tau]\!]}$ mapping a type τ into an LF object T is defined by

$$\begin{aligned} [\![\mathbf{nat}]\!] &= \mathbf{nat} \\ [\![\tau_1 \to \tau_2]\!] &= \mathbf{arrow}\ [\![\tau_1]\!]\ [\![\tau_2]\!] \\ [\![\tau_1 \times \tau_2]\!] &= \mathbf{product}\ [\![\tau_1]\!]\ [\![\tau_2]\!] \end{aligned}$$

and its adequacy is described in the following two lemmas:

Lemma 3.2 (Type representation adequacy $\to$)**.** For any type τ we have that

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} [\![\tau]\!] : \mathbf{tp}$$

Proof. The proof can be done by induction on the structure of τ. □

Lemma 3.3 (Type representation adequacy $\leftarrow$)**.** Assume that T is a canonical LF object with

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} T : \mathbf{tp}$$

where Λ contains no declarations of constants for building **tp**-typed objects. Then we can find a type τ such that $[\![\tau]\!] = T$.

Proof. The proof can be done by induction on the structure of the canonical object T. □

Also we can prove the following convenient lemma:

Lemma 3.4.

- If $[\![\tau]\!] = \mathbf{nat}$ then $\tau = \mathbf{nat}$.
- If $[\![\tau]\!] = \mathbf{arrow}\ T_1\ T_2$ then we can find τ_1 and τ_2 such that $\tau = \tau_1 \to \tau_2$ and such that $[\![\tau_1]\!] = T_1$ and $[\![\tau_2]\!] = T_2$.
- If $[\![\tau]\!] = \mathbf{product}\ T_1\ T_2$ then we can find τ_1 and τ_2 such that $\tau = \tau_1 \times \tau_2$ and such that $[\![\tau_1]\!] = T_1$ and $[\![\tau_2]\!] = T_2$.

Proof. The lemma can be proved by case study and by use of the above adequacy lemmas. □

The function $\boxed{\llbracket \Gamma \rrbracket}$ mapping a context Γ into an LF context Λ is defined by

$$\begin{array}{rcl} \llbracket \cdot \rrbracket & = & \cdot \\ \llbracket \Gamma,\ x : \tau \rrbracket & = & \llbracket \Gamma \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau \rrbracket \end{array}$$

Lemma 3.5. For a valid Mini-ML context Γ the LF context $\llbracket \Gamma \rrbracket$ will also be valid and it will only contain assumptions of the form $x : \mathbf{exp}\ T$.

Proof. The proof can be done by induction on the structure of Γ. □

Due to the structure of translated contexts the following variation of the permutation property of the logical framework LF can be proved:

Lemma 3.6 (Permutation)**.** If we have that

$$\Lambda,\ x : \mathbf{exp}\ T_1,\ \llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M\ :\ \mathbf{exp}\ T_2$$

then also

$$\Lambda,\ \llbracket \Gamma \rrbracket\ ,\ x : \mathbf{exp}\ T_1\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M\ :\ \mathbf{exp}\ T_2$$

and vice versa.

Proof. The lemma can be proved by use of **Lemma 3.5** and multiple applications of the permutation property of the logical framework, **Theorem 2.5**. □

Using intrinsic encoding of types we will not need a function directly mapping expressions into LF objects but instead we define a function $\boxed{\llbracket \mathcal{F} \rrbracket}$ mapping the typing derivation $\mathcal{F}$ for an expression e into an LF object M such that $\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M\ :\ \mathbf{exp}\ \llbracket \tau \rrbracket$ when $\mathcal{F} :: \Gamma \vdash e\ :\ \tau$:

$$\left\llbracket \frac{\Gamma(x) = \tau}{\Gamma \vdash x\ :\ \tau} \right\rrbracket = x$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma\{x \mapsto \tau_1\} \vdash e\ :\ \tau_2}{\Gamma \vdash \lambda x : \tau\,.\,e\ :\ \tau_1 \to \tau_2} \right\rrbracket = \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F} \rrbracket)$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Gamma \vdash e_1\ :\ \tau_1 \to \tau_2 \qquad \mathcal{F}_2 :: \Gamma \vdash e_2\ :\ \tau_1}{\Gamma \vdash e_1\ e_2\ :\ \tau_2} \right\rrbracket = \mathbf{app}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ \llbracket \mathcal{F}_1 \rrbracket\ \llbracket \mathcal{F}_2 \rrbracket$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma\{x \mapsto \tau\} \vdash e\ :\ \tau}{\Gamma \vdash \mathbf{fix}\ x : \tau\,.\,e\ :\ \tau} \right\rrbracket = \mathbf{fix}\ \llbracket \tau \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau \rrbracket\ .\ \llbracket \mathcal{F} \rrbracket)$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Gamma \vdash e_1\ :\ \tau_1 \qquad \mathcal{F}_2 :: \Gamma \vdash e_2\ :\ \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle\ :\ \tau_1 \times \tau_2} \right\rrbracket = \mathbf{pair}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ \llbracket \mathcal{F}_1 \rrbracket\ \llbracket \mathcal{F}_2 \rrbracket$$

$$\left[\!\left[\frac{\mathcal{F} :: \Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}\ e : \tau_1} \right]\!\right] = \mathbf{fst}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ \llbracket\mathcal{F}\rrbracket$$

$$\left[\!\left[\frac{\mathcal{F} :: \Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}\ e : \tau_2} \right]\!\right] = \mathbf{snd}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ \llbracket\mathcal{F}\rrbracket$$

$$\left[\!\left[\frac{}{\Gamma \vdash \mathbf{z} : \mathbf{nat}} \right]\!\right] = \mathbf{z}$$

$$\left[\!\left[\frac{\mathcal{F} :: \Gamma \vdash e : \mathbf{nat}}{\Gamma \vdash \mathbf{s}\ e : \mathbf{nat}} \right]\!\right] = \mathbf{s}\ \llbracket\mathcal{F}\rrbracket$$

$$\left[\!\left[\frac{\mathcal{F} :: \Gamma \vdash e : \mathbf{nat} \qquad \mathcal{F}_1 :: \Gamma \vdash e_1 : \tau \qquad \mathcal{F}_2 :: \Gamma\{x \mapsto \mathbf{nat}\} \vdash e_2 : \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 : \tau} \right]\!\right]$$
$$= \mathbf{case}\ \llbracket\tau\rrbracket\ \llbracket\mathcal{F}\rrbracket\ \llbracket\mathcal{F}_1\rrbracket\ (\lambda x : \mathbf{exp}\ \mathbf{nat}\,.\ \llbracket\mathcal{F}_2\rrbracket)$$

The adequacy of the selected representation of typed expressions is the subject of the next two theorems.

Theorem 3.1 (Expression representation adequacy $\rightarrow$)**.** If $\mathcal{F} :: \Gamma \vdash e : \tau$ then

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\mathcal{F}\rrbracket : \mathbf{exp}\ \llbracket\tau\rrbracket\,.$$

Proof. The proof is done by induction on the structure of the derivation $\mathcal{F}$. Below the rules TP-VAR, TP-ABS, and TP-APP are in turn considered the last rule applied in $\mathcal{F}$. The cases of the remaining rules can be proved in similar ways and are therefore left out.

1. TP-VAR. In this case $\mathcal{F}$ is of the form

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,.$$

When $\Gamma(x) = \tau$ we must have that $\llbracket\Gamma\rrbracket\,(x) = \mathbf{exp}\ \llbracket\tau\rrbracket$, and thus

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} x : \mathbf{exp}\ \llbracket\tau\rrbracket\,.$$

As $\llbracket\mathcal{F}\rrbracket = x$, we are done.

2. TP-ABS. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}' :: \Gamma\{x \mapsto \tau_1\} \vdash e' : \tau_2}{\Gamma \vdash \lambda x : \tau_1\,.\ e' : \tau_1 \rightarrow \tau_2}\,,$$

and applying the induction hypothesis to $\mathcal{F}'$ we get that

$$\llbracket\Gamma\{x \mapsto \tau_1\}\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\mathcal{F}'\rrbracket : \mathbf{exp}\ \llbracket\tau_2\rrbracket\,.$$

First assuming that the variable x is not already present in Γ we have that $\llbracket \Gamma\{x \mapsto \tau_1\} \rrbracket = \llbracket \cdot,\ x : \tau_1,\ \Gamma \rrbracket$, and as $\llbracket \cdot,\ x : \tau_1,\ \Gamma \rrbracket = \cdot,\ x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ ,\ \llbracket \Gamma \rrbracket$, we have that

$$\cdot,\ x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ ,\ \llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \mathcal{F}' \rrbracket\ :\ \mathbf{exp}\ \llbracket \tau_2 \rrbracket\ .$$

To this we can apply **Lemma 3.6** and get that

$$\llbracket \Gamma \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \mathcal{F}' \rrbracket\ :\ \mathbf{exp}\ \llbracket \tau_2 \rrbracket\ ,$$

and then applying the TP$^{\mathrm{LF}}$-TERM-ABS rule of the logical framework we get that

$$\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket\ :\ \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ \rightarrow\ \mathbf{exp}\ \llbracket \tau_2 \rrbracket\ .$$

As

$$\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \tau_1 \rrbracket\ :\ \mathbf{tp}$$

and

$$\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \tau_2 \rrbracket\ :\ \mathbf{tp}$$

according to **Lemma 3.2**, we can now apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)\ :\ \mathbf{exp}\ (\mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket).$$

As $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket$ and $\llbracket \mathcal{F} \rrbracket = \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)$, the case where x is not present in Γ is done.

Now assuming that x is actually present in Γ we have that Γ is of the form $\Gamma = \Gamma_1,\ x : \tau_1',\ \Gamma_2$, and thus we have that $\Gamma\{x \mapsto \tau_1\} = \Gamma_1,\ x : \tau_1,\ \Gamma_2$ and $\llbracket \Gamma\{x \mapsto \tau_1\} \rrbracket = \llbracket \Gamma_1 \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket$. Having

$$\llbracket \Gamma_1 \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1' \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \mathcal{F}' \rrbracket\ :\ \mathbf{exp}\ \llbracket \tau_2 \rrbracket$$

we can apply **Lemma 3.6** and get that

$$\llbracket \Gamma_1 \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket \mathcal{F}' \rrbracket\ :\ \mathbf{exp}\ \llbracket \tau_2 \rrbracket\ ,$$

and then a list of arguments similar to above brings us to

$$\llbracket \Gamma_1 \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)\ :\ \mathbf{exp}\ (\mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket).$$

Applying the weakening property **Theorem 2.2** and then **Lemma 3.6** we now get that

$$\begin{aligned}\llbracket \Gamma_1 \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1' \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}}& \\ \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)\ :\ \mathbf{exp}\ (\mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket),&\end{aligned}$$

and as $\llbracket \Gamma_1 \rrbracket\ ,\ x : \mathbf{exp}\ \llbracket \tau_1' \rrbracket\ ,\ \llbracket \Gamma_2 \rrbracket = \llbracket \Gamma \rrbracket$, we can rewrite this into

$$\llbracket \Gamma \rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)\ :\ \mathbf{exp}\ (\mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket).$$

Again as $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \mathbf{arrow}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket$ and $\llbracket \mathcal{F} \rrbracket = \mathbf{lam}\ \llbracket \tau_1 \rrbracket\ \llbracket \tau_2 \rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket \tau_1 \rrbracket\ .\ \llbracket \mathcal{F}' \rrbracket)$, we are done.

3. TP-APP. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}_1 :: \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \mathcal{F}_2 :: \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\,.$$

Applying the induction hypothesis to $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively we get that

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\mathcal{F}_1\rrbracket : \mathbf{exp}\ (\mathbf{arrow}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket)$$

and that

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\mathcal{F}_2\rrbracket : \mathbf{exp}\ \llbracket\tau_1\rrbracket\,.$$

Now, as

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\tau_1\rrbracket : \mathbf{tp}$$

and

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \llbracket\tau_2\rrbracket : \mathbf{tp}$$

according to **Lemma 3.2**, we can apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework four times and get that

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} \mathbf{app}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ \llbracket\mathcal{F}_1\rrbracket\ \llbracket\mathcal{F}_2\rrbracket : \mathbf{exp}\ \llbracket\tau_2\rrbracket\,.$$

As $\llbracket\mathcal{F}\rrbracket = \mathbf{app}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ \llbracket\mathcal{F}_1\rrbracket\ \llbracket\mathcal{F}_2\rrbracket$, we are done. □

Theorem 3.2 (Expression representation adequacy ←)**.** If M is a canonical LF object with

$$\llbracket\Gamma\rrbracket \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M : \mathbf{exp}\ \llbracket\tau\rrbracket$$

then we can find a typing derivation $\mathcal{F} :: \Gamma \vdash e : \tau$ such that $\llbracket\mathcal{F}\rrbracket = M$.

Proof. The proof is done by induction on the structure of the canonical object M. According to **Lemma 3.5** there will be no variables in $\llbracket\Gamma\rrbracket$ of function type, and thus we will have no proof cases concerning variable applications. Leaving out the cases involving **fix**, **pair**, **fst**, **snd**, **z**, **s**, and **case**, which can be proved in similar ways as the cases involving **lam** and **app**, we have the following three proof cases:

1. $M = x$. According to the rule TP$^{\mathrm{LF}}$-TERM-VAR we must have that $\llbracket\Gamma\rrbracket\,(x) = \mathbf{exp}\ \llbracket\tau\rrbracket$, and as this is only true when $\Gamma(x) = \tau$ and as

$$\left\llbracket \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \right\rrbracket = x,$$

we are done.

2. $M = \mathbf{lam}\ T_1\ T_2\ (\lambda x : \mathbf{exp}\ T_1\,.\,M')$ where x is not present in $\llbracket\Gamma\rrbracket$. From the rules for typing of LF objects we can derive that $\mathbf{arrow}\ T_1\ T_2 = \llbracket\tau\rrbracket$ and that

$$\llbracket\Gamma\rrbracket\,,\ x : \mathbf{exp}\ T_1 \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M' : \mathbf{exp}\ T_2.$$

Applying **Lemma 3.6** to this last judgement we get that

$$\cdot,\ x : \mathbf{exp}\ T_1,\ \llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M'\ :\ \mathbf{exp}\ T_2,$$

and as according to **Lemma 3.4** we can find types τ_1 and τ_2 such that $\tau = \tau_1 \to \tau_2$ and such that $T_1 = \llbracket\tau_1\rrbracket$ and $T_2 = \llbracket\tau_2\rrbracket$, we can rewrite this into

$$\cdot,\ x : \mathbf{exp}\ \llbracket\tau_1\rrbracket,\ \llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M'\ :\ \mathbf{exp}\ \llbracket\tau_2\rrbracket.$$

As $\cdot,\ x : \mathbf{exp}\ \llbracket\tau_1\rrbracket,\ \llbracket\Gamma\rrbracket = \llbracket\cdot,\ x : \tau_1,\ \Gamma\rrbracket$, and as $\cdot,\ x : \tau_1,\ \Gamma = \Gamma\{x \mapsto \tau_1\}$, when x is not present in Γ, we can again rewrite into

$$\llbracket\Gamma\{x \mapsto \tau_1\}\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M'\ :\ \mathbf{exp}\ \llbracket\tau_2\rrbracket.$$

Now applying the induction hypothesis to M' we get a derivation $\mathcal{F}' :: \Gamma\{x \mapsto \tau_1\} \vdash e' : \tau_2$ such that $\llbracket\mathcal{F}'\rrbracket = M'$, and then applying the TP-ABS rule we get a derivation of $\Gamma \vdash \lambda x : \tau_1 .\ e' : \tau_1 \to \tau_2$. As $\tau_1 \to \tau_2 = \tau$ and as the derivation is represented by the LF object $\mathbf{lam}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ (\lambda x : \mathbf{exp}\ \llbracket\tau_1\rrbracket\ .\ \llbracket\mathcal{F}'\rrbracket)$ which is α-equivalent to M, we are done.

2. $M = \mathbf{app}\ T_1\ T_2\ M_1\ M_2$. From the rules for typing of LF objects we can derive that $T_2 = \llbracket\tau\rrbracket$ and that

$$\llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M_1\ :\ \mathbf{exp}\ (\mathbf{arrow}\ T_1\ \llbracket\tau\rrbracket)$$

and

$$\llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M_2\ :\ \mathbf{exp}\ T_1.$$

As according to **Lemma 3.3** we can find a type τ' such that $\llbracket\tau'\rrbracket = T_1$, and as $\llbracket\tau' \to \tau\rrbracket = \mathbf{arrow}\ \llbracket\tau'\rrbracket\ \llbracket\tau\rrbracket$, we can rewrite these last two judgements into

$$\llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M_1\ :\ \mathbf{exp}\ \llbracket\tau' \to \tau\rrbracket$$

and

$$\llbracket\Gamma\rrbracket\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{si}}} M_2\ :\ \mathbf{exp}\ \llbracket\tau'\rrbracket.$$

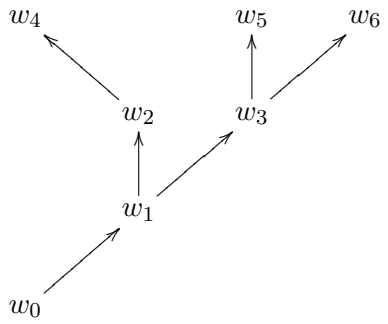
Now applying the induction hypothesis to M_1 and M_2 respectively we get typing derivations $\mathcal{F}_1 :: \Gamma \vdash e_1 : \tau' \to \tau$ and $\mathcal{F}_2 :: \Gamma \vdash e_2 : \tau'$ such that $\llbracket\mathcal{F}_1\rrbracket = M_1$ and $\llbracket\mathcal{F}_2\rrbracket = M_2$. Then by application of the TP-APP rule we get a derivation of $\Gamma \vdash e_1\ e_2 : \tau$, and as this derivation is represented by the LF object $\mathbf{app}\ \llbracket\tau'\rrbracket\ \llbracket\tau\rrbracket\ \llbracket\mathcal{F}_1\rrbracket\ \llbracket\mathcal{F}_2\rrbracket$ which is α-equivalent to M, we are done. □

Chapter 4

Staged Computation and Modal Logic of Necessity

The earliest example of a logic-based metaprogramming system is presented in [2] and in more thorough detail in [3]. In these papers Davies and Pfenning elaborate on how an extension of the Curry-Howard isomorphism to include modal necessity can be used as a foundation for programming languages providing internal support for staged computation.

In this first metaprogramming system computation stages are compared to the worlds of a Kripke model whose accessibility relation obeys that each world has at most one predecessor but can have any number of successors. The following directed graph illustrates an example of such a model:



The necessity operator $\Box$ is for reasoning about universal truth within the Kripke model: the formula $\Box\, A$ is true in some given world iff the formula A will be true in all worlds reachable from there, including the world itself. Thus if $\Box\, A$ can be proved in for instance w_1, then we know that A will be true in both w_1, w_2, ..., and w_6. To actually prove $\Box\, A$ in some current world we should be able to prove A in that world using only knowledge which will remain valid in all future worlds. Let us illustrate by assuming that B, $B \to \Box\, C$, and $\Box\,(C \to A)$ are locally true. From the last of these formulas we know that $C \to A$ will be true in all future worlds. As we can apply implication elimination to the first two formulas and get that $\Box\, C$ is true, we also know that C will be true in all future worlds. Now, from the knowledge that both C and $C \to A$ will be true in all future worlds we can finally derive that A will be true in all future worlds, i.e., that $\Box\, A$ is true.

When doing metaprogramming we would like generated code to be executable in the current as well as in all future stages. Thus its type should not depend on any local assumptions but should be universally valid. By the extension of the Curry-Howard isomorphism to include the necessity modal operator □ we obtain a type constructor tailored to embody these characteristics: an expression having type $\Box\,\tau$ in some given stage represents code which will compute a value of type τ in the current and all reachable stages. By having appropriate modal restrictions of the modal operator □ reflected in the type system it can be guaranteed that any computation connected with the first argument of a function of type $\tau_1 \to \Box\,\tau_2$ can be immediately carried out, leaving residual code of type τ_2.

Above we saw how true formulas of the world w_1 could lead to the knowledge of A being true in the later world w_5. In metaprogramming terminology the transfer of such knowledge between worlds corresponds to transfer of code between stages, and there seems to be different code transfer strategies to chose from. One style of programming can be sketched as preparing for future computation stages along the way by extracting codes from local environments, which we think that we shall need later. The generated codes are then saved within a universally accessible environment, and the local environment of some current stage will no longer be needed when the stage is left. Another style of programming would be to maintain a memory of local environments as we pass through the computation stages, and when the need for some code arises in some computation stage we browse back through the saved environments to see if the conditions for establishment of the requested code were present. Using the first programming style, substitution of code can be said to occur rather explicitly whereas in the second style, transfer of code between computation stages happens more implicitly.

Davies and Pfenning present two typed metaprogramming languages supporting the explicit and the implicit programming style sketched above respectively. The implicit language is considered a more natural programming language than the explicit language, but as reasonings do not go back in time in the later, operational semantics are easilier formulated for this language. Consequently no operational semantics are defined for the implicit language, but instead a type-preserving translation into the explicit language is defined. Below we present the two languages and the translation between them in more detail, and we investigate on adequate representations within the logical framework LF. The correctness of the given translation as well as evaluation determinacy for Mini-ML$^{\Box}_{\text{ex}}$ will also be formalized within LF and machine-verified using the metalogical framework Twelf.

4.1 The explicit □-language

The explicit language is an extension of the functional language Mini-ML and is referred to as Mini-ML$^{\Box}_{\text{ex}}$.

4.1.1 Syntax of Mini-ML$^{\Box}_{\text{ex}}$

The syntax categories of Mini-ML$^{\Box}_{\text{ex}}$ are given by

$$\begin{array}{rrcl} \text{Types:} & \tau & ::= & \mathbf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \Box\,\tau \\ \text{Expressions:} & e & ::= & x \mid u \mid \lambda x : \tau\,.\,e \mid e_1\ e_2 \mid \mathbf{fix}\ x : \tau\,.\,e \mid \langle e_1, e_2 \rangle \mid \end{array}$$

$$\mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \mid \mathbf{box}\ e \mid \mathbf{let}\ \mathbf{box}\ u = e_1\ \mathbf{in}\ e_2$$

$$\text{Modal contexts:} \quad \Delta ::= \cdot \mid \Delta,\ u :: \tau$$
$$\text{Non-modal contexts:} \quad \Gamma ::= \cdot \mid \Gamma,\ x : \tau$$

Here the necessity type $\Box\,\tau$ will be used for classification of code expressions just as we motivated above. Code expressions are constructed using the box-constructor, and code evaluations and substitutions into future computation stages are done by use of let-box-constructions. Assumptions concerning variables containing code are kept in a modal context Δ, which corresponds to the universal environment mentioned above. A non-modal context Γ is what we referred to earlier as a local environment.

We will use $\boxed{\Gamma_1,\ \Gamma_2}$ to mean the concatenation of the non-modal contexts Γ_1 and Γ_2 and similarly $\boxed{\Delta_1,\ \Delta_2}$ to mean the concatenation of the modal contexts Δ_1 and Δ_2.

The function $\boxed{\Gamma\{x \mapsto \tau\}}$ updating a non-modal context with a new assumption is defined just as in the previous chapter, and the function $\boxed{\Delta\{u \mapsto \tau\}}$ updating a modal context with a new assumption is defined similarly.

In the following we presume validity for modal and non-modal contexts in the sense of unique variable names unless otherwise explicitly mentioned.

4.1.2 Typing rules for Mini-ML$^{\Box}_{\text{ex}}$

The typing judgement for Mini-ML$^{\Box}_{\text{ex}}$ has the form $\boxed{\Delta \mid \Gamma \vdash^{\text{ex}} e : \tau}$ and with the assumption overwriting methodology incorporated it is defined by the following inference rules:

$$\frac{\Gamma(x) = \tau}{\Delta \mid \Gamma \vdash^{\text{ex}} x : \tau}\ \text{TP}^{\text{ex}}\text{-VAR-X} \qquad \frac{\Delta(u) = \tau}{\Delta \mid \Gamma \vdash^{\text{ex}} u : \tau}\ \text{TP}^{\text{ex}}\text{-VAR-U}$$

$$\frac{\Delta \mid \Gamma\{x \mapsto \tau_1\} \vdash^{\text{ex}} e : \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \text{TP}^{\text{ex}}\text{-ABS}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{ex}} e_1 : \tau_1 \to \tau_2 \qquad \Delta \mid \Gamma \vdash^{\text{ex}} e_2 : \tau_1}{\Delta \mid \Gamma \vdash^{\text{ex}} e_1\ e_2 : \tau_2}\ \text{TP}^{\text{ex}}\text{-APP}$$

$$\frac{\Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\text{ex}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{fix}\ x : \tau . e : \tau}\ \text{TP}^{\text{ex}}\text{-FIX}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{ex}} e_1 : \tau_1 \qquad \Delta \mid \Gamma \vdash^{\text{ex}} e_2 : \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \text{TP}^{\text{ex}}\text{-PAIR}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{ex}} e \;:\; \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{fst}\; e \;:\; \tau_1}\ \text{TP}^{\text{ex}}\text{-FST} \qquad \frac{\Delta \mid \Gamma \vdash^{\text{ex}} e \;:\; \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{snd}\; e \;:\; \tau_2}\ \text{TP}^{\text{ex}}\text{-SND}$$

$$\frac{}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{z} \;:\; \mathbf{nat}}\ \text{TP}^{\text{ex}}\text{-ZERO} \qquad \frac{\Delta \mid \Gamma \vdash^{\text{ex}} e \;:\; \mathbf{nat}}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{s}\; e \;:\; \mathbf{nat}}\ \text{TP}^{\text{ex}}\text{-SUCC}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{ex}} e \;:\; \mathbf{nat} \qquad \Delta \mid \Gamma \vdash^{\text{ex}} e_1 \;:\; \tau \qquad \Delta \mid \Gamma\{x \mapsto \mathbf{nat}\} \vdash^{\text{ex}} e_2 \;:\; \tau}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{case}\; e\; \mathbf{of}\; \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\; x \Rightarrow e_2 \;:\; \tau}\ \text{TP}^{\text{ex}}\text{-CASE}$$

$$\frac{\Delta \mid \cdot \vdash^{\text{ex}} e \;:\; \tau}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{box}\; e \;:\; \Box\tau}\ \text{TP}^{\text{ex}}\text{-BOX}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{ex}} e_1 \;:\; \Box\tau_1 \qquad \Delta\{u \mapsto \tau_1\} \mid \Gamma \vdash^{\text{ex}} e_2 \;:\; \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{let\ box}\; u = e_1\; \mathbf{in}\; e_2 \;:\; \tau_2}\ \text{TP}^{\text{ex}}\text{-LET-BOX}$$

where $\boxed{\Gamma(x) = \tau}$ means that $x : \tau$ is present in the non-modal context Γ and $\boxed{\Delta(u) = \tau}$ means that $u :: \tau$ is present in the modal context Δ.

The empty non-modal context at the top of the TP$^{\text{ex}}$-BOX rule reflects that entering a box-construct can be seen as entering a new computation stage. Here the "universal" context Δ remains accessible though, just as it was described in the introduction. New assumptions are only added to Δ in the rule TP$^{\text{ex}}$-LET-BOX.

Having both syntax and typing rules in place it is now appropriate to illustrate the nature of Mini-ML$^{\Box}_{\text{ex}}$ by reformulating the proof of $\Box\, A$ given in the introduction as a typing derivation within Mini-ML$^{\Box}_{\text{ex}}$:

$$\dfrac{\dfrac{\dfrac{\Gamma_0(f) = B \to \Box\, C}{\cdot \mid \Gamma_0 \vdash^{\text{ex}} f \;:\; B \to \Box\, C} \qquad \dfrac{\Gamma_0(x) = B}{\cdot \mid \Gamma_0 \vdash^{\text{ex}} x \;:\; B}}{\cdot \mid \Gamma_0 \vdash^{\text{ex}} f\; x \;:\; \Box\, C} \qquad \dfrac{\Gamma_0(g) = \Box\,(C \to A)}{\Delta_0 \mid \Gamma_0 \vdash^{\text{ex}} g \;:\; \Box\,(C \to A)} \qquad \dfrac{\dfrac{\Delta_1(v) = C \to A}{\Delta_1 \mid \cdot \vdash^{\text{ex}} v \;:\; C \to A} \qquad \dfrac{\Delta_1(u) = C}{\Delta_1 \mid \cdot \vdash^{\text{ex}} u \;:\; C}}{\Delta_1 \mid \cdot \vdash^{\text{ex}} v\; u \;:\; A}}{\Delta_1 \mid \Gamma_0 \vdash^{\text{ex}} \mathbf{box}\; (v\; u) \;:\; \Box\, A}}{\Delta_0 \mid \Gamma_0 \vdash^{\text{ex}} \mathbf{let\ box}\; v = g\; \mathbf{in\ box}\; (v\; u) \;:\; \Box\, A}}{\cdot \mid \Gamma_0 \vdash^{\text{ex}} \mathbf{let\ box}\; u = f\; x\; \mathbf{in\ let\ box}\; v = g\; \mathbf{in\ box}\; (v\; u) \;:\; \Box\, A}$$

where

$$\begin{aligned} \Gamma_0 &= \cdot,\ x : B,\ f : B \to \Box\, C,\ g : \Box\,(C \to A) \\ \Delta_0 &= \cdot,\ u :: C \\ \Delta_1 &= \cdot,\ v :: C \to A,\ u :: C \end{aligned}$$

4.1.3 Representing well-typed Mini-ML$^{\Box}_{\text{ex}}$ expressions in LF

Before proceeding to the operational semantics and metatheorems we will investigate on how the syntax of Mini-ML$^{\Box}_{\text{ex}}$ can be adequately represented within our logical framework LF.

Just as in the single-staged case in **Chapter 3** we can, due to typing uniqueness, make use of an intrinsic LF syntax where the type of a well-typed explicit expression is attached intrinsically on translation. In this way only well-typed expressions are represented in LF, and we will not have to consider the typing judgement separately.

Naive LF representation

An obvious representation strategy is obtained by straightforwardly extending the strategy presented in **Chapter 3** with the new modal cases:

The first version of the LF signature Σ_{ex} to be used in the LF representation of Mini-ML$^{\Box}_{\text{ex}}$ extends Σ_{si} with the following three constants:

$$\begin{array}{rcl} \mathbf{code} & : & \mathbf{tp} \rightarrow \mathbf{tp} \\ \\ \mathbf{box} & : & \Pi\, T : \mathbf{tp}\,.\, \mathbf{exp}\ T \rightarrow \mathbf{exp}\ (\mathbf{code}\ T) \\ \mathbf{let\_box} & : & \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,. \\ & & \quad \mathbf{exp}\ (\mathbf{code}\ T_1) \rightarrow (\mathbf{exp}\ T_1 \rightarrow \mathbf{exp}\ T_2) \rightarrow \mathbf{exp}\ T_2 \end{array}$$

The function $\boxed{[\![\tau]\!]^{\text{ex}}}$ mapping an explicit type τ into an LF object T is then defined straightforwardly by

$$\begin{array}{rcl} [\![\mathbf{nat}]\!]^{\text{ex}} & = & \mathbf{nat} \\ [\![\tau_1 \rightarrow \tau_2]\!]^{\text{ex}} & = & \mathbf{arrow}\ [\![\tau_1]\!]^{\text{ex}}\ [\![\tau_2]\!]^{\text{ex}} \\ [\![\tau_1 \times \tau_2]\!]^{\text{ex}} & = & \mathbf{product}\ [\![\tau_1]\!]^{\text{ex}}\ [\![\tau_2]\!]^{\text{ex}} \\ [\![\Box\, \tau]\!]^{\text{ex}} & = & \mathbf{code}\ [\![\tau]\!]^{\text{ex}} \end{array}$$

This straightforward type representation can be proved adequate just like in **Chapter 3**. A lemma similar to **Lemma 3.4** can also be proved.

At this point the function $[\![\Gamma]\!]^{\text{ex}}$ mapping a non-modal context Γ into an LF context Λ is defined in the exact same way as $[\![\Gamma]\!]$ in **Chapter 3**, and the function $[\![\Delta]\!]^{\text{ex}}$ mapping a modal context Δ into an LF context Λ follows the same pattern:

$$\begin{array}{rcl} [\![\cdot]\!]^{\text{ex}} & = & \cdot \\ [\![\Delta,\ u :: \tau]\!]^{\text{ex}} & = & [\![\Delta]\!]^{\text{ex}},\ u : \mathbf{exp}\ [\![\tau]\!]^{\text{ex}} \end{array}$$

Finally, the function $[\![\mathcal{F}]\!]^{\mathrm{ex}}$ naively mapping a typing derivation $\mathcal{F}$ for an explicit expression e into an LF object M such that $[\![\Delta]\!]^{\mathrm{ex}}$, $[\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M \ : \ \mathbf{exp} \ [\![\tau]\!]^{\mathrm{ex}}$ when $\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e \ : \ \tau$ is defined similar to $[\![\mathcal{L}]\!]$ besides for the following extra modal cases:

$$\left[\!\!\left[\frac{\Delta(u) = \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} u \ : \ \tau} \right]\!\!\right]^{\mathrm{ex}} = u$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \cdot \vdash^{\mathrm{ex}} e \ : \ \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{box}\, e \ : \ \Box\tau} \right]\!\!\right]^{\mathrm{ex}} = \mathbf{box} \ [\![\tau]\!]^{\mathrm{ex}} \ [\![\mathcal{F}]\!]^{\mathrm{ex}}$$

$$\left[\!\!\left[\frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1 \ : \ \Box\tau_1 \qquad \mathcal{F}_2 :: \Delta\{u \mapsto \tau_1\} \mid \Gamma \vdash^{\mathrm{ex}} e_2 \ : \ \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \ : \ \tau_2} \right]\!\!\right]^{\mathrm{ex}} =$$

$$\mathbf{let\_box} \ [\![\tau_1]\!]^{\mathrm{ex}} \ [\![\tau_2]\!]^{\mathrm{ex}} \ [\![\mathcal{F}_1]\!]^{\mathrm{ex}} \ (\lambda u : \mathbf{exp} \ [\![\tau_1]\!]^{\mathrm{ex}} \,.\, [\![\mathcal{F}_2]\!]^{\mathrm{ex}})$$

With these straight forward representation functions each well-typed expression in Mini-ML$^{\Box}_{\mathrm{ex}}$ corresponds to a well-typed LF object. However, not every well-typed LF object is the representation of some well-typed Mini-ML$^{\Box}_{\mathrm{ex}}$ expression and therefore adequacy is violated. Take for instance the well-typed LF object appearing in this judgement:

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \mathbf{lam} \ T \ (\mathbf{code}\, T) \ (\lambda x : T \,.\, (\mathbf{box} \ T \ x)) \ : \ \mathbf{exp} \ T. \tag{4.1}$$

The reason why (4.1) is derivable in the given setting is the fact that the variable x remains accessible in the LF context during typing of the body of the box expression. This is opposed to what happens within the object language Mini-ML$^{\Box}_{\mathrm{ex}}$ where the body of a box expression is typed in an empty non-modal context. To prevent LF objects like the one in (4.1) we also have to model the kind of assumption disappearance occurring in TP$^{\mathrm{ex}}$-BOX. As assumptions cannot be literally removed in LF we instead need to find a way to make some assumptions in the LF context inaccessible at certain points. Below we will see how this can be done by introduction of the notion of worlds.

LF representation using worlds

We try to solve the problem described above by making it explicit on translation in which of the worlds of the underlying Kripke model the variables occurring freely within a given expression should be found.

The current contents of the LF signature Σ_{ex} are replaced with the following constant declarations:

$$\mathbf{world} \ : \ \mathbf{type}$$

$$\mathbf{tp} \ : \ \mathbf{type}$$

$$
\begin{array}{rcl}
\mathbf{nat} & : & \mathbf{tp} \\
\mathbf{arrow} & : & \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp} \\
\mathbf{product} & : & \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp} \\
\mathbf{code} & : & \mathbf{tp} \to \mathbf{tp}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{exp} & : & \mathbf{world} \to \mathbf{tp} \to \mathbf{type} \\
\mathbf{lam} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,. \\
 & & \quad (\mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2) \to \mathbf{exp}\ W\ (\mathbf{arrow}\ T_1\ T_2) \\
\mathbf{app} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ (\mathbf{arrow}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2 \\
\mathbf{fix} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,.\, (\mathbf{exp}\ W\ T \to \mathbf{exp}\ W\ T) \to \mathbf{exp}\ W\ T \\
\mathbf{pair} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2 \to \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \\
\mathbf{fst} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_1 \\
\mathbf{snd} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,.\, \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_2 \\
\mathbf{z} & : & \Pi\, W : \mathbf{world}\,.\, \mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{s} & : & \Pi\, W : \mathbf{world}\,.\, \mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{case} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ T \to (\mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ T) \to \\
 & & \quad\ \mathbf{exp}\ W\ T \\
\mathbf{box} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,. \\
 & & \quad (\Pi\, W' : \mathbf{world}\,.\, \mathbf{exp}\ W'\ T) \to \mathbf{exp}\ W\ (\mathbf{code}\ T) \\
\mathbf{let\_box} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T_1 : \mathbf{tp}\,.\, \Pi\, T_2 : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ (\mathbf{code}\ T_1) \to \\
 & & \quad\ ((\Pi\, W' : \mathbf{world}\,.\, \mathbf{exp}\ W'\ T_1) \to \mathbf{exp}\ W\ T_2) \to \mathbf{exp}\ W\ T_2
\end{array}
$$

The function $\boxed{[\![\Delta]\!]^{\mathrm{ex}}}$ mapping a modal context Δ into an LF context Λ now becomes

$$
\begin{array}{rcl}
[\![\cdot]\!]^{\mathrm{ex}} & = & \cdot \\
[\![\Delta,\ u :: \tau]\!]^{\mathrm{ex}} & = & [\![\Delta]\!]^{\mathrm{ex}},\ u : (\Pi\, W : \mathbf{world}\,.\, \mathbf{exp}\ W\ \ [\![\tau]\!]^{\mathrm{ex}})
\end{array}
$$

and the function $\boxed{[\![\Gamma]\!]^{\mathrm{ex}}}$ mapping a non-modal context Γ into an LF context Λ is redefined as

$$\begin{aligned}\llbracket \cdot \rrbracket^{\mathrm{ex}} &= \cdot,\ \mathbf{w} : \mathbf{world} \\ \llbracket \Gamma,\ x : \tau \rrbracket^{\mathrm{ex}} &= \llbracket \Gamma \rrbracket^{\mathrm{ex}},\ x : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\mathrm{ex}}\end{aligned}$$

Finally the function $\boxed{\llbracket \mathcal{F} \rrbracket^{\mathrm{ex}}}$ mapping the typing derivation $\mathcal{F}$ for an explicit expression e into an LF object M such that $\llbracket \Delta \rrbracket^{\mathrm{ex}},\ \llbracket \Gamma \rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\mathrm{ex}}$ when $\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \tau$ is now defined by:

$$\left\llbracket \frac{\Gamma(x) = \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} x : \tau} \right\rrbracket^{\mathrm{ex}} = x$$

$$\left\llbracket \frac{\Delta(u) = \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} u : \tau} \right\rrbracket^{\mathrm{ex}} = u\ \mathbf{w}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma\{x \mapsto \tau_1\} \vdash^{\mathrm{ex}} e : \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \lambda x : \tau\,.\, e : \tau_1 \to \tau_2} \right\rrbracket^{\mathrm{ex}} =$$

$$\mathbf{lam}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\ \llbracket \tau_2 \rrbracket^{\mathrm{ex}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\,.\, \llbracket \mathcal{F} \rrbracket^{\mathrm{ex}})$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1 : \tau_1 \to \tau_2 \qquad \mathcal{F}_2 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_2 : \tau_1}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1\ e_2 : \tau_2} \right\rrbracket^{\mathrm{ex}} =$$

$$\mathbf{app}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\ \llbracket \tau_2 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F}_1 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F}_2 \rrbracket^{\mathrm{ex}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\mathrm{ex}} e : \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{fix}\ x : \tau\,.\, e : \tau} \right\rrbracket^{\mathrm{ex}} = \mathbf{fix}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\mathrm{ex}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\mathrm{ex}}\,.\, \llbracket \mathcal{F} \rrbracket^{\mathrm{ex}})$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1 : \tau_1 \qquad \mathcal{F}_2 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_2 : \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \right\rrbracket^{\mathrm{ex}} =$$

$$\mathbf{pair}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\ \llbracket \tau_2 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F}_1 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F}_2 \rrbracket^{\mathrm{ex}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{fst}\ e : \tau_1} \right\rrbracket^{\mathrm{ex}} = \mathbf{fst}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\ \llbracket \tau_2 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{ex}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{snd}\ e : \tau_2} \right\rrbracket^{\mathrm{ex}} = \mathbf{snd}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\mathrm{ex}}\ \llbracket \tau_2 \rrbracket^{\mathrm{ex}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{ex}}$$

$$\left\llbracket \frac{}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{z} : \mathbf{nat}} \right\rrbracket^{\mathrm{ex}} = \mathbf{z}\ \mathbf{w}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \mathbf{nat}}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{s}\ e : \mathbf{nat}} \right\rrbracket^{\mathrm{ex}} = \mathbf{s}\ \mathbf{w}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{ex}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \mathbf{nat} \quad \mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1 : \tau \quad \mathcal{F}_2 :: \Delta \mid \Gamma\{x \mapsto \mathbf{nat}\} \vdash^{\mathrm{ex}} e_2 : \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 : \tau} \right\rrbracket^{\mathrm{ex}}$$

$$= \mathbf{case}\ \mathbf{w}\ \ [\![\tau]\!]^{\mathrm{ex}}\ \ [\![\mathcal{F}]\!]^{\mathrm{ex}}\ \ [\![\mathcal{F}_1]\!]^{\mathrm{ex}}\ \ (\lambda x : \mathbf{exp}\ \mathbf{w}\ \mathbf{nat}\,.\ [\![\mathcal{F}_2]\!]^{\mathrm{ex}})$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \cdot\ \vdash^{\mathrm{ex}} e\ :\ \tau}{\Delta \mid \Gamma\ \vdash^{\mathrm{ex}} \mathbf{box}\ e\ :\ \Box\tau} \right]\!\!\right]^{\mathrm{ex}} = \mathbf{box}\ \mathbf{w}\ \ [\![\tau]\!]^{\mathrm{ex}}\ \ (\lambda \mathbf{w} : \mathbf{world}\,.\ [\![\mathcal{F}]\!]^{\mathrm{ex}})$$

$$\left[\!\!\left[\frac{\mathcal{F}_1 :: \Delta \mid \Gamma\ \vdash^{\mathrm{ex}} e_1\ :\ \Box\tau_1 \qquad \mathcal{F}_2 :: (\Delta,\ u :: \tau_1) \mid \Gamma\ \vdash^{\mathrm{ex}} e_2\ :\ \tau_2}{\Delta \mid \Gamma\ \vdash^{\mathrm{ex}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2\ :\ \tau_2} \right]\!\!\right]^{\mathrm{ex}} =$$

$$\mathbf{let\_box}\ \mathbf{w}\ \ [\![\tau_1]\!]^{\mathrm{ex}}\ \ [\![\tau_2]\!]^{\mathrm{ex}}\ \ [\![\mathcal{F}_1]\!]^{\mathrm{ex}}\ \ (\lambda u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ \ [\![\tau_1]\!]^{\mathrm{ex}})\,.\ [\![\mathcal{F}_2]\!]^{\mathrm{ex}})$$

Before proving that the just defined signature and representation functions are actually adequate we present a definition and two convenient lemmas.

As we will need to refer to the structure of LF contexts generated by translation of Mini-ML$^{\Box}_{\mathrm{ex}}$ environments into LF, we make the following definition:

Definition 4.1. An LF context Λ is said to be Mini-ML$^{\Box}_{\mathrm{ex}}$-*formed* iff it only contains declarations of the form $w : \mathbf{world}$, $x : \mathbf{exp}\ W\ T$ and $u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ T)$.

Then we state that the translation functions actually generate Mini-ML$^{\Box}_{\mathrm{ex}}$-formed LF contexts:

Lemma 4.1. For a modal context Δ and a non-modal context Γ the LF context $[\![\Delta]\!]^{\mathrm{ex}}$, $[\![\Gamma]\!]^{\mathrm{ex}}$ will be Mini-ML$^{\Box}_{\mathrm{ex}}$-formed.

Proof. The lemma can be proved by proving that both $[\![\Delta]\!]^{\mathrm{ex}}$ and $[\![\Gamma]\!]^{\mathrm{ex}}$ are Mini-ML$^{\Box}_{\mathrm{ex}}$-formed. This can be done by induction on the structure of Δ and Γ respectively. □

Finally we extract how the worlds of free variables relate to the world of the LF object within which they occur freely:

Lemma 4.2. Assume that Λ is Mini-ML$^{\Box}_{\mathrm{ex}}$-formed and that M is canonical with

$$\Lambda\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M\ :\ \mathbf{exp}\ W\ T.$$

Then the following two statements are true:

a) If x is a free variable in M of type $\mathbf{exp}\ W'\ T'$ then $W = W'$.

b) If w is a free variable in M of type **world** then $W = w$.

Proof. Both a) and b) can be be proved by induction on the structure of the canonical object M. As the proofs are similar we will only write the one for a) here.

Due to the structure of the Mini-ML$^{\Box}_{\mathrm{ex}}$-formed context Λ there are no variables in the context of function type, and thus we will have no proof cases concerning variable applications. Leaving

out the cases involving **fix**, **pair**, **fst**, **snd**, **z**, **s**, and **case**, which can be proved in similar ways as the cases involving **lam** and **app**, we have the following six proof cases:

1. $M = x$. According to the TP^{LF}-TERM-VAR rule of the logical framework x can only be of type $\mathbf{exp}\ W\ T$ if $\Lambda(x) = \mathbf{exp}\ W\ T$. Also having $\Lambda(x) = \mathbf{exp}\ W'\ T'$ and Λ being valid we can derive that $W = W'$.

2. $M = u\ W''$ where $\Lambda(u) = \Pi\, W''' : \mathbf{world}\,.\,\mathbf{exp}\ W'''\ T$. Obviously x cannot be free in M and thus this case is empty.

3. $M = \mathbf{lam}\ W''\ T_1\ T_2\ (\lambda x' : \mathbf{exp}\ W''\ T_1\,.\,M')$ where x' is not present in Λ. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda,\ x' : \mathbf{exp}\ W\ T_1 \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M'\ :\ \mathbf{exp}\ W\ T_2.$$

When x is free in M it must also be free in M', and as the addition of $x' : \mathbf{exp}\ W\ T_1$ does not ruin Mini-$\text{ML}^{\Box}_{\text{ex}}$-formedness, we can apply the induction hypothesis to M' and get that $W = W'$.

4. $M = \mathbf{app}\ W''\ T_1\ T_1\ M_1\ M_2$. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M_1\ :\ \mathbf{exp}\ W\ T_1$$

and

$$\Lambda \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M_2\ :\ \mathbf{exp}\ W\ T_2.$$

When x is free in M it must also be free in at least one of the two objects M_1 and M_2. Assuming that x is free in M_1 we can apply the induction hypothesis to M_1 and we are done. Similar if x is free in M_2.

5. $M = \mathbf{box}\ W''\ T''\ (\lambda w : \mathbf{world}\,.\,M')$ where w is not present in Λ. From the rules for typing of LF objects we can derive that

$$\Lambda,\ w : \mathbf{world} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M'\ :\ \mathbf{exp}\ w\ T''.$$

When x is free in M it must also be free in M', and as the addition of $w : \mathbf{world}$ does not ruin Mini-$\text{ML}^{\Box}_{\text{ex}}$-formedness, we can apply the induction hypothesis to M' and get that $w = W'$. As W' but not w is present in Λ, this is a contradiction, and thus x cannot be free in either M or M'. The **box**-case is thus an empty case.

6. $M = \mathbf{let\_box}\ W''\ T_1\ T_2\ M_1\ (\lambda u : (\Pi\, W''' : \mathbf{world}\,.\,\mathbf{exp}\ W'''\ T_1)\,.\,M_2)$ where u is not present in Λ. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M_1\ :\ \mathbf{exp}\ W\ (\mathbf{code}\ T_1)$$

and

$$\Lambda,\ u : (\Pi\, W''' : \mathbf{world}\,.\,\mathbf{exp}\ W'''\ T_1) \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} M_2\ :\ \mathbf{exp}\ W\ T_2.$$

When x is free in M it must also be free in at least one of the two objects M_1 and M_2. Assuming that x is free in M_1 we can apply the induction hypothesis to M_1 and we are done. As $u : (\Pi\, W''' : \mathbf{world}\,.\,\mathbf{exp}\ W'''\ T_1)$ does not ruin Mini-$\text{ML}^{\Box}_{\text{ex}}$-formedness, we can also apply the induction hypothesis and reach $W = W'$ if x is free in M_2. $\Box$

What **Lemma 4.2** says is that even though assumptions about earlier worlds may be present in the context, these assumptions cannot be used to show anything about the current world.

Below we now state and prove the two theorems concerning the adequacy of the LF representation function $[\![\mathcal{F}]\!]^{\mathrm{ex}}$.

Theorem 4.1 (Expression representation adequacy →)**.** If $\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \tau$ then

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} [\![\mathcal{F}]\!]^{\mathrm{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau]\!]^{\mathrm{ex}}.$$

Proof. The proof is done by induction on the structure of the derivation $\mathcal{F}$, and thus each of the typing rules should in turn be considered the last rule applied in $\mathcal{F}$. Here we will only write down the cases of TP$^{\mathrm{ex}}$-VAR-U, TP$^{\mathrm{ex}}$-BOX, and TP$^{\mathrm{ex}}$-LET-BOX as the rest of the cases are similar to cases in the proof for **Theorem 3.1**.

1. TP$^{\mathrm{ex}}$-VAR-U. In this case $\mathcal{F}$ is of the form

$$\frac{\Delta(u) = \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} u : \tau}.$$

When $\Delta(u) = \tau$ we have that $[\![\Delta]\!]^{\mathrm{ex}}(u) = \Pi W : \mathbf{world}.\ \mathbf{exp}\ W\ [\![\tau]\!]^{\mathrm{ex}}$, and thus

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} u\ \mathbf{w} : \mathbf{exp}\ \mathbf{w}\ [\![\tau]\!]^{\mathrm{ex}}.$$

As $[\![\mathcal{F}]\!]^{\mathrm{ex}} = u\ \mathbf{w}$, we are done.

2. TP$^{\mathrm{ex}}$-BOX. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}' :: \Delta \mid \cdot \vdash^{\mathrm{ex}} e' : \tau'}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{box}\ e' : \Box\tau'}.$$

Applying the induction hypothesis to $\mathcal{F}'$ we get that

$$[\![\Delta]\!]^{\mathrm{ex}},\ \mathbf{w} : \mathbf{world} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} [\![\mathcal{F}']\!]^{\mathrm{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau']\!]^{\mathrm{ex}}$$

and then applying the TP$^{\mathrm{LF}}$-TERM-ABS rule of the logical framework we get that

$$[\![\Delta]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \lambda\mathbf{w} : \mathbf{world}.\ [\![\mathcal{F}']\!]^{\mathrm{ex}} : \Pi\,\mathbf{w} : \mathbf{world}.\ \mathbf{exp}\ \mathbf{w}\ [\![\tau']\!]^{\mathrm{ex}}.$$

Exploiting the weakening property **Theorem 2.2** we now get that

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \lambda\mathbf{w} : \mathbf{world}.\ [\![\mathcal{F}']\!]^{\mathrm{ex}} : \Pi\,\mathbf{w} : \mathbf{world}.\ \mathbf{exp}\ \mathbf{w}\ [\![\tau']\!]^{\mathrm{ex}},$$

and as $[\![\Gamma]\!]^{\mathrm{ex}}(\mathbf{w}) = \mathbf{world}$ and as

$$[\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} [\![\tau']\!]^{\mathrm{ex}} : \mathbf{tp}$$

due to adequacy of type representation, we can apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \mathbf{box}\ \mathbf{w}\ [\![\tau']\!]^{\mathrm{ex}}\ (\lambda\mathbf{w} : \mathbf{world}.\ [\![\mathcal{F}']\!]^{\mathrm{ex}}) : \mathbf{exp}\ \mathbf{w}\ (\mathbf{code}\ [\![\tau']\!]^{\mathrm{ex}}).$$

As $[\![\tau]\!]^{\text{ex}} = [\![\Box\,\tau']\!]^{\text{ex}} = \mathbf{code}\ [\![\tau']\!]^{\text{ex}}$ and $[\![\mathcal{F}]\!]^{\text{ex}} = \mathbf{box}\ \mathbf{w}\ [\![\tau']\!]^{\text{ex}}\ (\lambda \mathbf{w} : \mathbf{world}\,.\ [\![\mathcal{F}']\!]^{\text{ex}})$, we are done.

3. TP$^{\text{ex}}$-LET-BOX. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\text{ex}} e_1 : \tau_1 \qquad \mathcal{F}_2 :: \Delta\{u \mapsto \tau_1\} \mid \Gamma \vdash^{\text{ex}} e_2 : \tau_2}{\Delta \mid \Gamma \vdash^{\text{ex}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

and applying the induction hypothesis to $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively we get that

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\mathcal{F}_1]\!]^{\text{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{ex}}$$

and

$$[\![\Delta\{u \mapsto \tau_1\}]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\mathcal{F}_2]\!]^{\text{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}.$$

First assuming that the variable u is not already present in Δ we have that $[\![\Delta\{u \mapsto \tau_1\}]\!]^{\text{ex}} = [\![\cdot,\ u :: \tau_1,\ \Delta]\!]^{\text{ex}}$, and as $[\![\cdot,\ u :: \tau_1,\ \Delta]\!]^{\text{ex}} = \cdot,\ u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}),\ [\![\Delta]\!]^{\text{ex}}$, we can rewrite the second judgement into

$$\cdot,\ u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}),\ [\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\mathcal{F}_2]\!]^{\text{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}.$$

To this we can apply a suitable variant of **Lemma 3.6** and get that

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}},\ u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}) \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\mathcal{F}_2]\!]^{\text{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}$$

and by then applying the TP$^{\text{LF}}$-TERM-ABS rule of the logical framework we get that

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} \lambda u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}})\,.\ [\![\mathcal{F}_2]\!]^{\text{ex}} : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}) \to \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}.$$

As

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\tau_1]\!]^{\text{ex}} : \mathbf{tp}$$

and

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\tau_2]\!]^{\text{ex}} : \mathbf{tp}$$

due to adequacy of type representation and as $\mathbf{w} : \mathbf{world} \in [\![\Gamma]\!]^{\text{ex}}$, we can now apply the TP$^{\text{LF}}$-TERM-APP rule of the logical framework five times and get that

$$[\![\Delta]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} \mathbf{let\_box}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{ex}}\ [\![\tau_2]\!]^{\text{ex}}\ [\![\mathcal{F}_1]\!]^{\text{ex}}\ (\lambda u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}})\,.\ [\![\mathcal{F}_2]\!]^{\text{ex}}) : \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}.$$

As $[\![\tau]\!]^{\text{ex}} = [\![\tau_2]\!]^{\text{ex}}$ and $[\![\mathcal{F}]\!]^{\text{ex}} = \mathbf{let\_box}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{ex}}\ [\![\tau_2]\!]^{\text{ex}}\ [\![\mathcal{F}_1]\!]^{\text{ex}}\ (\lambda u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}})\,.\ [\![\mathcal{F}_2]\!]^{\text{ex}})$, the case where u is not present in Δ is done.

Now assuming that u is actually present in Δ we have that Δ is of the form $\Delta = \Delta_1,\ u :: \tau_1,\ \Delta_2$, and thus we have that $\Delta\{u \mapsto \tau_1\} = \Delta_1,\ u :: \tau_1,\ \Delta_2$ and $[\![\Delta\{u \mapsto \tau_1\}]\!]^{\text{ex}} = [\![\Delta_1]\!]^{\text{ex}},\ u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}),\ [\![\Delta_2]\!]^{\text{ex}}$. Having

$$[\![\Delta_1]\!]^{\text{ex}},\ u : (\Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau_1]\!]^{\text{ex}}),\ [\![\Delta_2]\!]^{\text{ex}},\ [\![\Gamma]\!]^{\text{ex}} \vdash^{\text{LF}}_{\Sigma_{\text{ex}}} [\![\mathcal{F}_2]\!]^{\text{ex}} : \mathbf{exp}\ \mathbf{w}\ [\![\tau_2]\!]^{\text{ex}}$$

we can again apply a suitable variant of **Lemma 3.6** and get that

$$\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}},\ u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}) \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}} : \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}.$$

and then applying the TP$^{\mathrm{LF}}$-TERM-ABS rule of the logical framework we get that

$$\begin{aligned}\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \lambda u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}}\ :\\ (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}) \to \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}.\end{aligned}$$

As

$$\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \llbracket\tau_1\rrbracket^{\mathrm{ex}} : \mathbf{tp}$$

and

$$\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \llbracket\tau_2\rrbracket^{\mathrm{ex}} : \mathbf{tp}$$

due to adequacy of type representation and as $\mathbf{w} : \mathbf{world} \in \llbracket\Gamma\rrbracket^{\mathrm{ex}}$, we can now apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework five times and get that

$$\begin{aligned}\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \mathbf{let\_box}\ \mathbf{w}\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}\ \ \llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}}\\ (\lambda u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}}) : \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}.\end{aligned}$$

Applying the weakening property **Theorem 2.2** and a suitable variant of **Lemma 3.6** we get that

$$\begin{aligned}&\llbracket\Delta_1\rrbracket^{\mathrm{ex}},\ u :: (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}),\ \llbracket\Delta_2\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}}\\ &\quad \mathbf{let\_box}\ \mathbf{w}\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}\ \ \llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}}\\ &\qquad (\lambda u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}}) : \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}\end{aligned}$$

which is the same as

$$\begin{aligned}\llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} \mathbf{let\_box}\ \mathbf{w}\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}\ \ \llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}}\\ (\lambda u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}}) : \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}.\end{aligned}$$

As again $\llbracket\tau\rrbracket^{\mathrm{ex}} = \llbracket\tau_2\rrbracket^{\mathrm{ex}}$ and $\llbracket\mathcal{F}\rrbracket^{\mathrm{ex}} = \mathbf{let\_box}\ \mathbf{w}\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}}\ \ \llbracket\tau_2\rrbracket^{\mathrm{ex}}\ \ \llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}}\ \ (\lambda u : (\Pi\, W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \ \llbracket\tau_1\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}})$, we are done. □

Theorem 4.2 (Expression representation adequacy ←)**.** If M is canonical and

$$\llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M : \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau\rrbracket^{\mathrm{ex}}$$

then we can find a typing derivation $\mathcal{F} :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e : \tau$ such that $\llbracket\mathcal{F}\rrbracket^{\mathrm{ex}} = M$.

Proof. The proof is done by induction on the structure of the canonical object M. Due to the structure of the Mini-ML$^{\Box}_{\mathrm{ex}}$-formed context $\llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}}$ (see **Lemma 4.1**) there are no variables in the context of function type, and thus we will have no proof cases concerning variable applications. The cases below are the proof cases which do not have a similar case in the proof of **Theorem 3.1**.

1. $M = u\ W'$ where $[\![\Delta]\!]^{\mathrm{ex}}(u) = \Pi W : \mathbf{world}\,.\ \mathbf{exp}\ W\ T$. For M to have the type $\mathbf{exp}\ \mathbf{w}\ [\![\tau]\!]^{\mathrm{ex}}$ it must be the case that $W' = \mathbf{w}$ and $T = [\![\tau]\!]^{\mathrm{ex}}$. Now, as $[\![\Delta]\!]^{\mathrm{ex}}(u) = \Pi W : \mathbf{world}\,.\ \mathbf{exp}\ W\ [\![\tau]\!]^{\mathrm{ex}}$ only if $\Delta(u) = \tau$ and as

$$\left[\!\!\left[\frac{\Delta(u) = \tau}{\Delta \mid \Gamma \vdash^{\mathrm{ex}} u : \tau} \right]\!\!\right]^{\mathrm{ex}} = u\ \mathbf{w},$$

we are done.

2. $M = \mathbf{box}\ W\ T\ (\lambda w : \mathbf{world}\,.\ M')$ where w is not present in $[\![\Delta]\!]^{\mathrm{ex}}$, $[\![\Gamma]\!]^{\mathrm{ex}}$. From the rules for typing of LF objects we can derive that $W = \mathbf{w}$ and $\mathbf{code}\ T = [\![\tau]\!]^{\mathrm{ex}}$ and that

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}},\ w : \mathbf{world} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M' : \mathbf{exp}\ w\ T.$$

As we due to a suitable variant of **Lemma 3.4** can find τ' such that $\tau = \Box\,\tau'$ and $T = [\![\tau']\!]^{\mathrm{ex}}$, this judgement can be rewritten into

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}},\ w : \mathbf{world} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M' : \mathbf{exp}\ w\ \left[\!\left[\tau'\right]\!\right]^{\mathrm{ex}}.$$

Now, according to **Lemma 4.1** the context $[\![\Delta]\!]^{\mathrm{ex}}$, $[\![\Gamma]\!]^{\mathrm{ex}}$ is Mini-$\mathrm{ML}^{\Box}_{\mathrm{ex}}$-formed, and as the addition of $w : \mathbf{world}$ does not ruin this, we can apply **Lemma 4.2** and derive that M' cannot contain any free variables referring to world variables different from w. Exploiting the strengthening property described in **Theorem 2.3** we reach that

$$[\![\Delta]\!]^{\mathrm{ex}},\ w : \mathbf{world} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M' : \mathbf{exp}\ w\ \left[\!\left[\tau'\right]\!\right]^{\mathrm{ex}}$$

in which we can substitute $\mathbf{w}$ for w and get

$$[\![\Delta]\!]^{\mathrm{ex}},\ \mathbf{w} : \mathbf{world} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M' : \mathbf{exp}\ \mathbf{w}\ \left[\!\left[\tau'\right]\!\right]^{\mathrm{ex}}.$$

As $[\![\cdot]\!]^{\mathrm{ex}} = \cdot,\ \mathbf{w} : \mathbf{world}$, we can now apply the induction hypothesis to M' and get a derivation $\mathcal{F}' :: \Delta \mid \cdot \vdash^{\mathrm{ex}} e' : \tau'$ such that $[\![\mathcal{F}']\!]^{\mathrm{ex}} = M'$. By application of $\textsc{tp}^{\mathrm{ex}}$-BOX we then get a derivation of $\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{box}\ e' : \Box\,\tau'$. As $\Box\,\tau' = \tau$ and as the derivation is represented by the LF object $\mathbf{box}\ \mathbf{w}\ [\![\tau']\!]^{\mathrm{ex}}\ (\lambda w : \mathbf{world}\,.\ [\![\mathcal{F}']\!]^{\mathrm{ex}})$ which is α-equivalent to M, we are done.

3. $M = \mathbf{let\_box}\ W\ T_1\ T_2\ M_1\ (\lambda u : (\Pi W' : \mathbf{world}\,.\ \mathbf{exp}\ W'\ T_1)\,.\ M_2)$ where u is not present in $[\![\Delta]\!]^{\mathrm{ex}}$, $[\![\Gamma]\!]^{\mathrm{ex}}$. From the rules for typing of LF objects we can derive that $W = \mathbf{w}$ and $T_2 = [\![\tau]\!]^{\mathrm{ex}}$ and that

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M_1 : \mathbf{exp}\ \mathbf{w}\ (\mathbf{code}\ T_1)$$

and

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}},\ u : (\Pi W' : \mathbf{world}\,.\ \mathbf{exp}\ W'\ T_1) \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M_2 : \mathbf{exp}\ \mathbf{w}\ [\![\tau]\!]^{\mathrm{ex}}.$$

As, due to the adequacy of the type representation, we can find τ' such that $T_1 = [\![\tau']\!]^{\mathrm{ex}}$ these last two judgements can be rewritten into

$$[\![\Delta]\!]^{\mathrm{ex}},\ [\![\Gamma]\!]^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M_1 : \mathbf{exp}\ \mathbf{w}\ (\mathbf{code}\ \left[\!\left[\tau'\right]\!\right]^{\mathrm{ex}})$$

and

$$\llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}},\ u : (\Pi\, W' : \mathbf{world}\,.\,\mathbf{exp}\ W'\ \ \llbracket\tau'\rrbracket^{\mathrm{ex}}) \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M_2\ :\ \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau\rrbracket^{\mathrm{ex}}.$$

By application of a suitable variant of **Lemma 3.6** the last judgement can be further rewritten into

$$\cdot,\ u : (\Pi\, W' : \mathbf{world}\,.\,\mathbf{exp}\ W'\ \ \llbracket\tau'\rrbracket^{\mathrm{ex}}),\ \llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{ex}}} M_2\ :\ \mathbf{exp}\ \mathbf{w}\ \ \llbracket\tau\rrbracket^{\mathrm{ex}}.$$

Now, as $\mathbf{code}\ \llbracket\tau'\rrbracket^{\mathrm{ex}} = \llbracket\Box\,\tau'\rrbracket^{\mathrm{ex}}$ and

$$\cdot,\ u : (\Pi\, W' : \mathbf{world}\,.\,\mathbf{exp}\ W'\ \ \llbracket\tau'\rrbracket^{\mathrm{ex}}),\ \llbracket\Delta\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}} = \llbracket\Delta\{u \mapsto \tau'\}\rrbracket^{\mathrm{ex}},\ \llbracket\Gamma\rrbracket^{\mathrm{ex}}$$

when u is not present in $\llbracket\Delta\rrbracket^{\mathrm{ex}}$, we can apply the induction hypothesis to M_1 and M_2 respectively and get derivations $\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\mathrm{ex}} e_1 : \Box\,\tau'$ and $\mathcal{F}_2 :: \Delta\{u \mapsto \tau'\} \mid \Gamma \vdash^{\mathrm{ex}} e_2 : \tau$ such that $\llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}} = M_1$ and $\llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}} = M_2$. Then by application of TP$^{\mathrm{ex}}$-LET-BOX we get a derivation of $\Delta \mid \Gamma \vdash^{\mathrm{ex}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : \tau$, and as this derivation is represented by the LF object $\mathbf{let\_box}\ \mathbf{w}\ \ \llbracket\tau'\rrbracket^{\mathrm{ex}}\ \ \llbracket\tau\rrbracket^{\mathrm{ex}}\ \ \llbracket\mathcal{F}_1\rrbracket^{\mathrm{ex}}\ \ (\lambda u : (\Pi\, W' : \mathbf{world}\,.\,\mathbf{exp}\ W'\ \ \llbracket\tau'\rrbracket^{\mathrm{ex}})\,.\ \llbracket\mathcal{F}_2\rrbracket^{\mathrm{ex}})$ which is α-equivalent to M, we are done. □

4.1.4 Operational semantics for Mini-ML$^{\Box}_{\mathrm{ex}}$

The values among the expressions in Mini-ML$^{\Box}_{\mathrm{ex}}$ are given by

$$\text{Values:}\quad v \ ::= \ \lambda x : \tau\,.\,e \mid \langle v_1,\, v_2\rangle \mid \mathbf{z} \mid \mathbf{s}\ v \mid \mathbf{box}\ e$$

Big-step evaluation

The big-step evaluation relation $\boxed{e \hookrightarrow^{\mathrm{ex}} v}$ evaluating an explicit expression e to an explicit value v is defined by the following inference rules:

$$\frac{}{\lambda x : \tau\,.\,e \ \hookrightarrow^{\mathrm{ex}}\ \lambda x : \tau\,.\,e}\ \text{EVAL}^{\mathrm{ex}}\text{-ABS}$$

$$\frac{e_1 \ \hookrightarrow^{\mathrm{ex}}\ \lambda x : \tau\,.\,e_1' \qquad e_2 \ \hookrightarrow^{\mathrm{ex}}\ v_2 \qquad \{v_2/x\}\,e_1' \ \hookrightarrow^{\mathrm{ex}}\ v}{e_1\ e_2 \ \hookrightarrow^{\mathrm{ex}}\ v}\ \text{EVAL}^{\mathrm{ex}}\text{-APP}$$

$$\frac{\{\mathbf{fix}\ x : \tau\,.\,e/x\}\,e \ \hookrightarrow^{\mathrm{ex}}\ v}{\mathbf{fix}\ x : \tau\,.\,e \ \hookrightarrow^{\mathrm{ex}}\ v}\ \text{EVAL}^{\mathrm{ex}}\text{-FIX}$$

$$\frac{e_1 \ \hookrightarrow^{\mathrm{ex}}\ v_1 \qquad e_2 \ \hookrightarrow^{\mathrm{ex}}\ v_2}{\langle e_1,\, e_2\rangle \ \hookrightarrow^{\mathrm{ex}}\ \langle v_1,\, v_2\rangle}\ \text{EVAL}^{\mathrm{ex}}\text{-PAIR}$$

$$\frac{e \ \hookrightarrow^{\mathrm{ex}}\ \langle v_1,\, v_2\rangle}{\mathbf{fst}\ e \ \hookrightarrow^{\mathrm{ex}}\ v_1}\ \text{EVAL}^{\mathrm{ex}}\text{-FST} \qquad \frac{e \ \hookrightarrow^{\mathrm{ex}}\ \langle v_1,\, v_2\rangle}{\mathbf{snd}\ e \ \hookrightarrow^{\mathrm{ex}}\ v_2}\ \text{EVAL}^{\mathrm{ex}}\text{-SND}$$

$$\frac{}{\mathbf{z} \hookrightarrow^{\text{ex}} \mathbf{z}} \text{ EVAL}^{\text{ex}}\text{-ZERO} \qquad \frac{e \hookrightarrow^{\text{ex}} v}{\mathbf{s}\ e \hookrightarrow^{\text{ex}} \mathbf{s}\ v} \text{ EVAL}^{\text{ex}}\text{-SUCC}$$

$$\frac{e_1 \hookrightarrow^{\text{ex}} \mathbf{z} \qquad e_2 \hookrightarrow^{\text{ex}} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\text{ex}} v} \text{ EVAL}^{\text{ex}}\text{-CASE-Z}$$

$$\frac{e_1 \hookrightarrow^{\text{ex}} \mathbf{s}\ v_1 \qquad \{v_1/x\}\, e_3 \hookrightarrow^{\text{ex}} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\text{ex}} v} \text{ EVAL}^{\text{ex}}\text{-CASE-S}$$

$$\frac{}{\mathbf{box}\ e \hookrightarrow^{\text{ex}} \mathbf{box}\ e} \text{ EVAL}^{\text{ex}}\text{-BOX}$$

$$\frac{e_1 \hookrightarrow^{\text{ex}} \mathbf{box}\ e_1' \qquad \{e_1'/u\}\, e_2 \hookrightarrow^{\text{ex}} v}{\mathbf{let}\ \mathbf{box}\ u = e_1\ \mathbf{in}\ e_2 \hookrightarrow^{\text{ex}} v} \text{ EVAL}^{\text{ex}}\text{-LET-BOX}$$

The operational semantics made up by these rules cannot be categorized as being completely call-by-value. This due to the last EVAL$^{\text{ex}}$-LET-BOX rule where the code e_1 is not evaluated before being substituted into the body expression e_2. Only if the code ends up at an unboxed place within e_2 it will be evaluated during the evaluation of e_2.

Having the big-step operational semantics in place we are now ready to consider an example of a Mini-ML$^{\Box}_{\text{ex}}$ program:

$$\begin{aligned} \mathit{power}^{\text{ex}} \equiv\ & \mathbf{fix}\ p : \mathbf{nat} \to \Box\,(\mathbf{nat} \to \mathbf{nat})\,. \\ & \lambda n : \mathbf{nat}\,. \\ & \quad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda x : \mathbf{nat}\,.\ \mathbf{s}\ \mathbf{z}) \\ & \qquad \mid\ \mathbf{s}\ m \Rightarrow \mathbf{let}\ \mathbf{box}\ q = p\ m \\ & \qquad\qquad\quad \mathbf{in}\ \ \mathbf{box}\ (\lambda x : \mathbf{nat}\,.\ \mathit{times}\ x\ (q\ x)) \end{aligned}$$

This program can very well be characterized as a specialized-power-program-generator. Applied to some natural number n, $\mathit{power}^{\text{ex}}$ generates code for a program of type $\mathbf{nat} \to \mathbf{nat}$ which when applied to some other natural number x will return the value of x^n. For $n = 2$ we can derive the following evaluation using the operational semantics given above:

$$\begin{aligned} & \mathit{power}^{\text{ex}}\ (\mathbf{s}\ (\mathbf{s}\ \mathbf{z})) \hookrightarrow^{\text{ex}} \\ & \quad \mathbf{box}\ (\lambda x : \mathbf{nat}\,.\ \mathit{times}\ x\ ((\lambda x : \mathbf{nat}\,.\ \mathit{times}\ x\ ((\lambda x : \mathbf{nat}\,.\ \mathbf{s}\ \mathbf{z})\ x))\ x)) \end{aligned}$$

Apart from a couple of β-redexes the code within the box-constructor on the right indeed looks very similar to the specialized power program power_2 introduced in **Chapter 1**. The metaprogramming systems considered in later chapters will allow us to avoid superfluous β-redexes like the ones present here.

4.1.5 Representing the Mini-ML$^{\Box}_{\text{ex}}$ operational semantics in LF

As there is no sub-typing within LF, the sub-grammar of values defined in the previous section is not very suitable for representation within LF. Instead we translate the sub-grammar into a judgement on expressions which can then be represented using the standard judgements-as-types methodology. Using v to range over expressions expected to be values, we define a value judgement $\boxed{\text{value}^{\text{ex}}(v)}$ by the following inference rules:

$$\frac{}{\text{value}^{\text{ex}}(\lambda x : \tau\ .\ e)}\ \text{VAL}^{\text{ex}}\text{-ABS}$$

$$\frac{\text{value}^{\text{ex}}(v_1) \qquad \text{value}^{\text{ex}}(v_2)}{\text{value}^{\text{ex}}(\langle v_1,\ v_2\rangle)}\ \text{VAL}^{\text{ex}}\text{-PAIR}$$

$$\frac{}{\text{value}^{\text{ex}}(\mathbf{z})}\ \text{VAL}^{\text{ex}}\text{-ZERO} \qquad \frac{\text{value}^{\text{ex}}(v)}{\text{value}^{\text{ex}}(\mathbf{s}\ v)}\ \text{VAL}^{\text{ex}}\text{-SUCC}$$

$$\frac{}{\text{value}^{\text{ex}}(\mathbf{box}\ e)}\ \text{VAL}^{\text{ex}}\text{-BOX}$$

We will not list all the object constants added to the LF signature due to this new value judgement but instead refer to the Twelf code in Section **A.3**. The Twelf type family used for representing instances of the value judgement is the one named `val_e`.

With the new value judgement the big-step judgement becomes a judgement evaluating one expression into some other expression and this later expression can then be proved (by straightforward induction on the structure of the evaluation derivation) to be a value. For a representation of such value soundness proof within LF the higher-level judgement

$$\begin{array}{c}\mathcal{G}\\ e\ \hookrightarrow^{\text{ex}}\ v\end{array}\ \Longrightarrow\ \begin{array}{c}\mathcal{H}\\ \text{value}^{\text{ex}}(v)\end{array}$$

and appropriate inference rules defining it should be considered. The representation is straightforward and in our Twelf code the type family `val_sound_e` is the one capturing the form of this higher-level judgement. Big-step evaluations are represented by use of the type family `eval_e`.

4.1.6 Metatheory about Mini-ML$^{\Box}_{\text{ex}}$

Davies and Pfenning state the following properties about typing and evaluation within Mini-ML$^{\Box}_{\text{ex}}$:

Lemma 4.3 (Substitution)**.**

a) If

$$\Delta \mid \Gamma \vdash^{\text{ex}} e_1 \ : \ \tau_1$$

and

$$\Delta \mid (\Gamma,\ x : \tau_1,\ \Gamma') \vdash^{\text{ex}} e_2 \ : \ \tau_2$$

then also

$$\Delta \mid (\Gamma,\ \Gamma') \vdash^{\text{ex}} \{e_1/x\}\, e_2 \ : \ \tau_2.$$

b) If

$$\Delta \mid \cdot \vdash^{\text{ex}} e_1 \ : \ \tau_1$$

and

$$(\Delta,\ u :: \tau_1,\ \Delta') \mid \Gamma \vdash^{\text{ex}} e_2 \ : \ \tau_2$$

then also

$$(\Delta,\ \Delta') \mid \Gamma \vdash^{\text{ex}} \{e_1/u\}\, e_2 \ : \ \tau_2.$$

Proof. The lemma can be proved by straightforward induction on the structure of the typing derivation for e_2. □

Theorem 4.3 (Value soundness)**.** If $e \hookrightarrow^{\text{ex}} v$ then v is a value.

In fact the contents of this theorem do not make much sense if, as in the papers of Davies and Pfenning, the explicit values are defined as a sub-grammar and the big-step evaluation relation is defined as a subset of the product set of expressions and values. In that case the theorem can be said to be trivially valid. The issue of value soundness was already addressed in the previous section where we considered the LF representation of the operational semantics.

Theorem 4.4 (Evaluation determinancy)**.** If $e \hookrightarrow^{\text{ex}} v$ and $e \hookrightarrow^{\text{ex}} v'$ then $v = v'$.

Proof. The theorem can be proved by straightforward induction on the structure of the derivation of $e \hookrightarrow^{\text{ex}} v$. □

Theorem 4.5 (Type preservation)**.** If $e \hookrightarrow^{\text{ex}} v$ and

$$\cdot \mid \cdot \vdash^{\text{ex}} e \ : \ \tau$$

then also

$$\cdot \mid \cdot \vdash^{\text{ex}} v \ : \ \tau.$$

Proof. The theorem can be proved by induction on the structure of the derivation of $e \hookrightarrow^{\text{ex}} v$. The only interesting case is the one where EVAL$^{\text{ex}}$-LET-BOX is assumed to be the last rule applied in the derivation and a proof of this case can be found in [2]. □

Type soundness

Apart from the above properties, stated and proved by Pfenning and Davies, it would also be most relevant to investigate on whether the given type system is strong enough to ensure that no well-typed explicit expression gets stuck during evaluation. However, as Mini-ML$^{\Box}_{\text{ex}}$ can be obtained as a fragment of the language based on contextual modal logic presented in **Chapter 6**, it makes good sense to postpone the proof of type soundness until we get to the contextual language.

4.1.7 Representing the Mini-ML$^{\Box}_{\text{ex}}$ metatheory in LF

Using intrinsically typed LF syntax for the representation of explicit expressions, both the substitution and the type preservation property follow for free.

Evaluation determinacy

As mentioned above, **Theorem 4.4** can be proved by straightforward induction on the structure of the derivation of $e \hookrightarrow^{\text{ex}} v$. When formalized within LF the proof cases are slightly complicated though. We need to explicitly represent the fact that when a certain expression constructor is applied to equal values then the resulting expressions will also be equal. That is, if for instance $v = v'$ then also $\mathbf{s}\ v = \mathbf{s}\ v'$. The constants in our Twelf code related to these lemmas are all prefixed with `eq_e_`.

Further, we need an explicit representation of the fact that $\mathbf{z}$ can never be equal to some successor expression $\mathbf{s}\ e$. Put in a more higher-level judgement friendly wording, we need to prove that if we can prove $\mathbf{z} = \mathbf{s}\ e$ then we can prove equality of any two explicit expressions:

$$\frac{\mathcal{E}}{\mathbf{z} = \mathbf{s}\ e} \Longrightarrow \frac{\mathcal{E}'}{e' = e''}$$

As we can never prove $\mathbf{z} = \mathbf{s}\ e$ there will be no inference rules defining this judgement. The lemma is needed to finish the empty proof cases where the last rule applied in the two evaluations found in the theorem is either EVAL$^{\text{ex}}$-CASE-Z and EVAL$^{\text{ex}}$-CASE-S respectively or EVAL$^{\text{ex}}$-CASE-S and EVAL$^{\text{ex}}$-CASE-Z respectively. In our Twelf code the type family `neq_e_zs` is the one used for the representation of the described lemma.

4.2 The implicit □-language

The implicit language is an extension of the functional language Mini-ML and is referred to as Mini-ML$^{\Box}$.

4.2.1 Syntax of Mini-ML$^{\Box}$

The syntax categories of Mini-ML$^{\Box}$ are given by

$$
\begin{array}{rrcl}
\text{Types:} & \tau & ::= & \mathbf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \Box\,\tau \\
\text{Expressions:} & m & ::= & x \mid \lambda x : \tau\,.\,m \mid m_1\ m_2 \mid \mathbf{fix}\ x : \tau\,.\,m \mid \langle m_1, m_2 \rangle \mid \\
 & & & \mathbf{fst}\ m \mid \mathbf{snd}\ m \mid \mathbf{z} \mid \mathbf{s}\ m \mid \mathbf{case}\ m\ \mathbf{of}\ \mathbf{z} \Rightarrow m_1 \mid \mathbf{s}\ x \Rightarrow m_2 \mid \\
 & & & \mathbf{box}\ m \mid \mathbf{unbox}\ p \\
\text{Pops:} & p & ::= & m \mid \mathbf{pop}\ p \\
 & & & \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma,\ x : \tau \\
\text{Context stacks:} & \Psi & ::= & \cdot \mid \Psi;\ \Gamma
\end{array}
$$

Just like we saw it in the explicit language, the necessity type $\Box\,\tau$ is meant for classification of code expressions where these are constructed using the box-constructor. As also indicated by the name, the unbox-constructor is used for code unboxing, and the pop-constructor transfers code from some previous computation stage. Corresponding to how the implicit programming style was described in the introduction, Ψ can be seen as a memory of previous local Γ-contexts to be used when a need for earlier generated code arises.

We will use $\boxed{\Gamma_1,\ \Gamma_2}$ to mean the concatenation of the contexts Γ_1 and Γ_2, and the function $\boxed{\Gamma\{x \mapsto \tau\}}$ updating a context with a new assumption is defined similarly to what we previously saw.

Whereas a context Γ will be considered valid iff it does not contain multiple entries for some same variable name, a context stack Ψ will be considered valid iff the contexts making it up are all valid. That is, we do not require unique variable names across an entire context stack. In the following we will presume validity of all mentioned contexts and context stacks unless otherwise is explicitly stated.

4.2.2 Typing rules for Mini-ML$^{\Box}$

The typing judgement for Mini-ML$^{\Box}$ has the form $\boxed{(\Psi;\ \Gamma)\ \vdash^{\text{im}}\ m\ :\ \tau}$ and with the assumption overwriting methodology incorporated it is defined by the following inference rules:

$$\frac{\Gamma(x) = \tau}{(\Psi;\ \Gamma)\ \vdash^{\text{im}}\ x\ :\ \tau}\ \text{TP}^{\text{im}}\text{-VAR}$$

$$\frac{(\Psi;\ \Gamma\{x \mapsto \tau_1\})\ \vdash^{\text{im}}\ m\ :\ \tau_2}{(\Psi;\ \Gamma)\ \vdash^{\text{im}}\ \lambda x : \tau_1\,.\,m\ :\ \tau_1 \to \tau_2}\ \text{TP}^{\text{im}}\text{-ABS}$$

$$\frac{(\Psi;\ \Gamma)\ \vdash^{\text{im}}\ m_1\ :\ \tau_1 \to \tau_2 \qquad (\Psi;\ \Gamma)\ \vdash^{\text{im}}\ m_2\ :\ \tau_1}{(\Psi;\ \Gamma)\ \vdash^{\text{im}}\ m_1\ m_2\ :\ \tau_2}\ \text{TP}^{\text{im}}\text{-APP}$$

$$\frac{(\Psi;\ \Gamma\{x \mapsto \tau\}) \vdash^{\mathrm{im}} m\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fix}\ x : \tau\,.\ m\ :\ \tau}\ \text{TP}^{\mathrm{im}}\text{-FIX}$$

$$\frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ :\ \tau_1 \qquad (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_2\ :\ \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \langle m_1, m_2\rangle\ :\ \tau_1 \times \tau_2}\ \text{TP}^{\mathrm{im}}\text{-PAIR}$$

$$\frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fst}\ m\ :\ \tau_1}\ \text{TP}^{\mathrm{im}}\text{-FST} \qquad \frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{snd}\ m\ :\ \tau_2}\ \text{TP}^{\mathrm{im}}\text{-SND}$$

$$\frac{}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{z}\ :\ \mathbf{nat}}\ \text{TP}^{\mathrm{im}}\text{-ZERO} \qquad \frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \mathbf{nat}}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{s}\ m\ :\ \mathbf{nat}}\ \text{TP}^{\mathrm{im}}\text{-SUCC}$$

$$\frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \mathbf{nat} \quad (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ :\ \tau \quad (\Psi;\ \Gamma\{x \mapsto \mathbf{nat}\}) \vdash^{\mathrm{im}} m_2\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{case}\ m\ \mathbf{of}\ \mathbf{z} \Rightarrow m_1 \mid \mathbf{s}\ x \Rightarrow m_2\ :\ \tau}\ \text{TP}^{\mathrm{im}}\text{-CASE}$$

$$\frac{((\Psi;\ \Gamma);\ \cdot) \vdash^{\mathrm{im}} m\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{box}\ m\ :\ \Box\tau}\ \text{TP}^{\mathrm{im}}\text{-BOX} \qquad \frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{unbox}\ p\ :\ \tau}\ \text{TP}^{\mathrm{im}}\text{-UNBOX}$$

where $\boxed{\Gamma(x) = \tau}$ means that $x : \tau$ is present in the context Γ and where the judgement typing pop expressions $\boxed{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}$ is defined by

$$\frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \Box\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} m\ :\ \Box\tau}\ \text{TP}^{\mathrm{im,p}}\text{-0} \qquad \frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}{((\Psi;\ \Gamma);\ \Gamma') \vdash^{\mathrm{im,p}} \mathbf{pop}\ p\ :\ \Box\tau}\ \text{TP}^{\mathrm{im,p}}\text{-M}$$

The fact that entering a box construct corresponds to entering a new computation stage is reflected by a fresh local context being pushed onto the stack of contexts at the top of the TP$^{\mathrm{im}}$-BOX rule. The browsing back through old local environments looking for code like we described it in the introduction is reflected in the TP$^{\mathrm{im,p}}$-M rule.

For convenience $\mathbf{unbox}_1\ m$ is introduced as syntactic sugar for $\mathbf{unbox}\ (\mathbf{pop}\ m)$, and we then have the following derived typing inference rule:

$$\frac{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}{((\Psi;\ \Gamma);\ \Gamma') \vdash^{\mathrm{im}} \mathbf{unbox}_1\ p\ :\ \tau}\ \text{TP}^{\mathrm{im}}\text{-UNBOX}_1$$

As we also did in the explicit case, we now illustrate the nature of Mini-ML$^{\Box}$ by reformulating the proof of $\Box\, A$ given in the introduction as a typing derivation within Mini-ML$^{\Box}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_0(g) = \Box\,(C \to A)}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im}} g\ :\ \Box\,(C \to A)}}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im,\,p}} g\ :\ \Box\,(C \to A)}}{((\cdot;\ \Gamma_0);\ \cdot) \vdash^{\mathrm{im}} \mathbf{unbox}_1\ g\ :\ C \to A} \qquad \dfrac{\dfrac{\dfrac{\Gamma_0(f) = B \to \Box\,C}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im}} f\ :\ B \to \Box\,C} \qquad \dfrac{\Gamma_0(x) = B}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im}} x\ :\ B}}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im}} f\ x\ :\ \Box\,C}}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im,\,p}} f\ x\ :\ \Box\,C}}{((\cdot;\ \Gamma_0);\ \cdot) \vdash^{\mathrm{im}} \mathbf{unbox}_1\ (f\ x)\ :\ C}$$

$$\dfrac{((\cdot;\ \Gamma_0);\ \cdot) \vdash^{\mathrm{im}} (\mathbf{unbox}_1\ g)\ (\mathbf{unbox}_1\ (f\ x))\ :\ A}{(\cdot;\ \Gamma_0) \vdash^{\mathrm{im}} \mathbf{box}\ ((\mathbf{unbox}\ (\mathbf{pop}\ g))\ (\mathbf{unbox}_1\ (f\ x)))\ :\ \Box\,A}$$

where $\Gamma_0 = \cdot,\ x : B,\ f : B \to \Box\,C,\ g : \Box\,(C \to A)$.

4.2.3 Representing well-typed Mini-ML$^{\Box}$ expressions in LF

Now we need to find a way to adequately represent implicit expressions and their typings within LF. Just as in the explicit case each well-typed implicit expression has exactly one derivable type and thus we will once again choose a type-indexed LF syntax for our expression representation.

Naive LF representation

We begin by outlining why a straightforward representation approach does not work in the implicit case either. In fact, as opposed to the explicit case, none of the adequacy directions holds for the naive representation in the implicit case.

The naive version of the LF signature Σ_{im} extends the signature Σ_{si} with the following constants:

$$\begin{aligned}
\mathbf{code} &: \mathbf{tp} \to \mathbf{tp} \\
\\
\mathbf{box} &: \Pi\,T : \mathbf{tp}\,.\,\mathbf{exp}\ T \to \mathbf{exp}\ (\mathbf{code}\ T) \\
\mathbf{unbox} &: \Pi\,T : \mathbf{tp}\,.\,\mathbf{pop}\ (\mathbf{code}\ T) \to \mathbf{exp}\ T \\
\\
\mathbf{pop} &: \mathbf{tp} \to \mathbf{type} \\
\mathbf{pop_0} &: \Pi\,T : \mathbf{tp}\,.\,\mathbf{exp}\ (\mathbf{code}\ T) \to \mathbf{pop}\ (\mathbf{code}\ T) \\
\mathbf{pop_m} &: \Pi\,T : \mathbf{tp}\,.\,\mathbf{pop}\ (\mathbf{code}\ T) \to \mathbf{pop}\ (\mathbf{code}\ T)
\end{aligned}$$

Regarding type representation we define the function $\boxed{\llbracket \tau \rrbracket^{\mathrm{im}}}$ mapping an implicit type τ into an LF object T similar to the type representation functions previously seen:

$$\begin{aligned}
\llbracket \mathbf{nat} \rrbracket^{\mathrm{im}} &= \mathbf{nat} \\
\llbracket \tau_1 \to \tau_2 \rrbracket^{\mathrm{im}} &= \mathbf{arrow}\ \llbracket \tau_1 \rrbracket^{\mathrm{im}}\ \llbracket \tau_2 \rrbracket^{\mathrm{im}} \\
\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathrm{im}} &= \mathbf{product}\ \llbracket \tau_1 \rrbracket^{\mathrm{im}}\ \llbracket \tau_2 \rrbracket^{\mathrm{im}} \\
\llbracket \Box\,\tau \rrbracket^{\mathrm{im}} &= \mathbf{code}\ \llbracket \tau \rrbracket^{\mathrm{im}}
\end{aligned}$$

This straightforward type representation can be proved adequate just like in the previous chapters. A lemma similar to **Lemma 3.4** can also be proved.

At the current point we assume that the functions $\llbracket\Psi\rrbracket^{\mathrm{im}}$, $\llbracket\Gamma\rrbracket^{\mathrm{im}}$, and $\llbracket\mathcal{M}\rrbracket^{\mathrm{im}}$ and $\llbracket\mathcal{N}\rrbracket^{\mathrm{im}}$ mapping context stacks, contexts, and typing derivations respectively are defined in straightforward ways similar to what was considered initially in the explicit case.

For the given LF signature and representation functions to be an adequate collection it should be the case that any well-typed implicit expression is mapped into a well-typed LF object. Written more formally, we should be able to prove that if $\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau$ then

$$\llbracket\Psi;\ \Gamma\rrbracket^{\mathrm{im}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}\ :\ \mathbf{exp}\ \llbracket\tau\rrbracket^{\mathrm{im}}.$$

This is not the case though. If we for instance take the typing derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\cdot,\ x:\tau_1\to\tau_2)(x) = \tau_1\to\tau_2}}{(\cdot;\ (\cdot,\ x:\Box\,\tau_1));\ (\cdot,\ x:\tau_1\to\tau_2)\ \vdash^{\mathrm{im}}\ x\ :\ \tau_1\to\tau_2} \quad \dfrac{\dfrac{\dfrac{}{(\cdot,\ x:\Box\,\tau_1)(x) = \Box\,\tau_1}}{\cdot;\ (\cdot,\ x:\Box\,\tau_1)\ \vdash^{\mathrm{im}}\ x\ :\ \Box\,\tau_1}}{\cdot;\ (\cdot,\ x:\Box\,\tau_1)\ \vdash^{\mathrm{im,\,p}}\ x\ :\ \Box\,\tau_1}}{(\cdot;\ (\cdot,\ x:\Box\,\tau_1));\ (\cdot,\ x:\tau_1\to\tau_2)\ \vdash^{\mathrm{im}}\ \mathbf{unbox}_1\ x\ :\ \tau_1}}{(\cdot;\ (\cdot,\ x:\Box\,\tau_1));\ (\cdot,\ x:\tau_1\to\tau_2)\ \vdash^{\mathrm{im}}\ x\ (\mathbf{unbox}_1\ x)\ :\ \tau_2}}{(\cdot;\ (\cdot,\ x:\Box\,\tau_1));\ \cdot\ \vdash^{\mathrm{im}}\ \lambda x:\tau_1\to\tau_2\,.\,(x\ (\mathbf{unbox}_1\ x))\ :\ (\tau_1\to\tau_2)\to\tau_2}}{\cdot;\ (\cdot,\ x:\Box\,\tau_1)\ \vdash^{\mathrm{im}}\ \mathbf{box}\ (\lambda x:\tau_1\to\tau_2\,.\,(x\ (\mathbf{unbox}_1\ x)))\ :\ \Box\,((\tau_1\to\tau_2)\to\tau_2)}}{\cdot;\ \cdot\ \vdash^{\mathrm{im}}\ \lambda x:\Box\,\tau_1\,.\,(\mathbf{box}\ (\lambda x:\tau_1\to\tau_2\,.\,(x\ (\mathbf{unbox}_1\ x))))\ :\ \Box\,\tau_1\to\Box\,((\tau_1\to\tau_2)\to\tau_2)}$$

then for the adequacy property to be fulfilled we should be able to derive that

$$x:\mathbf{exp}\ (\mathbf{code}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}),\ x:\mathbf{exp}\ (\mathbf{arrow}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}}) \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} x\ :\ \mathbf{exp}\ (\mathbf{code}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}).$$

This is not possible as the assumption $x:\mathbf{exp}\ (\mathbf{arrow}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}})$ shades the assumption we would have to make use of in such a derivation. Well, the context is actually not a valid LF context in the first place.

LF representation using level counter

From the example just given we learned that we need to figure out how to collapse a multi-level paper context that may contain overlapping variable names into a one-level LF context in such a way that no assumption from one level shades an assumption from a different level. Introducing intrinsic worlds the way we saw it done in the explicit case does not help this problem as world indexing will not contribute to distinction of colliding variable names. An actual solution is to appropriately equip the representation functions with a counter k which can be used to keep track of the current context level. We consider the following improved representation functions:

The function $\llbracket\Gamma\rrbracket^{\mathrm{im}}_k$mapping a context Γ into an LF context Λ translates each variable into a variable labeled with the level to which it originally belonged:

$$\begin{aligned}\llbracket\cdot\rrbracket^{\mathrm{im}}_k &= \cdot\\ \llbracket\Gamma,\ x:\tau\rrbracket^{\mathrm{im}}_k &= \llbracket\Gamma\rrbracket^{\mathrm{im}}_k,\ x_k:\mathbf{exp}\ \llbracket\tau\rrbracket^{\mathrm{im}}\end{aligned}$$

The context stack function $[\![\Psi]\!]^{\mathrm{im}}$ maps a context stack into a pair consisting of an integer k and an LF context Λ:

$$\begin{aligned} [\![\cdot]\!]^{\mathrm{im}} &= \{0;\ \cdot\} \\ [\![\Psi;\ \Gamma]\!]^{\mathrm{im}} &= \{k+1;\ \Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k\} \text{ when } [\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\} \end{aligned}$$

The function $[\![\mathcal{M}]\!]^{\mathrm{im}}_k$ mapping a typing derivation $\mathcal{M}$ for an implicit expression m into an LF object M such that $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M : \mathbf{exp}\ [\![\tau]\!]^{\mathrm{im}}$ when $\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m : \tau$ and $[\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\}$ is defined by

$$\left[\!\!\left[\frac{\Gamma(x) = \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} x : \tau} \right]\!\!\right]^{\mathrm{im}}_k = x_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma\{x \mapsto \tau_1\}) \vdash^{\mathrm{im}} m : \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \lambda x{:}\tau_1\,.\,m : \tau_1 \to \tau_2} \right]\!\!\right]^{\mathrm{im}}_k =$$

$$\mathbf{lam}\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ (\lambda x_k : \mathbf{exp}\ [\![\tau_1]\!]^{\mathrm{im}}\ .\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k)$$

$$\left[\!\!\left[\frac{\mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1 : \tau_1 \to \tau_2 \qquad \mathcal{M}_2 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_2 : \tau_1}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ m_2 : \tau_2} \right]\!\!\right]^{\mathrm{im}}_k =$$

$$\mathbf{app}\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ [\![\mathcal{M}_1]\!]^{\mathrm{im}}_k\ [\![\mathcal{M}_2]\!]^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma\{x \mapsto \tau\}) \vdash^{\mathrm{im}} m : \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fix}\ x{:}\tau\,.\,m : \tau} \right]\!\!\right]^{\mathrm{im}}_k = \mathbf{fix}\ [\![\tau]\!]^{\mathrm{im}}\ (\lambda x_k : \mathbf{exp}\ [\![\tau]\!]^{\mathrm{im}}\ .\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k)$$

$$\left[\!\!\left[\frac{\mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1 : \tau_1 \qquad \mathcal{M}_2 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_2 : \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \langle m_1, m_2\rangle : \tau_1 \times \tau_2} \right]\!\!\right]^{\mathrm{im}}_k =$$

$$\mathbf{pair}\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ [\![\mathcal{M}_1]\!]^{\mathrm{im}}_k\ [\![\mathcal{M}_2]\!]^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m : \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fst}\ m : \tau_1} \right]\!\!\right]^{\mathrm{im}}_k = \mathbf{fst}\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m : \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{snd}\ m : \tau_2} \right]\!\!\right]^{\mathrm{im}}_k = \mathbf{snd}\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{z} : \mathbf{nat}} \right]\!\!\right]^{\mathrm{im}}_k = \mathbf{z}$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m : \mathbf{nat}}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{s}\ m : \mathbf{nat}} \right]\!\!\right]^{\mathrm{im}}_k = \mathbf{s}\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m : \mathbf{nat} \quad \mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1 : \tau \quad \mathcal{M}_2 :: (\Psi;\ \Gamma\{x \mapsto \mathbf{nat}\}) \vdash^{\mathrm{im}} m_2 : \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{case}\ m\ \mathbf{of}\ \mathbf{z} \Rightarrow m_1 \mid \mathbf{s}\ x \Rightarrow m_2 : \tau} \right]\!\!\right]^{\mathrm{im}}_k$$

$$= \mathbf{case}\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k}\ \llbracket\mathcal{M}_1\rrbracket^{\mathrm{im}}_{k}\ (\lambda x_k : \mathbf{exp}\ \mathbf{nat}\,.\ \llbracket\mathcal{M}_2\rrbracket^{\mathrm{im}}_{k})$$

$$\left\llbracket \frac{\mathcal{M} :: ((\Psi;\ \Gamma);\ \cdot)\ \vdash^{\mathrm{im}}\ m\ :\ \tau}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ \mathbf{box}\ m\ :\ \Box\tau} \right\rrbracket^{\mathrm{im}}_{k} = \mathbf{box}\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k+1}$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\tau}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ \mathbf{unbox}\ p\ :\ \tau} \right\rrbracket^{\mathrm{im}}_{k} = \mathbf{unbox}\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k}$$

Note how the level jump appearing in the typing rule $\textsc{tp}^{\mathrm{im}}$-$\textsc{box}$ is explicitly reflected in this definition.

Finally the function $\llbracket\mathcal{N}\rrbracket^{\mathrm{im}}_{k}$ mapping a typing derivation $\mathcal{N}$ for a pop expression p into an LF object M such that $\Lambda,\ \llbracket\Gamma\rrbracket^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ M\ :\ \mathbf{exp}\ (\mathbf{code}\ \llbracket\tau\rrbracket^{\mathrm{im}})$ when $\mathcal{N} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\tau$ and when $\llbracket\Psi\rrbracket^{\mathrm{im}} = \{k;\ \Lambda\}$ is defined by:

$$\left\llbracket \frac{\mathcal{N} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \Box\tau}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ m\ :\ \Box\tau} \right\rrbracket^{\mathrm{im}}_{k} = \mathbf{pop\_0}\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k}$$

$$\left\llbracket \frac{\mathcal{N} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\tau}{((\Psi;\ \Gamma);\ \Gamma')\ \vdash^{\mathrm{im,p}}\ \mathbf{pop}\ p\ :\ \Box\tau} \right\rrbracket^{\mathrm{im}}_{k} = \mathbf{pop\_m}\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k-1} \text{ for } k > 0$$

Note how the level decrease introduced by the typing rule $\textsc{tp}^{\mathrm{im,p}}$-$\textsc{m}$ is explicitly reflected in this definition.

With these new representation functions the above mentioned troublesome LF typing takes the form

$$x_0 : \mathbf{exp}\ (\mathbf{code}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}),\ x_1 : \mathbf{exp}\ (\mathbf{arrow}\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}})\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ x_0\ :\ \mathbf{exp}\ (\mathbf{code}\ \llbracket\tau_1\rrbracket^{\mathrm{im}})$$

which is indeed derivable.

The introduction of the level counter thus seems to have enabled the representation of all well-typed implicit expressions within LF. Also the counter prevents an untypable implicit expressions like for example

$$\lambda x : \tau\,.\ \mathbf{box}\ x.$$

from being translated into some well-typed LF object. However, we still have the problem which was already identified in the case of explicit expressions, namely the presence of valid LF objects that do not represent any well-typed implicit expressions. An example of this is the LF object given in

$$\cdot \vdash^{\mathrm{LF}}\ \mathbf{box}\ T\ (\mathbf{lam}\ T\ T\ (\lambda x_0 : T\,.\ (\mathbf{unbox}\ T\ (\mathbf{pop\_m}\ T$$

$$(\mathbf{pop\_0}\ T\ (\mathbf{lam}\ T\ T\ (\lambda x_1 : T\,.\,(\mathbf{box}\ T\ x_1))))))))\ :\ \mathbf{exp}\ (\mathbf{code}\ T).$$

As we also realized in the explicit case, this problem arises due to neglection of the need to have "invisible" assumptions in the LF context. Below we will again see how unpleasant LF objects can be prevented by the use of worlds.

LF representation using both level counter and worlds

Similar to what we did in the explicit case we will try to solve the problem described above by making it explicit on translation in which of the worlds of the underlying Kripke model the variables occurring freely within a given expression should be found.

We replace the current contents of the LF signature Σ_{im} with the following constant declarations:

$$\begin{aligned}
\mathbf{world} &: \mathbf{type}\\
\mathbf{jump} &: \mathbf{world} \to \mathbf{world} \to \mathbf{type}\\
\\
\mathbf{tp} &: \mathbf{type}\\
\mathbf{nat} &: \mathbf{tp}\\
\mathbf{arrow} &: \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp}\\
\mathbf{product} &: \mathbf{tp} \to \mathbf{tp} \to \mathbf{tp}\\
\mathbf{code} &: \mathbf{tp} \to \mathbf{tp}\\
\\
\mathbf{exp} &: \mathbf{world} \to \mathbf{tp} \to \mathbf{type}\\
\mathbf{lam} &: \Pi W : \mathbf{world}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,.\\
&\quad (\mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2) \to \mathbf{exp}\ W\ (\mathbf{arrow}\ T_1\ T_2)\\
\mathbf{app} &: \Pi W : \mathbf{world}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,.\\
&\quad \mathbf{exp}\ W\ (\mathbf{arrow}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2\\
\mathbf{fix} &: \Pi W : \mathbf{world}\,.\,\Pi T : \mathbf{tp}\,.\,(\mathbf{exp}\ W\ T \to \mathbf{exp}\ W\ T) \to \mathbf{exp}\ W\ T\\
\mathbf{pair} &: \Pi W : \mathbf{world}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,.\\
&\quad \mathbf{exp}\ W\ T_1 \to \mathbf{exp}\ W\ T_2 \to \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2)\\
\mathbf{fst} &: \Pi W : \mathbf{world}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,.\,\mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_1\\
\mathbf{snd} &: \Pi W : \mathbf{world}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,.\,\mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \to \mathbf{exp}\ W\ T_2\\
\mathbf{z} &: \Pi W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \mathbf{nat}\\
\mathbf{s} &: \Pi W : \mathbf{world}\,.\,\mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ \mathbf{nat}\\
\mathbf{case} &: \Pi W : \mathbf{world}\,.\,\Pi T : \mathbf{tp}\,.\\
&\quad \mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ T \to (\mathbf{exp}\ W\ \mathbf{nat} \to \mathbf{exp}\ W\ T) \to
\end{aligned}$$

$$\begin{array}{rcl} & & \mathbf{exp}\ W\ T \\ \mathbf{box} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,. \\ & & \quad (\Pi\, W' : \mathbf{world}\,.\, \mathbf{jump}\ W\ W' \ \to\ \mathbf{exp}\ W'\ T) \ \to\ \mathbf{exp}\ W\ (\mathbf{code}\ T) \\ \mathbf{unbox} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,.\, \mathbf{pop}\ W\ (\mathbf{code}\ T) \ \to\ \mathbf{exp}\ W\ T \end{array}$$

$$\begin{array}{rcl} \mathbf{pop} & : & \mathbf{world} \ \to\ \mathbf{tp} \ \to\ \mathbf{type} \\ \mathbf{pop\_0} & : & \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,.\, \mathbf{exp}\ W\ (\mathbf{code}\ T) \ \to\ \mathbf{pop}\ W\ (\mathbf{code}\ T) \\ \mathbf{pop\_m} & : & \Pi\, W' : \mathbf{world}\,.\, \Pi\, W : \mathbf{world}\,.\, \Pi\, T : \mathbf{tp}\,. \\ & & \quad \mathbf{jump}\ W'\ W \ \to\ \mathbf{pop}\ W'\ (\mathbf{code}\ T) \ \to\ \mathbf{pop}\ W\ (\mathbf{code}\ T) \end{array}$$

The corresponding LF representation functions working on contexts, context stacks and typed expressions respectively are defined in the following way:

The function $\boxed{[\![\Gamma]\!]^{\mathrm{im}}_k}$ mapping a context Γ into an LF context Λ:

$$\begin{array}{rcl} [\![\cdot]\!]^{\mathrm{im}}_k & = & \cdot \\ [\![\Gamma,\ x : \tau]\!]^{\mathrm{im}}_k & = & [\![\Gamma]\!]^{\mathrm{im}}_k\,,\ x_k : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}} \end{array}$$

The function $\boxed{[\![\Psi]\!]^{\mathrm{im}}}$ mapping a context stack into an integer k and an LF context Λ:

$$\begin{array}{rcl} [\![\cdot]\!]^{\mathrm{im}} & = & \{0;\ \mathbf{w}_0 : \mathbf{world}\} \\ [\![\Psi;\ \Gamma]\!]^{\mathrm{im}} & = & \{k+1;\ \Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k\,,\ \mathbf{w}_{k+1} : \mathbf{world},\ \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}\} \\ & & \text{when } [\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\} \end{array}$$

The function $\boxed{[\![\mathcal{M}]\!]^{\mathrm{im}}_k}$ mapping a typing derivation for an implicit expression m into an LF object M such that $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M\ :\ \mathbf{exp}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}$ when $\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau$ and $[\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\}$:

$$\left[\!\!\left[\frac{\Gamma(x) = \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} x\ :\ \tau} \right]\!\!\right]^{\mathrm{im}}_k = x_k$$

$$\left[\!\!\left[\frac{\mathcal{M} :: (\Psi;\ \Gamma\{x \mapsto \tau_1\}) \vdash^{\mathrm{im}} m\ :\ \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \lambda x : \tau_1\,.\, m\ :\ \tau_1 \to \tau_2} \right]\!\!\right]^{\mathrm{im}}_k =$$

$$\mathbf{lam}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ (\lambda x_k : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}}\ .\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k)$$

$$\left[\!\!\left[\frac{\mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ :\ \tau_1 \to \tau_2 \qquad \mathcal{M}_2 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_2\ :\ \tau_1}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ m_2\ :\ \tau_2} \right]\!\!\right]^{\mathrm{im}}_k =$$

$$\mathbf{app}\ \mathbf{w}_k\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}_1\rrbracket^{\mathrm{im}}_k\ \llbracket\mathcal{M}_2\rrbracket^{\mathrm{im}}_k$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma\{x \mapsto \tau\}) \vdash^{\mathrm{im}} m\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fix}\ x : \tau\,.\ m\ :\ \tau} \right\rrbracket^{\mathrm{im}}_k =$$

$$\mathbf{fix}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ (\lambda x_k : \mathbf{exp}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\,.\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k)$$

$$\left\llbracket \frac{\mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ :\ \tau_1 \qquad \mathcal{M}_2 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_2\ :\ \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \langle m_1, m_2\rangle\ :\ \tau_1 \times \tau_2} \right\rrbracket^{\mathrm{im}}_k =$$

$$\mathbf{pair}\ \mathbf{w}_k\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}_1\rrbracket^{\mathrm{im}}_k\ \llbracket\mathcal{M}_2\rrbracket^{\mathrm{im}}_k$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{fst}\ m\ :\ \tau_1} \right\rrbracket^{\mathrm{im}}_k = \mathbf{fst}\ \mathbf{w}_k\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau_1 \times \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{snd}\ m\ :\ \tau_2} \right\rrbracket^{\mathrm{im}}_k = \mathbf{snd}\ \mathbf{w}_k\ \llbracket\tau_1\rrbracket^{\mathrm{im}}\ \llbracket\tau_2\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k$$

$$\left\llbracket \frac{}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{z}\ :\ \mathbf{nat}} \right\rrbracket^{\mathrm{im}}_k = \mathbf{z}\ \mathbf{w}_k$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \mathbf{nat}}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{s}\ m\ :\ \mathbf{nat}} \right\rrbracket^{\mathrm{im}}_k = \mathbf{s}\ \mathbf{w}_k\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \mathbf{nat} \quad \mathcal{M}_1 :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m_1\ :\ \tau \quad \mathcal{M}_2 :: (\Psi;\ \Gamma\{x \mapsto \mathbf{nat}\}) \vdash^{\mathrm{im}} m_2\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{case}\ m\ \mathbf{of}\ \mathbf{z} \Rightarrow m_1 \mid \mathbf{s}\ x \Rightarrow m_2\ :\ \tau} \right\rrbracket^{\mathrm{im}}_k$$

$$= \mathbf{case}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k\ \llbracket\mathcal{M}_1\rrbracket^{\mathrm{im}}_k\ (\lambda x_k : \mathbf{exp}\ \mathbf{w}_k\ \mathbf{nat}\,.\ \llbracket\mathcal{M}_2\rrbracket^{\mathrm{im}}_k)$$

$$\left\llbracket \frac{\mathcal{M} :: ((\Psi;\ \Gamma);\ \cdot) \vdash^{\mathrm{im}} m\ :\ \tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{box}\ m\ :\ \Box\tau} \right\rrbracket^{\mathrm{im}}_k =$$

$$\mathbf{box}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ (\lambda \mathbf{w}_{k+1} : \mathbf{world}\,.\ \lambda \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}\,.\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k+1})$$

$$\left\llbracket \frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{unbox}\ p\ :\ \tau} \right\rrbracket^{\mathrm{im}}_k = \mathbf{unbox}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k$$

Finally the function $\boxed{\llbracket\mathcal{N}\rrbracket^{\mathrm{im}}_k}$ mapping a typing derivation for a pop expression p into an LF object M such that $\Lambda,\ \llbracket\Gamma\rrbracket^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M\ :\ \mathbf{exp}\ \mathbf{w}_k\ (\mathbf{code}\ \llbracket\tau\rrbracket^{\mathrm{im}})$ when $\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau$ and $\llbracket\Psi\rrbracket^{\mathrm{im}} = \{k;\ \Lambda\}$:

$$\left\llbracket \frac{\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \Box\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} m\ :\ \Box\tau} \right\rrbracket^{\mathrm{im}}_k = \mathbf{pop\_0}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_k$$

$$\left[\!\!\left[\frac{\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau}{((\Psi;\ \Gamma);\ \Gamma') \vdash^{\mathrm{im,p}} \mathbf{pop}\ p\ :\ \Box\tau} \right]\!\!\right]^{\mathrm{im}}_{k} =$$

$$\mathbf{pop\_m}\ \mathbf{w}_{k-1}\ \mathbf{w}_k\ \llbracket\tau\rrbracket^{\mathrm{im}}\ \mathbf{j}_{k-1}\ \llbracket\mathcal{M}\rrbracket^{\mathrm{im}}_{k-1} \text{ for } k > 0$$

With the just defined signature and representation functions each well-typed implicit expression has an LF representation and also each canonical LF object of type **exp** $W\ T$ is indeed the representation of some well-typed implicit expression. This adequacy is stated and proved more formally below. First we present some convenient definitions and lemmas though.

The first lemma describes the world variables to be found in a translated context stack. As expected, the translation function generates a world variable per stack level:

Lemma 4.4. If $\llbracket\Psi\rrbracket^{\mathrm{im}} = \{k;\ \Lambda\}$ then $\{w \mid \Lambda(w) = \mathbf{world}\} = \{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_k\}$.

Proof. The proof is done by induction on the structure of Ψ.

1. $\Psi = \ \cdot$. Then $\llbracket\Psi\rrbracket^{\mathrm{im}} = \{0;\ \mathbf{w}_0 : \mathbf{world}\}$ and we are obviously done.

2. $\Psi = (\Psi';\ \Gamma)$. Then Λ is of the form

$$\Lambda',\ \llbracket\Gamma\rrbracket^{\mathrm{im}}_{k-1},\ \mathbf{w}_k : \mathbf{world},\ \mathbf{j}_{k-1} : \mathbf{jump}\ \mathbf{w}_{k-1}\ \mathbf{w}_k$$

with $\llbracket\Psi'\rrbracket^{\mathrm{im}} = \{k-1;\ \Lambda'\}$. Applying the induction hypothesis to Ψ' we get that $\{w \mid \Lambda'(w) = \mathbf{world}\} = \{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{k-1}\}$ and as it must be the case that $\{w \mid \Lambda(w) = \mathbf{world}\} = \{w \mid \Lambda'(w) = \mathbf{world}\} \cup \{\mathbf{w}_k\}$, we are done. □

As we will need to refer to the structure of LF contexts generated by translation of environments from Mini-ML$^{\Box}$ to LF, we make the following definition:

Definition 4.2. An LF context Λ is said to be Mini-ML$^{\Box}$-*formed* iff

a) Λ only contains declarations of the form $w : \mathbf{world}$, $x : \mathbf{exp}\ W\ T$ and $j : \mathbf{jump}\ W_1\ W_2$.

b) Each world has at most one predecessor in Λ. That is, if $\Lambda(j') = \mathbf{jump}\ W'\ W$ and $\Lambda(j'') = \mathbf{jump}\ W''\ W$ then we have that $W' = W''$.

Then we state that the translation functions actually generate Mini-ML$^{\Box}$-formed LF contexts and we give a further characterization of the declarations occurring:

Lemma 4.5. Assume that $\llbracket\Psi\rrbracket^{\mathrm{im}} = \{k;\ \Lambda\}$ and consider the LF context $\Lambda' = \Lambda,\ \llbracket\Gamma\rrbracket^{\mathrm{im}}_k$ for some Mini-ML$^{\Box}$ context Γ. Then the following statements are true:

a) Λ' is Mini-ML$^{\Box}$-formed.

b) For any j with $\Lambda'(j) = \mathbf{jump}\ W\ W'$ we can find k such that $j = \mathbf{j}_k$, $W = \mathbf{w}_k$, and $W' = \mathbf{w}_{k+1}$.

c) For any x with $\Lambda'(x) = \mathbf{exp}\ W\ T$ we can find k, τ, and y such that $W = \mathbf{w}_k$, $T = [\![\tau]\!]^{\mathrm{im}}$, $\Gamma(y) = \tau$, and $x = y_k$.

Proof. The lemma can be proved by straightforward induction on the structure of Ψ and Γ. □

Finally we extract how the worlds of free variables relate to the world of the LF object within which the variables occur freely. First a definition:

Definition 4.3. w_0, w_1, ..., w_n is a *world path* in the LF context Λ iff $\Lambda(w_0) = \mathbf{world}$, $\Lambda(w_1) = \mathbf{world}$, ..., $\Lambda(w_n) = \mathbf{world}$ and we can find j_0, j_1, ..., j_{n-1} such that $\Lambda(j_0) = \mathbf{jump}\ w_0\ w_1$, $\Lambda(j_1) = \mathbf{jump}\ w_1\ w_2$, ..., $\Lambda(j_{n-1}) = \mathbf{jump}\ w_{n-1}\ w_n$.

Lemma 4.6. Assume that Λ is Mini-ML$^{\Box}$-formed, M is canonical, and that either

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M\ :\ \mathbf{exp}\ W\ T$$

or

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M\ :\ \mathbf{pop}\ W\ T.$$

Then the following statements are true:

a) If x is a free variable in M of type $\mathbf{exp}\ W'\ T'$ then Λ contains a world path from W' to W.

b) If w is a free variable in M of type **world** then Λ contains a world path from w to W.

c) If j is a free variable in M of type $\mathbf{jump}\ W'\ W''$ contains a world path from W'' to W.

Proof. Both a), b), and c) can be be proved by induction on the structure of the canonical object M. As the proofs are similar we will only write the one for a) here.

Due to the structure of the Mini-ML$^{\Box}$-formed context Λ there are no variables in the context of function type, and thus we will have no proof cases concerning variable applications. Leaving out the cases involving **fix**, **pair**, **fst**, **snd**, **z**, **s**, and **case**, which can be proved in similar ways as the cases involving **lam** and **app**, we have the following proof cases:

1. $M = x$. According to the rules for typing of LF objects x can only be of type $\mathbf{exp}\ W\ T$ if $\Lambda(x) = \mathbf{exp}\ W\ T$. Also having $\Lambda(x) = \mathbf{exp}\ W'\ T'$ we can derive that $W = W'$ and we are done.

2. $M = \mathbf{lam}\ W''\ T_1\ T_2\ (\lambda x' : \mathbf{exp}\ W''\ T_1\ .\ M')$ where x' is not present in Λ. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda,\ x' : \mathbf{exp}\ W\ T_1 \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{exp}\ W\ T_2.$$

When x is free in M it must also be free in M' and as the addition of $x' : \mathbf{exp}\ W\ T_1$ does not ruin the Mini-ML$^{\Box}$-formedness of the LF context, we can apply the induction hypothesis to M' and we are done.

3. $M = \mathbf{app}\ W''\ T_1\ T_2\ M_1\ M_2$. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M_1 : \mathbf{exp}\ W\ T_1$$

and

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M_2 : \mathbf{exp}\ W\ T_2.$$

When x is free in M it must also be free in at least one of the two objects M_1 and M_2. Assuming that x is free in M_1 we can apply the induction hypothesis to M_1 and we are done. Similar if x is free in M_2.

4. $M = \mathbf{box}\ W''\ T''\ (\lambda w : \mathbf{world}\,.\,\lambda j : \mathbf{jump}\ W''\ w\,.\,M')$ where w and j are not present in Λ. From the rules for typing of LF objects we can derive that $W'' = W$ and that

$$\Lambda' \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M' : \mathbf{exp}\ w\ T''$$

where $\Lambda' = \Lambda,\ w : \mathbf{world},\ j : \mathbf{jump}\ W\ w$. When x is free in M it must also be free in M' and as the addition of $w : \mathbf{world}$ and $j : \mathbf{jump}\ W\ w$ does not ruin Mini-ML$^{\Box}$-formedness, we can apply the induction hypothesis to M' and get w_0, w_1, ..., w_n with $\Lambda'(w_0) = \mathbf{world}$, $\Lambda'(w_1) = \mathbf{world}$, ..., $\Lambda'(w_n) = \mathbf{world}$ such that $W' = w_0$ and $w = w_n$ and such that we have j_0, j_1, ..., j_{n-1} with $\Lambda'(j_0) = \mathbf{jump}\ w_0\ w_1$, $\Lambda'(j_1) = \mathbf{jump}\ w_1\ w_2$, ..., $\Lambda'(j_{n-1}) = \mathbf{jump}\ w_{n-1}\ w_n$. As W must be the only predecessor of w in Λ' we have that $w_{n-1} = W$ and we are done.

5. $M = \mathbf{unbox}\ W''\ T''\ M'$. From the rules for typing of LF objects we can derive that $W'' = W$ and $T'' = T$ and that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M' : \mathbf{pop}\ W\ (\mathbf{code}\ T).$$

When x is free in M it must also be free in M' and applying the induction hypothesis finishes the case.

6. $M = \mathbf{pop\_0}\ W''\ T''\ M'$. From the rules for typing of LF objects we can derive that $W'' = W$ and $\mathbf{code}\ T'' = T$ and that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M' : \mathbf{exp}\ W\ T.$$

When x is free in M it must also be free in M' and applying the induction hypothesis finishes the case.

7. $M = \mathbf{pop\_m}\ W''\ W'''\ T''\ J\ M'$. From the rules for typing of LF objects we can derive that $W''' = W$ and $\mathbf{code}\ T'' = T$ and that

$$\Lambda \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M' : \mathbf{pop}\ W''\ T.$$

When x is free in M it must also be free in M' and thus we can apply the induction hypothesis and get w_0, w_1, ..., w_n with $\Lambda(w_0) = \mathbf{world}$, $\Lambda(w_1) = \mathbf{world}$, ..., $\Lambda(w_n) = \mathbf{world}$ such that $W' = w_0$ and $W'' = w_n$ and such that we have j_0, j_1, ..., j_{n-1} with $\Lambda(j_0) = \mathbf{jump}\ w_0\ w_1$, $\Lambda(j_1) = \mathbf{jump}\ w_1\ w_2$, ..., $\Lambda(j_{n-1}) = \mathbf{jump}\ w_{n-1}\ w_n$. As $\Lambda(J) = \mathbf{jump}\ W''\ W$ extending the found sequences with W and J respectively finishes the case. □

We are now ready to state and prove theorems concerning the adequacy of the functions $[\![\mathcal{M}]\!]^{\mathrm{im}}_k$ and $[\![\mathcal{N}]\!]^{\mathrm{im}}_k$.

Theorem 4.6 (Pop/Expression representation adequacy →)**.** Assuming that $[\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\}$ the following two statements will be true:

a) If $\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \tau$ then

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{M}]\!]^{\mathrm{im}}_k\ :\ \mathbf{exp}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}.$$

b) If $\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\tau$ then

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{N}]\!]^{\mathrm{im}}_k\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

Proof. The proof is done by mutual induction on the structure of the derivations $\mathcal{M}$ and $\mathcal{N}$. For a) we will only write down the proof cases where TP$^{\mathrm{im}}$-BOX and TP$^{\mathrm{im}}$-UNBOX are considered last rules applied in $\mathcal{M}$, as the rest of the cases are very similar to cases in adequacy proofs in earlier chapters. For b) both pop-typing rules are considered the last rule applied in $\mathcal{N}$.

1. TP$^{\mathrm{im}}$-BOX. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{M}' :: ((\Psi;\ \Gamma);\ \cdot) \vdash^{\mathrm{im}} m'\ :\ \tau'}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{box}\ m'\ :\ \Box\tau'}$$

and as $[\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\}$ we have that $[\![\Psi;\ \Gamma]\!]^{\mathrm{im}} = \{k+1;\ \Lambda'\}$ where

$$\Lambda' = \Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k,\ \mathbf{w}_{k+1} : \mathbf{world},\ \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}.$$

Applying the induction hypothesis to $\mathcal{M}'$ we get that

$$\Lambda',\ [\![\cdot]\!]^{\mathrm{im}}_{k+1} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1}\ :\ \mathbf{exp}\ \mathbf{w}_{k+1}\ [\![\tau']\!]^{\mathrm{im}}$$

which is the same as

$$\Lambda' \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1}\ :\ \mathbf{exp}\ \mathbf{w}_{k+1}\ [\![\tau']\!]^{\mathrm{im}}.$$

To this we can apply the TP$^{\mathrm{LF}}$-TERM-ABS rule of the logical framework twice and get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} (\lambda\mathbf{w}_{k+1} : \mathbf{world}.\ \lambda\mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}.\ [\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1})\ :$$
$$\Pi\,\mathbf{w}_{k+1} : \mathbf{world}.\ \Pi\,\mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}.\ \mathbf{exp}\ \mathbf{w}_{k+1}\ [\![\tau']\!]^{\mathrm{im}}.$$

As $\Lambda(\mathbf{w}_k) = \mathbf{world}$ according to **Lemma 4.4** and as

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\tau']\!]^{\mathrm{ex}}\ :\ \mathbf{tp}$$

due to adequacy of type representation, we can now apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework twice and get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} (\mathbf{box}\ \mathbf{w}_k\ [\![\tau']\!]^{\mathrm{im}}\ (\lambda\mathbf{w}_{k+1} : \mathbf{world}.\ \lambda\mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}.\ [\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1}))\ :$$
$$\mathbf{exp}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau']\!]^{\mathrm{im}}).$$

Having $[\![\tau]\!]^{\mathrm{im}} = [\![\Box\,\tau']\!]^{\mathrm{im}} = \mathbf{code}\ [\![\tau']\!]^{\mathrm{im}}$ and $[\![\mathcal{M}]\!]^{\mathrm{im}}_k = \mathbf{box}\ \mathbf{w}_k\ [\![\tau']\!]^{\mathrm{im}}\ (\lambda \mathbf{w}_{k+1} : \mathbf{world}\,.\,\lambda \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}\,.\ [\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1})$, we are done.

2. TP$^{\mathrm{im}}$-UNBOX. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\,\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{unbox}\ p\ :\ \tau}\,.$$

First applying the induction hypothesis of the proof of b) we get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{N}]\!]^{\mathrm{im}}_k\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}})$$

and as $\Lambda(\mathbf{w}_k) = \mathbf{world}$ according to **Lemma 4.4** and as

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\tau]\!]^{\mathrm{ex}}\ :\ \mathbf{tp}$$

due to adequacy of type representation, we can then apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} \mathbf{unbox}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{N}]\!]^{\mathrm{im}}_k\ :\ \mathbf{exp}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\,.$$

As $[\![\mathcal{M}]\!]^{\mathrm{im}}_k = \mathbf{unbox}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{N}]\!]^{\mathrm{im}}_k$, we are done.

3. TP$^{\mathrm{im,p}}$-0. In this case $\mathcal{N}$ is of the form

$$\frac{\mathcal{M} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \Box\,\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} m\ :\ \Box\,\tau}\,.$$

First applying the induction hypothesis of the proof of a) we get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\mathcal{M}]\!]^{\mathrm{im}}_k\ :\ \mathbf{exp}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}})$$

and as $\Lambda(\mathbf{w}_k) = \mathbf{world}$ according to **Lemma 4.4** and as

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} [\![\tau]\!]^{\mathrm{ex}}\ :\ \mathbf{tp}$$

due to adequacy of type representation, we can apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} \mathbf{pop\_0}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

As $[\![\mathcal{N}]\!]^{\mathrm{im}}_k = \mathbf{pop\_0}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{M}]\!]^{\mathrm{im}}_k$, we are done.

4. TP$^{\mathrm{im,p}}$-M. In this case $\mathcal{N}$ is of the form

$$\frac{\mathcal{N}' :: (\Psi';\ \Gamma') \vdash^{\mathrm{im,p}} p'\ :\ \Box\,\tau}{((\Psi';\ \Gamma');\ \Gamma) \vdash^{\mathrm{im,p}} \mathbf{pop}\ p'\ :\ \Box\,\tau}$$

and having $\Psi = (\Psi';\ \Gamma')$ an Λ' exists such that $[\![\Psi']\!]^{\mathrm{im}} = \{k-1;\ \Lambda'\}$ and

$$\Lambda = \Lambda',\ [\![\Gamma']\!]^{\mathrm{im}}_{k-1}\,,\ \mathbf{w}_k : \mathbf{world},\ \mathbf{j}_{k-1} : \mathbf{jump}\ \mathbf{w}_{k-1}\ \mathbf{w}_k.$$

Now applying the induction hypothesis we get that

$$\Lambda',\ [\![\Gamma']\!]^{\mathrm{im}}_{k-1}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ [\![\mathcal{N}']\!]^{\mathrm{im}}_{k-1}\ :\ \mathbf{pop}\ \mathbf{w}_{k-1}\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}})$$

and by then applying the weakening property of the logical framework, **Theorem 2.2**, we get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ [\![\mathcal{N}']\!]^{\mathrm{im}}_{k-1}\ :\ \mathbf{pop}\ \mathbf{w}_{k-1}\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

As $\Lambda(\mathbf{w}_{k-1}) = \mathbf{world}$, $\Lambda(\mathbf{w}_k) = \mathbf{world}$, and $\Lambda(\mathbf{j}_{k-1}) = \mathbf{jump}\ \mathbf{w}_{k-1}\ \mathbf{w}_k$ and as

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ [\![\tau]\!]^{\mathrm{ex}}\ :\ \mathbf{tp}$$

due to adequacy of type representation, we can apply the $\textsc{tp}^{\mathrm{LF}}$-TERM-APP rule of the logical framework five times and get that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ \mathbf{pop\_m}\ \mathbf{w}_{k-1}\ \mathbf{w}_k\ \ [\![\tau]\!]^{\mathrm{im}}\ \ \mathbf{j}_{k-1}\ \ [\![\mathcal{N}']\!]^{\mathrm{im}}_{k-1}\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

As $[\![\mathcal{N}]\!]^{\mathrm{im}}_{k} = \mathbf{pop\_m}\ \mathbf{w}_{k-1}\ \mathbf{w}_k\ \ [\![\tau]\!]^{\mathrm{im}}\ \ \mathbf{j}_{k-1}\ \ [\![\mathcal{N}']\!]^{\mathrm{im}}_{k-1}$, we are done. □

Theorem 4.7 (Pop/Expression representation adequacy ←)**.** Assuming that $[\![\Psi]\!]^{\mathrm{im}} = \{k;\ \Lambda\}$ the following two statements will be true:

a) If M is canonical with

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ M\ :\ \mathbf{exp}\ \mathbf{w}_k\ \ [\![\tau]\!]^{\mathrm{im}}$$

then we can find a typing derivation $\mathcal{M} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \tau$ such that $[\![\mathcal{M}]\!]^{\mathrm{im}}_{k} = M$.

b) If M is canonical with

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}\ \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}}\ M\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}})$$

then we can find a typing derivation $\mathcal{N} :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\,\tau$ such that $[\![\mathcal{N}]\!]^{\mathrm{im}}_{k} = M$.

Proof. The proof is done by mutual induction on the structure of the canonical LF object M. As according to **Lemma 4.5** the context $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k}$ is Mini-ML$^{\Box}$-formed, it cannot contain any variables of function type, and thus we will have no proof cases concerning variable applications. The cases below are the proof cases which are not completely similar to cases in earlier adequacy proofs.

1. $M = x$. According to the rule $\textsc{tp}^{\mathrm{LF}}$-TERM-VAR we must have that $(\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_{k})(x) = \mathbf{exp}\ \mathbf{w}_k\ \ [\![\tau]\!]^{\mathrm{im}}$ and due to **Lemma 4.5** we then know that M must be the variable y_k where $\Gamma(y) = \tau$. Choosing $\mathcal{M}$ to be

$$\frac{\Gamma(y) = \tau}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ y\ :\ \tau}$$

we have that $[\![\mathcal{M}]\!]^{\mathrm{im}}_{k} = y_k = M$ and we are done.

2. $M = \mathbf{lam}\ W\ T_1\ T_2\ (\lambda x : \mathbf{exp}\ W\ T_1\,.\ M')$. Assuming that x_k is a variable not present in the context $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k$, we can substitute x_k for x and reach the α-equivalent object $\mathbf{lam}\ W\ T_1\ T_2\ (\lambda x_k : \mathbf{exp}\ W\ T_1\,.\ M'')$ where $M'' = \{x_k/x\}\,M'$. From the rules for typing of LF objects we can then derive that $W = \mathbf{w}_k$ and $\mathbf{arrow}\ T_1\ T_2 = [\![\tau]\!]^{\mathrm{im}}$ and that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k,\ x_k : \mathbf{exp}\ \mathbf{w}_k\ T_1 \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M''\ :\ \mathbf{exp}\ \mathbf{w}_k\ T_2.$$

Due to a suitable variant of **Lemma 3.4** we can find types τ_1 and τ_2 such that $\tau = \tau_1 \to \tau_2$ and such that $T_1 = [\![\tau_1]\!]^{\mathrm{im}}$ and $T_2 = [\![\tau_2]\!]^{\mathrm{im}}$ and thus we can rewrite the last judgement into

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k,\ x_k : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M''\ :\ \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_2]\!]^{\mathrm{im}}.$$

Then by application of a suitable variant of **Lemma 3.6** we can rewrite this furtherly into

$$\Lambda,\ x_k : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}},\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M''\ :\ \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_2]\!]^{\mathrm{im}}.$$

As $\cdot,\ x_k : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}},\ [\![\Gamma]\!]^{\mathrm{im}}_k = [\![\Gamma\{x \mapsto \tau_1\}]\!]^{\mathrm{im}}_k$ we can now apply the induction hypothesis to M'' and get a derivation $\mathcal{M}' :: (\Psi;\ \Gamma\{x \mapsto \tau_1\}) \vdash^{\mathrm{im}} m' : \tau_2$ such that $[\![\mathcal{M}']\!]^{\mathrm{im}}_k = M''$ and then by application of the TP$^{\mathrm{im}}$-ABS rule we get a derivation of $(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \lambda x : \tau_1\,.\ m' : \tau_1 \to \tau_2$. As $\tau_1 \to \tau_2 = \tau$ and as the derivation is represented by the LF object $\mathbf{lam}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}}\ [\![\tau_2]\!]^{\mathrm{im}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}_k\ [\![\tau_1]\!]^{\mathrm{im}}\ .\ [\![\mathcal{M}']\!]^{\mathrm{im}}_k)$ which is α-equivalent to M, we are done.

3. $M = \mathbf{box}\ W\ T\ (\lambda w : \mathbf{world}\,.\,\lambda j : \mathbf{jump}\ W\ w\,.\ M')$. According to **Lemma 4.4** and the fact that $[\![\Gamma]\!]^{\mathrm{im}}_k$ does not introduce any world declarations we have that $\mathbf{w}_{k+1} : \mathbf{world}$ is not in $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k$ and therefore also that $\mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}$ is not in $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k$. The variables $\mathbf{w}_{k+1}$ and $\mathbf{j}_k$ thus cannot be free in M and we are allowed to substitute them for w and j respectively and reach the α-equivalent object $\mathbf{box}\ W\ T\ (\lambda \mathbf{w}_{k+1} : \mathbf{world}\,.\,\lambda \mathbf{j}_k : \mathbf{jump}\ W\ \mathbf{w}_{k+1}\,.\ M'')$ where $M'' = \{\mathbf{j}_k/j\}\,\{\mathbf{w}_{k+1}/w\}\,M'$. From the rules for typing of LF objects we can now derive that $W = \mathbf{w}_k$ and $\mathbf{code}\ T = [\![\tau]\!]^{\mathrm{im}}$ and that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k,\ \mathbf{w}_{k+1} : \mathbf{world},\ \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M''\ :\ \mathbf{exp}\ \mathbf{w}_{k+1}\ T.$$

As we due to a suitable variant of **Lemma 3.4** can find a type τ' such that $\tau = \Box\,\tau'$ and $T = [\![\tau']\!]^{\mathrm{im}}$ and as

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k,\ \mathbf{w}_{k+1} : \mathbf{world},\ \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1} = \Lambda',\ [\![\cdot]\!]^{\mathrm{im}}_{k+1}$$

where Λ' is given by $[\![\Psi;\ \Gamma]\!]^{\mathrm{im}} = \{k+1;\ \Lambda'\}$ this last judgement can be rewritten into

$$\Lambda',\ [\![\cdot]\!]^{\mathrm{im}}_{k+1} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M''\ :\ \mathbf{exp}\ \mathbf{w}_{k+1}\ [\![\tau']\!]^{\mathrm{im}}.$$

We can now apply the induction hypothesis to M'' and get a derivation $\mathcal{M}' :: ((\Psi;\ \Gamma);\ \cdot) \vdash^{\mathrm{im}} m' : \tau'$ such that $[\![\mathcal{M}']\!]^{\mathrm{im}}_{k+1} = M''$ and then we can apply the TP$^{\mathrm{im}}$-BOX rule and get a derivation of $(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{box}\ m' : \Box\,\tau'$. As $\Box\,\tau' = \tau$ and as the derivation is represented by the LF object $\mathbf{box}\ \mathbf{w}_k\ [\![\tau']\!]^{\mathrm{im}}\ (\lambda \mathbf{w}_{k+1} : \mathbf{world}\,.\,\lambda \mathbf{j}_k : \mathbf{jump}\ \mathbf{w}_k\ \mathbf{w}_{k+1}\,.\ [\![\mathcal{M}']\!]^{\mathrm{im}})$ which is α-equivalent to M, we are done.

4. $M = \mathbf{unbox}\ W\ T\ M'$. From the rules for typing of canonical LF objects we can derive that $W = \mathbf{w}_k$ and $T = [\![\tau]\!]^{\mathrm{im}}$ and that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{pop}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

Applying the induction hypothesis of the proof of b) we get a derivation $\mathcal{N} :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p\ :\ \Box\,\tau$ such that $[\![\mathcal{N}]\!]^{\mathrm{im}}_k = M'$ and then applying the rule TP$^{\mathrm{im}}$-UNBOX we get a derivation of $(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{unbox}\ p\ :\ \tau$. As this derivation is represented by the LF object $\mathbf{unbox}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{N}]\!]^{\mathrm{im}}_k$ which is α-equivalent to M we are done.

5. $M = \mathbf{pop\_0}\ W\ T\ M'$. From the rules for typing of LF objects we can derive that $W = \mathbf{w}_k$ and $T = [\![\tau]\!]^{\mathrm{im}}$ and that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{exp}\ \mathbf{w}_k\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

As $\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}} = [\![\Box\,\tau]\!]^{\mathrm{im}}$ we can apply the induction hypothesis of the proof of a) and get a derivation $\mathcal{M}' :: (\Psi;\ \Gamma) \vdash^{\mathrm{im}} m\ :\ \Box\,\tau$ such that $[\![\mathcal{M}']\!]^{\mathrm{im}}_k = M'$. By application of the rule TP$^{\mathrm{im,p}}$-0 we then get a derivation of $(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} m\ :\ \Box\,\tau$ and as this is represented by the LF object $\mathbf{pop\_0}\ \mathbf{w}_k\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{M}']\!]^{\mathrm{im}}_k$ which is α-equivalent to M, we are done.

6. $M = \mathbf{pop\_m}\ W\ W'\ T\ J\ M'$. From the rules for typing of LF objects we can derive that $W' = \mathbf{w}_k$ and $T = [\![\tau]\!]^{\mathrm{im}}$ and that

$$\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{pop}\ W\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

Also we have that $(\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k)(J) = \mathbf{jump}\ W\ \mathbf{w}_k$ and thus we know from **Lemma 4.5** that $W = \mathbf{w}_{k-1}$ and $J = \mathbf{j}_{k-1}$. Now, as $\Lambda,\ [\![\Gamma]\!]^{\mathrm{im}}_k$ contains both $\mathbf{w}_{k-1}$ and $\mathbf{w}_k$, the context stack Ψ must be of the form $(\Psi';\ \Gamma')$ and

$$\Lambda = \Lambda',\ [\![\Gamma']\!]^{\mathrm{im}}_{k-1},\ \mathbf{w}_k : \mathbf{world},\ \mathbf{j}_{k-1} : \mathbf{jump}\ \mathbf{w}_{k-1}\ \mathbf{w}_k$$

where $[\![\Psi']\!]^{\mathrm{im}} = \{k-1;\ \Lambda'\}$ and therefore we have that

$$\Lambda',\ [\![\Gamma']\!]^{\mathrm{im}}_{k-1},\ \mathbf{w}_k : \mathbf{world},\ \mathbf{j}_{k-1} : \mathbf{jump}\ \mathbf{w}_{k-1}\ \mathbf{w}_k,\ [\![\Gamma]\!]^{\mathrm{im}}_k \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{pop}\ \mathbf{w}_{k-1}\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}}).$$

According to **Lemma 4.5** and **Lemma 4.6** and the weakening property described in **Theorem 2.3** we can here strip off $\mathbf{w}_k$, $\mathbf{j}_{k-1}$ and $[\![\Gamma]\!]^{\mathrm{im}}_k$ and get that

$$\Lambda',\ [\![\Gamma']\!]^{\mathrm{im}}_{k-1} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{im}}} M'\ :\ \mathbf{pop}\ \mathbf{w}_{k-1}\ (\mathbf{code}\ [\![\tau]\!]^{\mathrm{im}})$$

and then by application of the induction hypothesis to M' we get a derivation $\mathcal{N}' :: (\Psi';\ \Gamma') \vdash^{\mathrm{im,p}} p\ :\ \Box\,\tau$ such that $[\![\mathcal{N}']\!]^{\mathrm{im}}_{k-1} = M'$. Now by application of the rule TP$^{\mathrm{im,p}}$-M we get a derivation of $(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} \mathbf{pop}\ p\ :\ \Box\,\tau$ and as this derivation is represented by the LF object $\mathbf{pop\_m}\ \mathbf{w}_{k-1}\ \mathbf{w}_k\ \mathbf{j}_{k-1}\ [\![\tau]\!]^{\mathrm{im}}\ [\![\mathcal{N}']\!]^{\mathrm{im}}_k$ which is α-equivalent to M, we are done. □

4.3 Translation from implicit into explicit language

As explained in the introduction of the current chapter, Davies and Pfenning do not define an operational semantics for the implicit language Mini-ML$^{\Box}$. Instead they present a type preserving translation from the implicit language into the explicit language Mini-ML$^{\Box}_{\mathrm{ex}}$.

The task of translating a Mini-ML$^{\Box}$ program into a Mini-ML$^{\Box}_{\mathrm{ex}}$ program consists of having all unbox expressions replaced by appropriate let-box-constructions. For instance the following program should be translated into the program *power*$^{\mathrm{ex}}$ from Section **4.1**:

$$
\begin{array}{rcl}
power^{\mathrm{im}} & \equiv & \mathbf{fix}\ p : \mathbf{nat} \to \Box\,(\mathbf{nat} \to \mathbf{nat})\,. \\
& & \quad \lambda n : \mathbf{nat}\,. \\
& & \quad\quad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \quad\ \Rightarrow \mathbf{box}\ (\lambda x : \mathbf{nat}\,.\ \mathbf{s}\ \mathbf{z}) \\
& & \quad\quad\quad\ \mid\ \mathbf{s}\ m \Rightarrow \mathbf{box}\ (\lambda x : \mathbf{nat}\,.\ \mathit{times}\ x \\
& & \quad\quad\quad\quad\quad\quad\quad\quad (\mathbf{unbox}\ (\mathbf{pop}\ (p\ m))\ x))
\end{array}
$$

The translation method described by Davies and Pfenning works with expressions containing free variables. As this is not easily represented within LF, we will here present a variant of the translation method which can be directly represented within LF.

4.3.1 Syntax of the translation

Whereas Davies and Pfenning replace unbox-sub-terms with free modal variables during translation of an implicit term into its corresponding explicit term, we use numbered labels. The nth label at stage m will be denoted

$$l_n^m$$

The replaced terms are kept in a list structure for later re-inclusion as let-box-constructions. We define the following lists structures:

$$
\begin{array}{rrcl}
\text{Expression lists:} & S & ::= & \cdot \mid e,\, S \\
\text{Lists of expression lists:} & Z & ::= & \cdot \mid Z;\, S
\end{array}
$$

The length of an expression list S, $\boxed{|S|}$, is defined by

$$
\begin{array}{rcl}
|\cdot| & = & 0 \\
|(e,\, S)| & = & 1 + |S|
\end{array}
$$

and the judgement $\boxed{S + e \overset{\mathrm{end}}{\rightsquigarrow} S'}$ appending an expression e to the end of a list S is defined by

$$
\frac{}{\cdot + e \overset{\mathrm{end}}{\rightsquigarrow} (e,\, \cdot)}\ \textsc{end}^{\mathrm{S}}\text{-0} \qquad \frac{S + e \overset{\mathrm{end}}{\rightsquigarrow} S'}{(e_0,\, S) + e \overset{\mathrm{end}}{\rightsquigarrow} (e_0,\, S')}\ \textsc{end}^{\mathrm{S}}\text{-N}
$$

Now we define the translation relation $\boxed{m \,/\, Z \ {}^{\mathrm{im}}\!\rightsquigarrow^{\mathrm{ex}}\ e \,/\, Z'}$ by the following rules:

$$
\frac{}{x \,/\, Z \ {}^{\mathrm{im}}\!\rightsquigarrow^{\mathrm{ex}}\ x \,/\, Z}\ \textsc{tr}^{\mathrm{im,ex}}\text{-VAR}
$$

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\lambda x : \tau \,.\, m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \lambda x : \tau \,.\, e \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-ABS}$$

$$\frac{m_1 \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_1 \,/\, Z' \qquad m_2 \,/\, Z' \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_2 \,/\, Z''}{m_1\; m_2 \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_1\; e_2 \,/\, Z''}\;\text{TR}^{\text{im,ex}}\text{-APP}$$

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\mathbf{fix}\; x : \tau \,.\, m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{fix}\; x : \tau \,.\, e \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-FIX}$$

$$\frac{m_1 \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_1 \,/\, Z' \qquad m_2 \,/\, Z' \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_2 \,/\, Z''}{\langle m_1,\, m_2\rangle \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \langle e_1,\, e_2\rangle \,/\, Z''}\;\text{TR}^{\text{im,ex}}\text{-PAIR}$$

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\mathbf{fst}\; m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{fst}\; e \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-FST} \qquad \frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\mathbf{snd}\; m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{snd}\; e \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-SND}$$

$$\frac{}{\mathbf{z} \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{z} \,/\, Z}\;\text{TR}^{\text{im,ex}}\text{-ZERO} \qquad \frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\mathbf{s}\; m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{s}\; e \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-SUCC}$$

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z' \qquad m_1 \,/\, Z' \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_1 \,/\, Z'' \qquad m_2 \,/\, Z'' \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e_2 \,/\, Z'''}{\mathbf{case}\; m\; \mathbf{of}\; \mathbf{z} \Rightarrow m_1 \mid \mathbf{s}\; x \Rightarrow m_2 \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{case}\; e\; \mathbf{of}\; \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\; x \Rightarrow e_2 \,/\, Z'''}\;\text{TR}^{\text{im,ex}}\text{-CASE}$$

$$\frac{m \,/\, (Z;\, \cdot) \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, (Z';\, S)}{\mathbf{box}\; m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; (\mathbf{box}\; e)\,[S] \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-BOX}$$

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}{\mathbf{unbox}\; m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; \mathbf{let}\; \mathbf{box}\; u = e\; \mathbf{in}\; u \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-UNBOX-0}$$

$$\frac{p \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; l_n^m \,/\, Z'}{\mathbf{unbox}\; (\mathbf{pop}\; p) \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; l_n^m \,/\, Z'}\;\text{TR}^{\text{im,ex}}\text{-UNBOX-M}$$

where $\boxed{p \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z'}$ is defined by

$$\frac{m \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z' \qquad S + e \;\overset{\text{end}}{\rightsquigarrow}\; S'}{m \,/\, (Z;\, S) \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; l^0_{|S|} \,/\, (Z';\, S')}\;\text{TR}^{\text{im,ex}}\text{-POP-0}$$

$$\frac{p \,/\, Z \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; l_n^m \,/\, Z'}{\mathbf{pop}\; p \,/\, (Z;\, S) \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; l_n^{m+1} \,/\, (Z';\, S)}\;\text{TR}^{\text{im,ex}}\text{-POP-M}$$

and $\boxed{e\,[S]}$ is defined by

$$\begin{array}{rcl} e\,[\cdot] & = & \downarrow e \\ e\,[(e_0,\, S)] & = & \textbf{let box}\ u = e_0\ \textbf{in}\ ((e\,[u/l_0^0])\,[S]) \end{array}$$

where $\boxed{\downarrow e}$ is defined by

$$\begin{array}{rcl} \downarrow x & = & x \\ \downarrow u & = & u \\ \downarrow l_n^{m+1} & = & l_n^m \\ \downarrow (e_1\ e_2) & = & (\downarrow e_1)\ (\downarrow e_2) \\ \downarrow (\lambda x : \tau\,.\ e) & = & \lambda x : \tau\,.\ (\downarrow e) \\ \downarrow (\textbf{box}\ e) & = & \textbf{box}\ (\downarrow e) \\ \downarrow (\textbf{let box}\ u' = e_0\ \textbf{in}\ e) & = & \textbf{let box}\ u' = (\downarrow e_0)\ \textbf{in}\ (\downarrow e) \end{array}$$

and $\boxed{e\,[u/l_0^0]}$ is defined by

$$\begin{array}{rcl} x\,[u/l_0^0] & = & x \\ u'\,[u/l_0^0] & = & u' \\ l_0^0\,[u/l_0^0] & = & u \\ l_{n+1}^0\,[u/l_0^0] & = & l_n^0 \\ l_n^{m+1}\,[u/l_0^0] & = & l_n^{m+1} \\ (e_1\ e_2)\,[u/l_0^0] & = & (e_1\,[u/l_0^0])\ (e_2\,[u/l_0^0]) \\ (\lambda x : \tau\,.\ e)\,[u/l_0^0] & = & \lambda x : \tau\,.\ (e\,[u/l_0^0]) \\ (\textbf{box}\ e)\,[u/l_0^0] & = & \textbf{box}\ (e\,[u/l_0^0]) \\ (\textbf{let box}\ u' = e_0\ \textbf{in}\ e)\,[u/l_0^0] & = & \textbf{let box}\ u' = (e_0\,[u/l_0^0])\ \textbf{in}\ (e\,[u/l_0^0]) \end{array}$$

4.3.2 Type preservation of the translation

To prove that the expression translation defined above is type preserving we need some extra definitions:

We define lists of types and lists of such lists by

$$\begin{array}{rrcl} \text{Type lists:} & \Sigma & ::= & \cdot \mid \tau,\, \Sigma \\ \text{Lists of type lists:} & \Omega & ::= & \cdot \mid \Omega;\, \Sigma \end{array}$$

The judgement $\boxed{\Sigma + \tau \overset{\text{end}}{\rightsquigarrow} \Sigma'}$ appending a type at the end of a type list is defined by

$$\frac{}{\cdot + \tau \overset{\text{end}}{\rightsquigarrow} (\tau, \cdot)}\ \text{END}^{\Sigma}\text{-0} \qquad \frac{\Sigma + \tau \overset{\text{end}}{\rightsquigarrow} \Sigma'}{(\tau_0, \Sigma) + \tau \overset{\text{end}}{\rightsquigarrow} (\tau_0, \Sigma')}\ \text{END}^{\Sigma}\text{-N}$$

Prefix relations:

The prefix relation $\boxed{\Sigma \leq^{\Sigma} \Sigma'}$ is defined by

$$\frac{}{\cdot \leq^{\Sigma} \Sigma}\ \text{PREF}^{\Sigma}\text{-0} \qquad \frac{\Sigma \leq^{\Sigma} \Sigma'}{(\tau, \Sigma) \leq^{\Sigma} (\tau, \Sigma')}\ \text{PREF}^{\Sigma}\text{-N}$$

The prefix relation $\boxed{\Omega \leq^{\Omega} \Omega'}$ is defined by

$$\frac{}{\cdot \leq^{\Omega} \cdot}\ \text{PREF}^{\Omega}\text{-0} \qquad \frac{\Omega \leq^{\Omega} \Omega' \qquad \Sigma \leq^{\Sigma} \Sigma'}{(\Omega; \Sigma) \leq^{\Omega} (\Omega'; \Sigma')}\ \text{PREF}^{\Omega}\text{-M}$$

Typing judgements:

The judgement $\boxed{\Omega \mid \Gamma \vdash^{\text{S}} S : \Sigma}$ typing an expression list S is defined by

$$\frac{}{\Omega \mid \Gamma \vdash^{\text{S}} \cdot : \cdot}\ \text{TP}^{\text{S}}\text{-0} \qquad \frac{\Omega \mid \Gamma \vdash^{\text{S}} S : \Sigma \qquad \Omega \mid \cdot \mid \Gamma \vdash^{\text{ex}^l} e : \Box\tau}{\Omega \mid \Gamma \vdash^{\text{S}} (e, S) : (\tau, \Sigma)}\ \text{TP}^{\text{S}}\text{-N}$$

The judgement $\boxed{\Psi \vdash^{\text{Z}} Z : \Omega}$ typing a list of expression lists is defined by

$$\frac{}{\cdot \vdash^{\text{Z}} \cdot : \cdot}\ \text{TP}^{\text{Z}}\text{-0} \qquad \frac{\Psi \vdash^{\text{Z}} Z : \Omega \qquad \Omega \mid \Gamma \vdash^{\text{S}} S : \Sigma}{(\Psi; \Gamma) \vdash^{\text{Z}} (Z; S) : (\Omega; \Sigma)}\ \text{TP}^{\text{Z}}\text{-M}$$

The judgement $\boxed{\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau}$ typing an explicit expression that may contain labels is defined by

$$\frac{\Gamma(x) = \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} x : \tau}\ \text{TP}^{\text{ex}^l}\text{-VAR-X} \qquad \frac{\Delta(u) = \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} u : \tau}\ \text{TP}^{\text{ex}^l}\text{-VAR-U}$$

$$\frac{\Omega \vdash^{l^{\Omega}} (m, n) : \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} l^m_n : \tau}\ \text{TP}^{\text{ex}^l}\text{-VAR-L}$$

$$\frac{\Omega \mid \Delta \mid \Gamma\{x \mapsto \tau_1\} \vdash^{\text{ex}^l} e : \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} \lambda x : \tau_1 \,.\, e : \tau_1 \to \tau_2}\ \text{TP}^{\text{ex}^l}\text{-ABS}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_1 \,:\, \tau_1 \to \tau_2 \qquad \Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_2 \,:\, \tau_1}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_1\ e_2 \,:\, \tau_2}\ \text{TP}^{\mathrm{ex}^l}\text{-APP}$$

$$\frac{\Omega \mid \Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\mathrm{ex}^l} e \,:\, \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{fix}\, x : \tau\,.\, e \,:\, \tau}\ \text{TP}^{\mathrm{ex}^l}\text{-FIX}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_1 \,:\, \tau_1 \qquad \Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_2 \,:\, \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \langle e_1, e_2 \rangle \,:\, \tau_1 \times \tau_2}\ \text{TP}^{\mathrm{ex}^l}\text{-PAIR}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e \,:\, \tau_1 \times \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{fst}\ e \,:\, \tau_1}\ \text{TP}^{\mathrm{ex}^l}\text{-FST} \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e \,:\, \tau_1 \times \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{snd}\ e \,:\, \tau_2}\ \text{TP}^{\mathrm{ex}^l}\text{-SND}$$

$$\frac{}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{z} \,:\, \mathbf{nat}}\ \text{TP}^{\mathrm{ex}^l}\text{-ZERO} \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e \,:\, \mathbf{nat}}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{s}\ e \,:\, \mathbf{nat}}\ \text{TP}^{\mathrm{ex}^l}\text{-SUCC}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e \,:\, \mathbf{nat} \qquad \Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_1 \,:\, \tau \qquad \Omega \mid \Delta \mid \Gamma\{x \mapsto \mathbf{nat}\} \vdash^{\mathrm{ex}^l} e_2 \,:\, \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \,:\, \tau}\ \text{TP}^{\mathrm{ex}^l}\text{-CASE}$$

$$\frac{\Omega \mid \Delta \mid \cdot \vdash^{\mathrm{ex}^l} e \,:\, \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{box}\ e \,:\, \Box\tau}\ \text{TP}^{\mathrm{ex}^l}\text{-BOX}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e_0 \,:\, \Box\tau_0 \qquad \Omega \mid \Delta\{u \mapsto \tau_0\} \mid \Gamma \vdash^{\mathrm{ex}^l} e \,:\, \tau}{\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{let\ box}\ u = e_0\ \mathbf{in}\ e \,:\, \tau}\ \text{TP}^{\mathrm{ex}^l}\text{-LET-BOX}$$

where the judgement $\boxed{\Omega \vdash^{l^\Omega} (m, n) \,:\, \tau}$ is defined by

$$\frac{\Sigma \vdash^{l^\Sigma} n \,:\, \tau}{(\Omega; \Sigma) \vdash^{l^\Omega} (0, n) \,:\, \tau}\ \text{TP}^{l^\Omega}\text{-0} \qquad \frac{\Omega \vdash^{l^\Omega} (m, n) \,:\, \tau}{(\Omega; \Sigma) \vdash^{l^\Omega} (m+1, n) \,:\, \tau}\ \text{TP}^{l^\Omega}\text{-M}$$

where the judgement $\boxed{\Sigma \vdash^{l^\Sigma} n \,:\, \tau}$ is defined by

$$\frac{}{(\tau, \Sigma) \vdash^{l^\Sigma} 0 \,:\, \tau}\ \text{TP}^{l^\Sigma}\text{-0} \qquad \frac{\Sigma \vdash^{l^\Sigma} n \,:\, \tau'}{(\tau, \Sigma) \vdash^{l^\Sigma} n+1 \,:\, \tau'}\ \text{TP}^{l^\Sigma}\text{-N}$$

The $\text{TP}^{\mathrm{ex}^l}$ rules are very similar to the TP^{ex} rules in Section **4.1** except from the additional rule $\text{TP}^{\mathrm{ex}^l}$-VAR-L for typing of label expressions.

Lemma 4.7 (Transitivity of Σ-prefix)**.** If $\Sigma \leq^{\Sigma} \Sigma'$ and $\Sigma' \leq^{\Sigma} \Sigma''$ then $\Sigma \leq^{\Sigma} \Sigma''$.

Proof. The proof is done by induction on the derivation of $\Sigma \leq^{\Sigma} \Sigma'$. If PREF$^{\Sigma}$-0 is the last rule applied we must have $\Sigma = \cdot$ and thus we can apply PREF$^{\Sigma}$-0 and get $\Sigma \leq^{\Sigma} \Sigma''$. If instead PREF$^{\Sigma}$-N is the last rule applied we must have τ_0, Σ_0, and Σ_0' such that $\Sigma = (\tau_0, \Sigma_0)$, $\Sigma' = (\tau_0, \Sigma_0')$, and $\Sigma_0 \leq^{\Sigma} \Sigma_0'$. From $\Sigma' = (\tau_0, \Sigma_0')$ we can derive that PREF$^{\Sigma}$-N must also be the last rule applied in the derivation of $\Sigma' \leq^{\Sigma} \Sigma''$ and thus we must have Σ_0'' such that $\Sigma'' = (\tau_0, \Sigma_0'')$ and $\Sigma_0' \leq^{\Sigma} \Sigma_0''$. Now applying the induction hypothesis we get $\Sigma_0 \leq^{\Sigma} \Sigma_0''$ and finally applying the rule PREF$^{\Sigma}$-N we can conclude that $\Sigma \leq^{\Sigma} \Sigma''$. □

Lemma 4.8 (Transitivity of Ω-prefix)**.** If $\Omega \leq^{\Omega} \Omega'$ and $\Omega' \leq^{\Omega} \Omega''$ then $\Omega \leq^{\Omega} \Omega''$.

Proof. The proof is done by induction on the derivation of $\Omega \leq^{\Omega} \Omega'$. If PREF$^{\Omega}$-0 is the last rule applied it is obvious that PREF$^{\Omega}$-0 must also be the last rule applied in the derivation of $\Omega \leq^{\Omega} \Omega''$ and we have that Ω, Ω', Ω'' are all empty. From PREF$^{\Omega}$-0 it is then known that $\Omega \leq^{\Omega} \Omega''$. If instead PREF$^{\Omega}$-M is the last rule applied we must have Ω_0, Σ_0, Ω_0', and Σ_0' such that $\Omega = (\Omega_0; \Sigma_0)$, $\Omega' = (\Omega_0'; \Sigma_0')$, $\Omega_0 \leq^{\Omega} \Omega_0'$, and $\Sigma_0 \leq^{\Sigma} \Sigma_0'$. From $\Omega' = (\Omega_0'; \Sigma_0')$ we can derive that PREF$^{\Omega}$-M must also be the last rule applied in the derivation of $\Omega' \leq^{\Omega} \Omega''$ and thus we must have Ω_0'' and Σ_0'', Ω_0' such that $\Omega'' = (\Omega_0''; \Sigma_0'')$, $\Omega_0' \leq^{\Omega} \Omega_0''$, and $\Sigma_0' \leq^{\Sigma} \Sigma_0''$. Now applying the induction hypothesis we get $\Omega_0 \leq^{\Omega} \Omega_0''$ and applying **Lemma 4.7** we get $\Sigma_0 \leq^{\Sigma} \Sigma_0''$. Finally applying PREF$^{\Omega}$-M we get $\Omega \leq^{\Omega} \Omega''$ as wanted. □

Lemma 4.9 (Reflexivity of Σ-prefix)**.** $\Sigma \leq^{\Sigma} \Sigma$.

Proof. The proof can be done by straightforward induction on the structure of Σ. □

Lemma 4.10 (Reflexivity of Ω-prefix)**.** $\Omega \leq^{\Omega} \Omega$.

Proof. The proof can be done by straightforward induction on the structure of Ω and by use of **Lemma 4.9**. □

Lemma 4.11. If $\Sigma + \tau \overset{\text{end}}{\rightsquigarrow} \Sigma'$ then $\Sigma \leq^{\Sigma} \Sigma'$.

Proof. The proof can be done by straightforward induction on the structure of Σ. □

Lemma 4.12 (Σ-weakening)**.** If $\Sigma \vdash^{l^{\Sigma}} n : \tau$ and $\Sigma \leq^{\Sigma} \Sigma'$ then $\Sigma' \vdash^{l^{\Sigma}} n : \tau$.

Proof. The proof is done by induction on n. Independent of the value of n we can derive from the TP$^{l^{\Sigma}}$ rules that we have τ_0 and Σ_0 such that $\Sigma = (\tau_0, \Sigma_0)$ and then from PREF$^{\Sigma}$-N that we have Σ_0' such that $\Sigma' = (\tau_0, \Sigma_0')$ and $\Sigma_0 \leq^{\Sigma} \Sigma_0'$. If $n = 0$ we get from TP$^{l^{\Sigma}}$-0 that $\tau_0 = \tau$ and we can apply TP$^{l^{\Sigma}}$-0 to get $\Sigma' \vdash^{l^{\Sigma}} n : \tau$. If instead $n = n_0 + 1$ for some $n_0 \geq 0$ we get from TP$^{l^{\Sigma}}$-N that $\Sigma_0 \vdash^{l^{\Sigma}} n_0 : \tau$ and applying the induction hypothesis to this and $\Sigma_0 \leq^{\Sigma} \Sigma_0'$ we get that $\Sigma_0' \vdash^{l^{\Sigma}} n_0 : \tau$. That $\Sigma' \vdash^{l^{\Sigma}} n_0 + 1 : \tau$ now follows from an application of TP$^{l^{\Sigma}}$-N. □

Lemma 4.13 (Ω-weakening 1)**.** If $\Omega \vdash^{l^\Omega} (m, n) : \tau$ and $\Omega \leq^\Omega \Omega'$ then $\Omega' \vdash^{l^\Omega} (m, n) : \tau$.

Proof. The proof is done by induction on m. Independent of the value of m we can derive from the TP^{l^Ω} rules that we have Ω_0 and Σ_0 such that $\Omega = (\Omega_0; \Sigma_0)$ and then from PREF^Ω-M that we have Ω'_0 and Σ'_0 such that $\Omega' = (\Omega'_0; \Sigma'_0)$ and such that $\Omega_0 \leq^\Omega \Omega'_0$ and $\Sigma_0 \leq^\Sigma \Sigma'_0$. If $m = 0$ the rule TP^{l^Ω}-0 implies that $\Sigma_0 \vdash^{l^\Sigma} n : \tau$ and applying **Lemma 4.12** to this and $\Sigma_0 \leq^\Sigma \Sigma'_0$ we get that $\Sigma'_0 \vdash^{l^\Sigma} n : \tau$. Now applying the rule TP^{l^Ω}-0 we get $(\Omega'_0; \Sigma'_0) \vdash^{l^\Omega} (m, n) : \tau$ as wanted. If instead $m = m_0 + 1$ for some $m_0 \geq 0$ we can derive from TP^{l^Ω}-M that $\Omega_0 \vdash^{l^\Omega} (m, n) : \tau$ and applying the induction hypothesis to this and $\Omega_0 \leq^\Omega \Omega'_0$ we get that $\Omega'_0 \vdash^{l^\Omega} (m, n) : \tau$. That $(\Omega'_0; \Sigma'_0) \vdash^{l^\Omega} (m, n) : \tau$ now follows from application of TP^{l^Ω}-M. □

Lemma 4.14 (Ω-weakening 2)**.** If $\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$ and $\Omega \leq^\Omega \Omega'$ then $\Omega' \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$.

Proof. The proof can be done by straightforward induction on the derivation of the typing $\Omega \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$ using **Lemma 4.13** in the case of TP^{ex^l}-VAR-L being the last rule applied. □

Lemma 4.15 (Ω-weakening 3)**.** If $\Omega \mid \Gamma \vdash^{\text{S}} S : \Sigma$ and $\Omega \leq^\Omega \Omega'$ then $\Omega' \mid \Gamma \vdash^{\text{S}} S : \Sigma$

Proof. The proof can be done by straightforward induction on the structure of S and by use of **Lemma 4.14**. □

Lemma 4.16 (l^0_0-substitution)**.** If $(\Omega; (\tau_0, \Sigma)) \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$ then $(\Omega; \Sigma) \mid (\Delta, u :: \tau_0) \mid \Gamma \vdash^{\text{ex}^l} e\,[u/l^0_0] : \tau$.

Proof. The proof is done by induction on the derivation $\mathcal{I}$ of the typing $(\Omega; (\tau_0, \Sigma)) \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$. Below each typing rule is in turn considered the last rule applied in the derivation.

1. TP^{ex^l}-VAR-X. In this case $\mathcal{I}$ is of the form

$$\frac{x : \tau \in \Gamma}{(\Omega; (\tau_0, \Sigma)) \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} x : \tau}\,.$$

From $x : \tau \in \Gamma$ we get $(\Omega; \Sigma) \mid (\Delta, u :: \tau_0) \mid \Gamma \vdash^{\text{ex}^l} x : \tau$ by applying TP^{ex^l}-VAR-X and as $x\,[u/l^0_0] = x$ we are done.

2. TP^{ex^l}-VAR-U. In this case $\mathcal{I}$ is of the form

$$\frac{u' :: \tau \in \Delta}{(\Omega; (\tau_0, \Sigma)) \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} u' : \tau}\,.$$

From $u' :: \tau \in \Delta$ we get $(\Omega; \Sigma) \mid (\Delta, u :: \tau_0) \mid \Gamma \vdash^{\text{ex}^l} u' : \tau$ by applying TP^{ex^l}-VAR-U and as $u'\,[u/l^0_0] = u'$ we are done.

3. TP^{ex^l}-VAR-L. In this case $\mathcal{I}$ is of the form

$$\frac{\mathcal{K} :: (\Omega;\, (\tau_0,\, \Sigma)) \;\vdash^{l^\Omega}\; (m,\, n) \;:\; \tau}{(\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; l_n^m \;:\; \tau}\,.$$

If $m = 0$ we know from TP^{l^Ω}-0 that $(\tau_0,\, \Sigma) \;\vdash^{l^\Sigma}\; n \;:\; \tau$. If also $n = 0$ we can derive from TP^{l^Σ}-0 that $\tau = \tau_0$ and as $l_0^0\,[u/l_0^0] = u$ we get from TP^{ex^l}-VAR-U that $\Sigma \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; e\,[u/l_0^0] \;:\; \tau$ as wanted. If instead $n = n' + 1$ for some $n' \geq 0$ we can derive from TP^{l^Σ}-N that $\Sigma \;\vdash^{l^\Sigma}\; n' \;:\; \tau$ and then by applying first TP^{l^Σ}-N and then TP^{ex^l}-VAR-L we get $\Sigma \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; l_{n'}^0 \;:\; \tau$. As $l_n^0\,[u/l_0^0] = l_{n'}^0$ we are done. Now, if $m = m' + 1$ for some $m' \geq 1$ we get from TP^{ex^l}-VAR-L that $\Omega \;\vdash^{l^\Omega}\; (m',\, n) \;:\; \tau$ and then by applying TP^{ex^l}-VAR-L to this we get $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; l_n^{m'+1} \;:\; \tau$. As $l_n^{m'+1}\,[u/l_0^0] = l_n^{m'+1}$ we are done.

4. TP^{ex^l}-APP. In this case $\mathcal{I}$ is of the form

$$\frac{\mathcal{I}_1 :: (\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e_1 \;:\; \tau' \to \tau \qquad \mathcal{I}_2 :: (\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e_2 \;:\; \tau'}{(\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e_1\; e_2 \;:\; \tau}\,.$$

Applying the induction hypothesis to $\mathcal{I}_1$ and $\mathcal{I}_2$ we get $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; e_1\,[u/l_0^0] \;:\; \tau' \to \tau$ and $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; e_2\,[u/l_0^0] \;:\; \tau'$ respectively. Now applying TP^{ex^l}-APP we get $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; (e_1\,[u/l_0^0])\; (e_2\,[u/l_0^0]) \;:\; \tau$. As $(e_1\,[u/l_0^0])\; (e_2\,[u/l_0^0]) = (e_1\; e_2)\,[u/l_0^0]$ we are done.

5. TP^{ex^l}-ABS. In this case $\mathcal{I}$ is of the form

$$\frac{\mathcal{I}' :: (\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid (\Gamma,\, x : \tau_1) \;\vdash^{\text{ex}^l}\; e' \;:\; \tau_2}{(\Omega;\, (\tau_0,\, \Sigma)) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; \lambda x : e' \,.\; \;:\; \tau_1 \to \tau_2}\,.$$

Applying the induction hypothesis to $\mathcal{I}'$ we get $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid (\Gamma,\, x : \tau_1) \;\vdash^{\text{ex}^l}\; e'\,[u/l_0^0] \;:\; \tau_2$. Now applying TP^{ex^l}-ABS we get $(\Omega;\, \Sigma) \mid (\Delta,\, u :: \tau_0) \mid \Gamma \;\vdash^{\text{ex}^l}\; \lambda x : e'\,[u/l_0^0] \,.\; (\;) :\; \tau_1 \to \tau_2$. As $\lambda x : e'\,[u/l_0^0] \,.\; (=)(\lambda x : e' \,.\;)\,[u/l_0^0]$ we are done.

6. TP^{ex^l}-BOX. This case goes by applying induction hypothesis just as in case 5.

7. TP^{ex^l}-LET-BOX. This case goes by applying induction hypothesis twice just as in case 4. □

Lemma 4.17. If $(\Omega;\, \cdot) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e \;:\; \tau$ then $\Omega \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l} \downarrow e \;:\; \tau$.

Proof. The proof can be done by straightforward induction on the derivation of the typing $(\Omega;\, \cdot) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e \;:\; \tau$. □

Lemma 4.18 (S-substitution)**.** If $(\Omega;\, \Sigma) \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e \;:\; \tau$ and $\Omega \mid \Gamma \;\vdash^{\text{S}}\; S \;:\; \Sigma$ then $\Omega \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e\,[S] \;:\; \tau$.

Proof. The proof is done by induction on the structure of S.

1. $S = \cdot$. In this case $\Sigma = \cdot$ and thus we have that $(\Omega; \cdot) \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e : \tau$. From **Lemma 4.17** we then get that $\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} \downarrow e : \tau$ and as $e\,[\cdot] = \downarrow e$ we are done.

2. $S = (e_0, S_0)$: Then the derivation of $\Omega \mid \Gamma \vdash^{\mathrm{S}} S : \Sigma$ is of the form

$$\frac{\mathcal{L} :: \Omega \mid \Gamma \vdash^{\mathrm{S}} S_0 : \Sigma_0 \qquad \mathcal{I} :: \Omega \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_0 : \Box\,\tau_0}{\Omega \mid \Gamma \vdash^{\mathrm{S}} (e_0, S_0) : (\tau_0, \Sigma_0)}$$

and we have that $(\Omega; (\tau_0, \Sigma_0)) \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l} e : \tau$. Now, from **Lemma 4.16** we get that $(\Omega; \Sigma_0) \mid (\Delta, u :: \tau_0) \mid \Gamma \vdash^{\mathrm{ex}^l} e\,[u/l_0^0] : \tau$ and applying the induction hypothesis to this and $\Omega \mid \Gamma \vdash^{\mathrm{S}} S_0 : \Sigma_0$ we get that $\Omega \mid (\Delta, u :: \tau_0) \mid \Gamma \vdash^{\mathrm{ex}^l} (e\,[u/l_0^0])\,[S_0] : \tau$. Also having that $\Omega \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_0 : \Box\,\tau_0$ we can now apply the rule TP$^{\mathrm{ex}}$-LET-BOX and get $\Omega \mid \Delta \mid \Gamma \vdash^{\mathrm{ex}^l}$ **let box** $u = e_0$ **in** $((e\,[u/l_0^0])\,[S_0]) : \tau$. As **let box** $u = e_0$ **in** $((e\,[u/l_0^0])\,[S_0])$ is equal to $e\,[(e_0, S_0)]$ we are done. □

Lemma 4.19. If $\Omega \mid \Gamma \vdash^{\mathrm{S}} S : \Sigma$ and $\Omega \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e : \Box\,\tau$ then $\Omega \mid \Gamma \vdash^{\mathrm{S}} S' : \Sigma'$ when $S + e \overset{\mathrm{end}}{\rightsquigarrow} S'$ and $\Sigma + \tau \overset{\mathrm{end}}{\rightsquigarrow} \Sigma'$

Proof. The proof is done by induction on the structure of S.

1. $S = \cdot$. In this case we must have that $S' = (e, \cdot)$ according to END$^{\mathrm{S}}$-0. As $\Sigma = \cdot$ according to TP$^{\mathrm{S}}$-0 we similarly have that $\Sigma' = (\tau, \cdot)$. Application of TP$^{\mathrm{S}}$-N finishes the case.

2. $S = (e_0, S_0)$. In this case we can, according to TP$^{\mathrm{S}}$-N, find τ_0 and Σ_0 such that $\Sigma = (\tau_0, \Sigma_0)$ and such that $\Omega \mid \Gamma \vdash^{\mathrm{S}} S_0 : \Sigma_0$ and $\Omega \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_0 : \Box\,\tau_0$. Also, according to END$^{\mathrm{S}}$-N we can find S_0' such that $S_0 + e \overset{\mathrm{end}}{\rightsquigarrow} S_0'$ and $S' = (e_0, S_0')$ and according to END$^{\Sigma}$-N we can find Σ_0' such that $\Sigma_0 + \tau \overset{\mathrm{end}}{\rightsquigarrow} \Sigma_0'$ and $\Sigma' = (\tau_0, \Sigma_0')$. Now applying the induction hypothesis to S_0 we get that $\Omega \mid \Gamma \vdash^{\mathrm{S}} S_0' : \Sigma_0'$ and an application of TP$^{\mathrm{S}}$-N to this and $\Omega \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_0 : \Box\,\tau_0$ finishes the case. □

Lemma 4.20. If $\Omega \mid \Gamma \vdash^{\mathrm{S}} S : \Sigma$ and $\Sigma + \tau \overset{\mathrm{end}}{\rightsquigarrow} \Sigma'$ then $\Sigma' \vdash^{l^{\Sigma}} |S| : \tau$.

Proof. The proof can be done by straightforward induction on the structure of S. □

Lemma 4.21 (Type preservation 1)**.** If

$$(\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p : \Box\,\tau$$

and

$$(\Psi;\ \Gamma) \vdash^{\mathrm{Z}} (Z;\ S) : (\Omega;\ \Sigma)$$

then m, n, Z', S', Ω', and Σ' can be found such that

$$p\,/\,(Z;\ S) \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} l_n^m\,/\,(Z';\ S'),$$

$$(\Psi;\ \Gamma)\ \vdash^{\mathrm{Z}}\ (Z';\ S')\ :\ (\Omega';\ \Sigma'),$$
$$(\Omega;\ \Sigma)\ \leq^{\Omega}\ (\Omega';\ \Sigma'),$$

and

$$(\Omega';\ \Sigma')\ \mid\ \cdot\ \mid\ \Gamma\ \vdash^{\mathrm{ex}^l}\ l^m_n\ :\ \tau.$$

Proof. The proof is done by induction on the derivation $\mathcal{N}$ of the typing $(\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\,\tau$. Below each typing rule is in turn considered the last rule applied in the derivation.

1. TP$^{\mathrm{im,p}}$-0. In this case $\mathcal{N}$ is of the form

$$\frac{\mathcal{N}'\ ::\ (\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \Box\,\tau}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im,p}}\ m\ :\ \Box\,\tau}.$$

When $(\Psi;\ \Gamma)\ \vdash^{\mathrm{Z}}\ (Z;\ S)\ :\ (\Omega;\ \Sigma)$ we know from TP$^{\mathrm{Z}}$-M that $\Psi\ \vdash^{\mathrm{Z}}\ Z\ :\ \Omega$ and thus we can apply **Lemma 4.22** to this and $(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \Box\,\tau$ and get e, Z', and Ω' such that

$$m\,/\,Z\ \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}}\ e\,/\,Z', \tag{4.2}$$

$$\Psi\ \vdash^{\mathrm{Z}}\ Z'\ :\ \Omega', \tag{4.3}$$

$$\Omega\ \leq^{\Omega}\ \Omega', \tag{4.4}$$

and

$$\Omega'\ \mid\ \cdot\ \mid\ \cdot\ \vdash^{\mathrm{ex}^l}\ e\ :\ \Box\,\tau. \tag{4.5}$$

From TP$^{\mathrm{Z}}$-M we also know that $\Omega\ \mid\ \Gamma\ \vdash^{\mathrm{S}}\ S\ :\ \Sigma$ and having (4.4) we can apply **Lemma 4.15** and get $\Omega'\ \mid\ \Gamma\ \vdash^{\mathrm{S}}\ S\ :\ \Sigma$. Now assuming that $S + e\ \overset{\mathrm{end}}{\rightsquigarrow}\ S'$ and $\Sigma + \tau\ \overset{\mathrm{end}}{\rightsquigarrow}\ \Sigma'$ we can derive from **Lemma 4.19** that $\Omega'\ \mid\ \Gamma\ \vdash^{\mathrm{S}}\ S'\ :\ \Sigma'$ and applying TP$^{\mathrm{Z}}$-M to this and (4.3) we get that

$$(\Psi;\ \Gamma)\ \vdash^{\mathrm{Z}}\ (Z';\ S')\ :\ (\Omega';\ \Sigma').$$

Having (4.2) and $S + e\ \overset{\mathrm{end}}{\rightsquigarrow}\ S'$ we can apply TR$^{\mathrm{im,ex}}$-POP-0 and get

$$m\,/\,(Z;\ S)\ \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}}\ l^0_{|S|}\,/\,(Z';\ S')$$

and as $\Sigma'\ \vdash^{l^\Sigma}\ |S|\ :\ \tau$ according to **Lemma 4.20** we can get from TP$^{l^\Omega}$-0 that

$$(\Omega';\ \Sigma')\ \mid\ \cdot\ \mid\ \cdot\ \vdash^{\mathrm{ex}^l}\ l^0_{|S|}\ :\ \tau.$$

Finally, as $\Sigma\ \leq^{\Sigma}\ \Sigma'$ according to **Lemma 4.11** we can apply PREF$^{\Omega}$-M to this and (4.4) to get that

$$(\Omega;\ \Sigma)\ \leq^{\Omega}\ (\Omega';\ \Sigma').$$

2. TP$^{\mathrm{im,p}}$-M. In this case $\mathcal{N}$ is of the form

$$\frac{\mathcal{N}'\ ::\ (\Psi_0;\ \Gamma_0)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\,\tau}{((\Psi_0;\ \Gamma_0);\ \Gamma)\ \vdash^{\mathrm{im,p}}\ \mathbf{pop}\ p\ :\ \Box\,\tau}$$

and according to TP$^{\mathrm{Z}}$-M, Z and Ω is of the form $(Z_0;\ S_0)$ and $(\Omega_0;\ \Sigma_0)$ respectively and $(\Psi_0;\ \Gamma_0)\ \vdash^{\mathrm{Z}}\ (Z_0;\ S_0)\ :\ (\Omega_0;\ \Sigma_0)$ and $(\Omega_0;\ \Sigma_0)\ |\ \Gamma\ \vdash^{\mathrm{S}}\ S\ :\ \Sigma$. Applying the induction hypothesis to $(\Psi_0;\ \Gamma_0)\ \vdash^{\mathrm{im,p}}\ p\ :\ \Box\,\tau$ and $(\Psi_0;\ \Gamma_0)\ \vdash^{\mathrm{Z}}\ (Z_0;\ S_0)\ :\ (\Omega_0;\ \Sigma_0)$ we get m, n, Z_0', S_0', Ω_0', and Σ_0' such that

$$p\,/\,(Z_0;\ S_0)\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ l_n^m\,/\,(Z_0';\ S_0'), \tag{4.6}$$

$$(\Psi_0;\ \Gamma_0)\ \vdash^{\mathrm{Z}}\ (Z_0';\ S_0')\ :\ (\Omega_0';\ \Sigma_0'), \tag{4.7}$$

$$(\Omega_0;\ \Sigma_0)\ \leq^{\Omega}\ (\Omega_0';\ \Sigma_0'), \tag{4.8}$$

and

$$(\Omega_0';\ \Sigma_0')\ |\ \cdot\ |\ \cdot\ \vdash^{\mathrm{ex}^l}\ l_n^m\ :\ \tau. \tag{4.9}$$

Applying TR$^{\mathrm{im,ex}}$-POP-M to (4.6) we get that

$$\mathbf{pop}\ p\,/\,((Z_0;\ S_0);\ S)\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ l_n^{m+1}\,/\,((Z_0';\ S_0');\ S)$$

and

$$((\Omega_0';\ \Sigma_0');\ \Sigma)\ |\ \cdot\ |\ \cdot\ \vdash^{\mathrm{ex}^l}\ l_n^{m+1}\ :\ \tau$$

follows from inverse application of TP$^{\mathrm{ex}^l}$-VAR-L to (4.9) and then forward application of TP$^{l^{\Omega}}$-M and TP$^{\mathrm{ex}^l}$-VAR-L. Now, having (4.8) and $(\Omega_0;\ \Sigma_0)\ |\ \Gamma\ \vdash^{\mathrm{S}}\ S\ :\ \Sigma$ we can apply **Lemma 4.15** and get $(\Omega_0';\ \Sigma_0')\ |\ \Gamma\ \vdash^{\mathrm{S}}\ S\ :\ \Sigma$ and applying TP$^{\mathrm{Z}}$-M to this and (4.7) we get that

$$((\Psi_0;\ \Gamma_0);\ \Gamma)\ \vdash^{\mathrm{Z}}\ ((Z_0';\ S_0');\ S)\ :\ ((\Omega_0';\ \Sigma_0');\ \Sigma).$$

Finally,

$$((\Omega_0;\ \Sigma_0);\ \Sigma)\ \leq^{\Omega}\ ((\Omega_0';\ \Sigma_0');\ \Sigma)$$

follows from application of PREF$^{\Omega}$-M to (4.8) and $\Sigma\ \leq^{\Sigma}\ \Sigma$ (**Lemma 4.9**). □

Lemma 4.22 (Type preservation 2)**.** If

$$(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \tau$$

and

$$\Psi\ \vdash^{\mathrm{Z}}\ Z\ :\ \Omega$$

then e, Z', and Ω' can be found such that

$$m\,/\,Z\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ e\,/\,Z',$$

$$\Psi\ \vdash^{\mathrm{Z}}\ Z'\ :\ \Omega',$$

$$\Omega\ \leq^{\Omega}\ \Omega',$$

and

$$\Omega'\ |\ \cdot\ |\ \Gamma\ \vdash^{\mathrm{ex}^l}\ e\ :\ \tau.$$

Proof. The proof is done by induction on the derivation $\mathcal{M}$ of the typing $(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m\ :\ \tau$. Below each typing rule is in turn considered the last rule applied in the derivation.

1. TP$^{\mathrm{im}}$-VAR. In this case $\mathcal{M}$ is of the form

$$\frac{x : \tau \in \Gamma}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ x\ :\ \tau}\,.$$

As we know from TR$^{\mathrm{im,ex}}$-VAR that $x\ /\ Z\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ x\ /\ Z$, know from **Lemma 4.10** that $\Omega\ \leq^{\Omega}\ \Omega$, and know from TP$^{\mathrm{ex}^l}$-VAR-X that $\Omega\ \mid\ \cdot\ \mid\ \Gamma\ \vdash^{\mathrm{ex}^l}\ x\ :\ \tau$ when $x : \tau \in \Gamma$ choosing $e = x$, $Z' = Z$, and $\Omega' = \Omega$ proves the case.

2. TP$^{\mathrm{im}}$-APP. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{M}_1 :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m_1\ :\ \tau' \to \tau \qquad \mathcal{M}_1 :: (\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m_2\ :\ \tau'}{(\Psi;\ \Gamma)\ \vdash^{\mathrm{im}}\ m_1\ m_2\ :\ \tau}\,.$$

Applying the induction hypothesis to $\mathcal{M}_1$ and $\Psi\ \vdash^{\mathrm{Z}}\ Z\ :\ \Omega$ we get e_1, Z_1, and Ω_1 such that

$$m_1\ /\ Z\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ e_1\ /\ Z_1, \tag{4.10}$$

$$\Psi\ \vdash^{\mathrm{Z}}\ Z_1\ :\ \Omega_1, \tag{4.11}$$

$$\Omega\ \leq^{\Omega}\ \Omega_1, \tag{4.12}$$

and

$$\Omega_1\ \mid\ \cdot\ \mid\ \Gamma\ \vdash^{\mathrm{ex}^l}\ e_1\ :\ \tau' \to \tau. \tag{4.13}$$

Applying the induction hypothesis to $\mathcal{M}_2$ and (4.11) we get e_2, Z_2, and Ω_2 such that

$$m_2\ /\ Z_1\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ e_2\ /\ Z_2, \tag{4.14}$$

$$\Psi\ \vdash^{\mathrm{Z}}\ Z_2\ :\ \Omega_2, \tag{4.15}$$

$$\Omega_1\ \leq^{\Omega}\ \Omega_2, \tag{4.16}$$

and

$$\Omega_2\ \mid\ \cdot\ \mid\ \Gamma\ \vdash^{\mathrm{ex}^l}\ e_2\ :\ \tau'. \tag{4.17}$$

Having (4.10) and (4.14) we can apply TR$^{\mathrm{im,ex}}$-APP to get

$$m_1\ m_2\ /\ Z\ \overset{\mathrm{im}}{\leadsto}^{\mathrm{ex}}\ e_1\ e_2\ /\ Z_2.$$

From (4.15) we have

$$\Psi\ \vdash^{\mathrm{Z}}\ Z_2\ :\ \Omega_2,$$

and applying the transitivity **Lemma 4.8** to (4.12) and (4.16) we get

$$\Omega\ \leq^{\Omega}\ \Omega_2.$$

Now applying **Lemma 4.14** to $\Omega_1 \leq^{\Omega} \Omega_2$ and (4.13) we get $\Omega_2 \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_1 : \tau' \to \tau$. Also having (4.17) we can apply $\textsc{tp}^{\mathrm{ex}^l}$-APP to get

$$\Omega_2 \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} e_1\ e_2 : \tau.$$

Thus choosing $e = e_1\ e_2$, $Z' = Z_2$ and $\Omega' = \Omega_2$ proves the case.

3. $\textsc{tp}^{\mathrm{im}}$-ABS. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{M}' :: (\Psi;\ (\Gamma,\ x : \tau_1)) \vdash^{\mathrm{im}} m' : \tau_2}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \lambda x : m' .\ : \tau_1 \to \tau_2}.$$

Applying the induction hypothesis to $\mathcal{M}'$ and $\Psi \vdash^{\mathrm{Z}} Z : \Omega$ we get e', Z', and Ω' such that

$$m' \,/\, Z \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} e' \,/\, Z', \tag{4.18}$$

$$\Psi \vdash^{\mathrm{Z}} Z' : \Omega', \tag{4.19}$$

$$\Omega \leq^{\Omega} \Omega', \tag{4.20}$$

and

$$\Omega' \mid \cdot \mid (\Gamma,\ x : \tau_1) \vdash^{\mathrm{ex}^l} e' : \tau_2. \tag{4.21}$$

Having (4.18) we can apply $\textsc{tr}^{\mathrm{im,ex}}$-ABS to get

$$\lambda x : m' .\ / Z \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} \lambda x : e' .\ / Z'$$

and by applying $\textsc{tp}^{\mathrm{ex}^l}$-ABS to (4.21) we get

$$\Omega' \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} \lambda x : e' .\ : \tau_1 \to \tau_2.$$

This proves the case.

4. $\textsc{tp}^{\mathrm{im}}$-BOX. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{M}' :: ((\Psi;\ \Gamma);\ \cdot) \vdash^{\mathrm{im}} m' : \tau'}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{box}\ m' : \Box\,\tau'}.$$

From $\textsc{tp}^{\mathrm{S}}$-0 we have that $\Omega \mid \Gamma \vdash^{\mathrm{S}} \cdot : \cdot$ and also having $\Psi \vdash^{\mathrm{Z}} Z : \Omega$ we can apply $\textsc{tp}^{\mathrm{Z}}$-M to get $(\Psi;\ \Gamma) \vdash^{\mathrm{Z}} (Z;\ \cdot) : (\Omega;\ \cdot)$. Now applying the induction hypothesis to $\mathcal{M}'$ and $(\Psi;\ \Gamma) \vdash^{\mathrm{Z}} (Z;\ \cdot) : (\Omega;\ \cdot)$ we get e', Z'', and Ω'' such that

$$m' \,/\, (Z;\ \cdot) \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} e' \,/\, Z'', \tag{4.22}$$

$$(\Psi;\ \Gamma) \vdash^{\mathrm{Z}} Z'' : \Omega'', \tag{4.23}$$

$$(\Omega;\ \cdot) \leq^{\Omega} \Omega'', \tag{4.24}$$

and

$$\Omega'' \mid \cdot \mid \cdot \vdash^{\mathrm{ex}^l} e' : \tau'. \tag{4.25}$$

From PREF$^{\Omega}$-M we know that for (4.24) to be true we must have Ω_0'' and Σ_0'' such that $\Omega'' = (\Omega_0''; \Sigma_0'')$ and

$$\Omega \leq^{\Omega} \Omega_0''.$$

But then, having $\Omega'' = (\Omega_0''; \Sigma_0'')$, the rule TP$^{\mathrm{Z}}$-M must be the last rule applied in the derivation of (4.23) and thus Z'' must be of the form $(Z_0''; S_0'')$ for some Z_0'' and S_0'' such that $\Omega_0'' \mid \Gamma \vdash^{\mathrm{S}} S_0'' : \Sigma_0''$ and

$$\Psi \vdash^{\mathrm{Z}} Z_0'' : \Omega_0''.$$

Applying TP$^{\mathrm{ex}^l}$-BOX to (4.25) we get $\Omega'' \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} \mathbf{box}\ e' : \Box\tau'$. Also having $\Omega_0'' \mid \Gamma \vdash^{\mathrm{S}} S_0'' : \Sigma_0''$ we can apply **Lemma 4.18** to get

$$\Omega_0'' \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} (\mathbf{box}\ e')\,[S_0''] : \Box\tau'.$$

By applying TR$^{\mathrm{im,ex}}$-BOX to (4.22) we get

$$\mathbf{box}\ m' \,/\, Z \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} (\mathbf{box}\ e')\,[S_0''] \,/\, Z_0''$$

and it has now been shown that $e = (\mathbf{box}\ e')\,[S_0'']$, $Z' = Z_0''$ and $\Omega' = \Omega_0''$ are appropriate choices for this case.

5. TP$^{\mathrm{im}}$-UNBOX. In this case $\mathcal{M}$ is of the form

$$\frac{\mathcal{N}' :: (\Psi;\ \Gamma) \vdash^{\mathrm{im,p}} p : \Box\tau}{(\Psi;\ \Gamma) \vdash^{\mathrm{im}} \mathbf{unbox}\ p : \tau}.$$

First assuming that p is of the form $\mathbf{pop}\ p'$ the rule TP$^{\mathrm{im,p}}$-M must be the last one applied in $\mathcal{M}'$ and thus Ψ must be of the form $(\Psi_0;\ \Gamma_0)$ and $(\Psi_0;\ \Gamma_0) \vdash^{\mathrm{im,p}} p' : \Box\tau$. Then, according to TP$^{\mathrm{Z}}$-M, Z must be of the form $(Z_0; S)$ and Ω must be of the form $(\Omega_0; \Sigma)$. Applying **Lemma 4.21** to $(\Psi_0;\ \Gamma_0) \vdash^{\mathrm{im,p}} p' : \Box\tau$ and $(\Psi_0;\ \Gamma_0) \vdash^{\mathrm{Z}} (Z_0; S) : (\Omega_0; \Sigma)$ we get m, n, Z_0', S', Ω_0', and Σ' such that

$$p' \,/\, (Z_0;\ S) \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} l_n^m \,/\, (Z_0';\ S'), \tag{4.26}$$

$$(\Psi_0;\ \Gamma_0) \vdash^{\mathrm{Z}} (Z_0';\ S') : (\Omega_0';\ \Sigma'), \tag{4.27}$$

$$(\Omega_0;\ \Sigma) \leq^{\Omega} (\Omega_0';\ \Sigma'), \tag{4.28}$$

and

$$(\Omega_0';\ \Sigma') \mid \cdot \mid \cdot \vdash^{\mathrm{ex}^l} l_n^m : \tau. \tag{4.29}$$

Having (4.26) we can apply TR$^{\mathrm{im,ex}}$-UNBOX-M to get

$$\mathbf{unbox}\ (\mathbf{pop}\ p') \,/\, (Z_0;\ S) \overset{\mathrm{im}}{\rightsquigarrow}^{\mathrm{ex}} l_n^m \,/\, (Z_0';\ S')$$

and

$$(\Omega_0';\ \Sigma') \mid \cdot \mid \Gamma \vdash^{\mathrm{ex}^l} \tau :$$

follows by first inverse application of TP$^{\text{ex}^l}$-VAR-L to (4.29) and then forward application. Thus $e = l_n^m$, $Z' = (Z'_0; S')$ and $\Omega' = (\Omega'_0; \Sigma')$ are appropriate choices when $p = \mathbf{pop}\ p'$. If we instead assume that p is of the form m' the rule TP$^{\text{im, P}}$-0 must be the last one applied in $\mathcal{M}'$ and we must have that $(\Psi;\ \Gamma) \vdash^{\text{im}} m' : \Box\,\tau$. Applying the induction hypothesis to this and $\Psi \vdash^{\text{Z}} Z : \Omega$ we get e', Z', and Ω' such that

$$m' \,/\, Z \overset{\text{im}}{\leadsto}^{\text{ex}} e' \,/\, Z' \tag{4.30}$$

$$\Psi \vdash^{\text{Z}} Z' : \Omega' \tag{4.31}$$

$$\Omega \leq^{\Omega} \Omega' \tag{4.32}$$

and

$$\Omega' \mid \cdot \mid \Gamma \vdash^{\text{ex}^l} e' : \Box\,\tau. \tag{4.33}$$

Having (4.30) we can apply TR$^{\text{im,ex}}$-UNBOX-0 to get

$$m' \,/\, Z \overset{\text{im}}{\leadsto}^{\text{ex}} \mathbf{let\ box}\ u = e'\ \mathbf{in}\ u \,/\, Z'$$

and

$$\Omega' \mid \cdot \mid \Gamma \vdash^{\text{ex}^l} \mathbf{let\ box}\ u = e'\ \mathbf{in}\ u : \tau$$

follows from application of TP$^{\text{ex}^l}$-LET-BOX to (4.33) and $\Omega' \mid u :: \tau \mid \Gamma \vdash^{\text{ex}^l} u : \tau$ (see TP$^{\text{ex}^l}$-VAR-U). This showed that $e = \mathbf{let\ box}\ u = e'\ \mathbf{in}\ u$ is an appropriate choice when $p = m'$ and it finishes the case. □

Lemma 4.23 (Bridge between ex^l- and ex-typing)**.** If $\cdot \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$ then $\Delta \mid \Gamma \vdash^{\text{ex}} e : \tau$.

Proof. The proof can be done by straightforward induction on the derivation of the typing $\cdot \mid \Delta \mid \Gamma \vdash^{\text{ex}^l} e : \tau$. □

Finally we have the following main theorem:

Theorem 4.8 (Main type preservation)**.** If

$$(\cdot;\ \cdot) \vdash^{\text{im}} m : \tau$$

then e can be found such that

$$m \,/\, \cdot \overset{\text{im}}{\leadsto}^{\text{ex}} e \,/\, \cdot$$

and

$$\cdot \mid \cdot \vdash^{\text{ex}} e : \tau.$$

Proof. According to **Lemma 4.22** we can find e, Z', and Ω' such that

$$m \,/\, \cdot \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, Z',$$

$$\Psi \;\vdash^{\text{Z}}\; Z' \;:\; \Omega',$$

$$\cdot \;\leq^{\Omega}\; \Omega',$$

and

$$\Omega' \mid \cdot \mid \cdot \;\vdash^{\text{ex}^l}\; e \;:\; \tau.$$

As PREF^{Σ}-0 must be the last rule applied in the derivation of $\cdot \;\leq^{\Omega}\; \Omega'$ we have $\Omega' = \cdot$. But then TP^{Z}-0 must be the last rule applied in the derivation of $\Psi \;\vdash^{\text{Z}}\; Z' \;:\; \Omega'$ and thus we also have $Z' = \cdot$. Substituting $\cdot$ for Z' we get

$$m \,/\, \cdot \;\overset{\text{im}}{\rightsquigarrow}^{\text{ex}}\; e \,/\, \cdot$$

and applying **Lemma 4.23** we get

$$\cdot \mid \cdot \;\vdash^{\text{ex}}\; e \;:\; \tau$$

which finishes the proof. □

4.3.3 LF representation of the translation

The Twelf code concerning the representation of the translation from Mini-ML$^{\Box}$ to Mini-ML$^{\Box}_{\text{ex}}$ can be found in Section **A.4**. We will not give all the representation details here but just give an overview of the main connections between the paper definitions above and the type families in the Twelf code:

Paper	Twelf
Σ	`ctx`
$\Sigma \;\leq^{\Sigma}\; \Sigma'$	`ctx_prefix`
Lemma 4.7	`ctx_prefix_trans`
Lemma 4.9	`ctx_prefix_refl`
Ω	`ktx`
$\Omega \;\leq^{\Omega}\; \Omega'$	`ktx_prefix`
Lemma 4.8	`ktx_prefix_trans`
Lemma 4.10	`ktx_prefix_refl`
$\Omega \mid \Gamma \;\vdash^{\text{S}}\; S \;:\; \Sigma$	`sequence`
$\Psi \;\vdash^{\text{Z}}\; Z \;:\; \Omega$	`zequence`
$\Sigma \;\vdash^{l^{\Sigma}}\; n \;:\; \tau$	`lab_c`
$\Omega \;\vdash^{l^{\Omega}}\; (m, n) \;:\; \tau$	`lab_k`
$\Omega \mid \Delta \mid \Gamma \;\vdash^{\text{ex}^l}\; e \;:\; \tau$	`eek`
Lemma 4.11+19+20	`sequence_end`

Paper	Twelf
Lemma 4.12	`weak_lab_c`
Lemma 4.13	`weak_lab_k`
Lemma 4.14	`weak_eek`
Lemma 4.15	`weak_sequence`
Lemma 4.16	`repl_lab`
Lemma 4.17	`drop_stage`
Lemma 4.18	`repl_sequence`
Lemma 4.21	`tr_i~>e_pop`
Lemma 4.22	`tr_i~>e`
Lemma 4.23	`rm_ktx`
Theorem 4.8	`tr_i~>e_final`

An important step in making the representation work was the incorporation of our knowledge of the structure of the underlying Kripke model: each world has at most one immediate predecessor and no world can be its own predecessor. Below we sketch the corresponding paper formalization of these properties.

We use $\mathcal{W}$ to range over world contexts defined by the judgement $\boxed{\mathrm{wctx}\,(\mathcal{W})}$:

$$\frac{}{\mathrm{wctx}\,(\cdot,\ \mathbf{w}_0)}\ \textsc{wld-ctx-0}$$

$$\frac{\mathrm{wctx}\,(\mathcal{W}) \qquad w\ \in\ \mathcal{W} \qquad w'\ \notin\ \mathcal{W}}{\mathrm{wctx}\,((\mathcal{W},\ w'),\ w\ \prec\ w')}\ \textsc{wld-ctx-n}$$

The world $\mathbf{w}_0$ can bee seen as the root of the world tree, and $w \prec w' \in \mathcal{W}$ means that it is possible to jump from world w to w'.

We define the four judgements $\boxed{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \prec\ w'}$, $\boxed{\mathcal{W}\ \vdash^{\mathrm{w}}\mathrm{norm}\,(w)}$, $\boxed{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w'}$, and $\boxed{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ <^{\mathrm{w}}\ w'}$ by the following inference rules:

$$\frac{w\ \prec\ w'\ \in\ \mathcal{W}}{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \prec\ w'}\ \textsc{wld-jump}$$

$$\frac{}{\mathcal{W}\ \vdash^{\mathrm{w}}\mathrm{norm}\,(\mathbf{w}_0)}\ \textsc{wld-norm-0}$$

$$\frac{\mathcal{W}\ \vdash^{\mathrm{w}}\mathrm{norm}\,(w') \quad \mathcal{W}\ \vdash^{\mathrm{w}}\ w'\ \prec\ w}{\mathcal{W}\ \vdash^{\mathrm{w}}\mathrm{norm}\,(w)}\ \textsc{wld-norm-n}$$

$$\frac{w\ \in\ \mathcal{W}}{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w}\ \textsc{wld-pred-eq-0}$$

$$\frac{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w' \quad \mathcal{W}\ \vdash^{\mathrm{w}}\ w'\ \prec\ w''}{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w''}\ \textsc{wld-pred-eq-n}$$

$$\frac{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w' \quad \mathcal{W}\ \vdash^{\mathrm{w}}\ w'\ \prec\ w''}{\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ <^{\mathrm{w}}\ w''}\ \textsc{wld-pred-n}$$

Lemma 4.24 (Front extension of predecessing)**.** If $\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \prec\ w'$ and $\mathcal{W}\ \vdash^{\mathrm{w}}\ w'\ \leq^{\mathrm{w}}\ w''$ then both $\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ \leq^{\mathrm{w}}\ w''$ and $\mathcal{W}\ \vdash^{\mathrm{w}}\ w\ <^{\mathrm{w}}\ w''$.

Proof. The lemma can be proved by straightforward induction on the structure of the derivation of $\mathcal{W}\ \vdash^{\mathrm{w}}\ w'\ \leq^{\mathrm{w}}\ w''$. □

Lemma 4.25 (Uniqueness of immediate predecessor)**.** If $\mathcal{W} \vdash^{\mathrm{w}} w' \prec w$ and $\mathcal{W} \vdash^{\mathrm{w}} w'' \prec w$ then $w' = w''$.

Proof. The lemma follows from inspection of the rule WLD-CTX-N: when a jump-assumption is added to the world context it is always a jump to a newly introduced world. □

Theorem 4.9 (Irreflexivity of predecessing)**.** If $\mathcal{W} \vdash^{\mathrm{w}} \mathrm{norm}\,(w)$ then $\mathcal{W} \nvdash^{\mathrm{w}} w <^{\mathrm{w}} w$.

Proof. The theorem is proved by induction on the structure of the derivation of $\mathcal{W} \vdash^{\mathrm{w}} \mathrm{norm}\,(w)$.

1. $w = \mathbf{w}_0$. If $\mathcal{W} \vdash^{\mathrm{w}} \mathbf{w}_0 <^{\mathrm{w}} \mathbf{w}_0$ we would according to WLD-PRED-N have a w' such that $\mathcal{W} \vdash^{\mathrm{w}} w' \prec \mathbf{w}_0$. But as $w' \prec \mathbf{w}_0 \in \mathcal{W}$ can never happen it must be the case that $\mathcal{W} \nvdash^{\mathrm{w}} \mathbf{w}_0 <^{\mathrm{w}} \mathbf{w}_0$.

2. $w \neq \mathbf{w}_0$. Assume that $\mathcal{W} \vdash^{\mathrm{w}} w <^{\mathrm{w}} w$. Then according to WLD-PRED-N we can find w' such that $\mathcal{W} \vdash^{\mathrm{w}} w \leq^{\mathrm{w}} w'$ and $\mathcal{W} \vdash^{\mathrm{w}} w' \prec w$. As $\mathcal{W} \vdash^{\mathrm{w}} \mathrm{norm}\,(w)$, we can also find w'' such that $\mathcal{W} \vdash^{\mathrm{w}} \mathrm{norm}\,(w'')$ and $\mathcal{W} \vdash^{\mathrm{w}} w'' \prec w$. Applying **Lemma 4.24** to $\mathcal{W} \vdash^{\mathrm{w}} w'' \prec w$ and $\mathcal{W} \vdash^{\mathrm{w}} w \leq^{\mathrm{w}} w'$ we get that $\mathcal{W} \vdash^{\mathrm{w}} w'' <^{\mathrm{w}} w'$ and as $w' = w''$ according to **Lemma 4.25**, we then have that $\mathcal{W} \vdash^{\mathrm{w}} w'' <^{\mathrm{w}} w''$. As $\mathcal{W} \nvdash^{\mathrm{w}} w'' <^{\mathrm{w}} w''$ according to the induction hypothesis, we have reached a contradiction, and thus it must be the case that $\mathcal{W} \nvdash^{\mathrm{w}} w <^{\mathrm{w}} w$. □

We have the following table:

Paper	Twelf
$\mathcal{W} \vdash^{\mathrm{w}} w \prec w'$	`world_jump`
$\mathcal{W} \vdash^{\mathrm{w}} \mathrm{norm}\,(w)$	`world_norm`
$\mathcal{W} \vdash^{\mathrm{w}} w \leq^{\mathrm{w}} w'$	`world_pred_eq`
$\mathcal{W} \vdash^{\mathrm{w}} w <^{\mathrm{w}} w'$	`world_pred`
Lemma 4.24	`world_pred_eq_front`
	`world_pred_front`
Lemma 4.25	`world_pred_unq`
Theorem 4.9	`world_pred_irrefl`

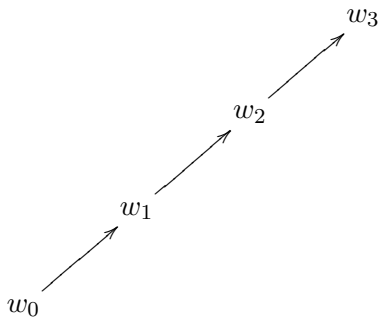
The fact that world predecessing is irreflexive is needed when representing the box-case of the proof of **Lemma 4.22**: if x occurs freely within m' then x have to be from som earlier world, and we know for sure that this earlier world will not be the current world itself. This is what the type family `elim_x` represents.

Chapter 5

Staged Computation and Temporal Logic

The second metaprogramming system to be explored in this project is the one presented by Davies in [4]. In this paper Davies extends the Curry-Howard isomorphism to a linear-time temporal logic and thereby obtains a metaprogramming language allowing for manipulation of code containing free variables. This is opposed to the metaprogramming languages presented in the previous chapter within which generated code tended to contain superfluous β-redexes. The system of the current chapter does not solely deliver improvements in comparison with the previous system. A significant drawback is the lost possibility of instant execution of generated code.

In the system of this chapter computation stages can be compared to the worlds of a Kripke model in which the worlds are placed on a linear time line and each world thus has at most one predecessor and at most one successor. The following directed graph illustrates an example of such a Kripke model:



The temporal operator $\bigcirc$ is for reasoning about truth in the next world. That is, $\bigcirc A$ is said to be true in for instance w_1 iff A can be proved to be true in the world w_2. On extension of the Curry-Howard isomorphism to include $\bigcirc$ the operator is imaged to a type constructor used for classification of expressions representing code of the next computation stage. It is the circumstance that this next computation stage will always be uniquely given which allows for the generation of code containing free variables. In the system of the previous chapter generated code had to be valid in *any* computation stage that might follow whereas

in the current setting code only has to be valid in *the* following computation. Therefore any assumption that might have been added to the next computation stage at an earlier point can be referred to from within the code.

The style of programming that can be done within the temporal metaprogramming language Mini-ML$^{\bigcirc}$ presented by Davies in [4] is much similar to how programming can be done within the implicit language Mini-ML$^{\Box}$. Code of a previous computation stage can be established when needed and does not have to be prepared beforehand as in the explicit language Mini-ML$^{\Box}_{\text{ex}}$.

Below we will present Mini-ML$^{\Box}$ in detail and find an adequate representation within the logical framework LF. Some soundness properties will also be represented within LF and these will be machine verified by use of the metalogical framework Twelf.

5.1 Syntax of Mini-ML$^{\bigcirc}$

The syntax categories of Mini-ML$^{\bigcirc}$ are the following:

$$
\begin{array}{rrcl}
\text{Types:} & \tau & ::= & \mathbf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \bigcirc\, \tau \\
\text{Expressions:} & e & ::= & x \mid \lambda x : \tau\,.\, e \mid e_1\ e_2 \mid \mathbf{fix}\ x : \tau\,.\, e \mid \langle e_1, e_2 \rangle \mid \\
 & & & \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \mid \\
 & & & \mathbf{next}\ e \mid \mathbf{prev}\ e \\
\text{World numbers:} & w & ::= & 0 \mid w + 1 \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma,\ x : \tau^w
\end{array}
$$

Here we have the "next" type $\bigcirc$ which will be used for classification of expressions representing code of the next stage. These expressions are constructed using the next-constructor whereas the previous-constructor is used for transferring code from the previous stage. The assumptions in the context are labeled with a number indicating in which world the given assumption is valid.

Similar to previous practice, we will use $\boxed{\Gamma_1,\ \Gamma_2}$ to mean the concatenation of the contexts Γ_1 and Γ_2.

The function $\boxed{\Gamma\{x \mapsto \tau^w\}}$ updating a context with a new assumption is defined in the following way:

$$
\begin{array}{rcl}
\cdot\{x \mapsto \tau^w\} & = & \cdot,\ x : \tau^w \\
(\Gamma,\ x : \tau_0^w)\{x \mapsto \tau^w\} & = & \Gamma,\ x : \tau^w \\
(\Gamma,\ x' : \tau_0^{w'})\{x \mapsto \tau^w\} & = & \Gamma\{x \mapsto \tau^w\},\ x' : \tau_0^{w'}, \text{ when } w \neq w'
\end{array}
$$

As this definition indicates, a variable is allowed to occur more than once in a given context as long as the world numbers attached are different. In the following we presume that contexts fulfill this validity constraint unless otherwise explicitly mentioned.

5.2 Typing rules for Mini-ML$^{\bigcirc}$

To know which of the assumptions within the context are allowed to be used at a given point in typing, the typing judgement has a world number attached. The typing judgement has the form $\boxed{\Gamma \vdash^{\text{te}}_{w} e : \tau}$ and with the assumption overwriting methodology incorporated it is defined by the following inference rules:

$$\frac{\Gamma(x) = \tau^{w}}{\Gamma \vdash^{\text{te}}_{w} x : \tau} \ \text{TP}^{\text{te}}\text{-VAR}$$

$$\frac{\Gamma\{x \mapsto \tau_1^{w}\} \vdash^{\text{te}}_{w} e : \tau_2}{\Gamma \vdash^{\text{te}}_{w} \lambda x : \tau_1 \,.\, e : \tau_1 \to \tau_2} \ \text{TP}^{\text{te}}\text{-ABS}$$

$$\frac{\Gamma \vdash^{\text{te}}_{w} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash^{\text{te}}_{w} e_2 : \tau_1}{\Gamma \vdash^{\text{te}}_{w} e_1\ e_2 : \tau_2} \ \text{TP}^{\text{te}}\text{-APP}$$

$$\frac{\Gamma\{x \mapsto \tau^{w}\} \vdash^{\text{te}}_{w} e : \tau}{\Gamma \vdash^{\text{te}}_{w} \textbf{fix}\ x : \tau \,.\, e : \tau} \ \text{TP}^{\text{te}}\text{-FIX}$$

$$\frac{\Gamma \vdash^{\text{te}}_{w} e_1 : \tau_1 \qquad \Gamma \vdash^{\text{te}}_{w} e_2 : \tau_2}{\Gamma \vdash^{\text{te}}_{w} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \ \text{TP}^{\text{te}}\text{-PAIR}$$

$$\frac{\Gamma \vdash^{\text{te}}_{w} e : \tau_1 \times \tau_2}{\Gamma \vdash^{\text{te}}_{w} \textbf{fst}\ e : \tau_1} \ \text{TP}^{\text{te}}\text{-FST} \qquad \frac{\Gamma \vdash^{\text{te}}_{w} e : \tau_1 \times \tau_2}{\Gamma \vdash^{\text{te}}_{w} \textbf{snd}\ e : \tau_2} \ \text{TP}^{\text{te}}\text{-SND}$$

$$\frac{}{\Gamma \vdash^{\text{te}}_{w} \textbf{z} : \textbf{nat}} \ \text{TP}^{\text{te}}\text{-ZERO} \qquad \frac{\Gamma \vdash^{\text{te}}_{w} e : \textbf{nat}}{\Gamma \vdash^{\text{te}}_{w} \textbf{s}\ e : \textbf{nat}} \ \text{TP}^{\text{te}}\text{-SUCC}$$

$$\frac{\Gamma \vdash^{\text{te}}_{w} e : \textbf{nat} \qquad \Gamma \vdash^{\text{te}}_{w} e_1 : \tau \qquad \Gamma\{x \mapsto \textbf{nat}^{w}\} \vdash^{\text{te}}_{w} e_2 : \tau}{\Gamma \vdash^{\text{te}}_{w} \textbf{case}\ e\ \textbf{of}\ \textbf{z} \Rightarrow e_1 \mid \textbf{s}\ x \Rightarrow e_2 : \tau} \ \text{TP}^{\text{te}}\text{-CASE}$$

$$\frac{\Gamma \vdash^{\text{te}}_{w+1} e : \tau}{\Gamma \vdash^{\text{te}}_{w} \textbf{next}\ e : \bigcirc\tau} \ \text{TP}^{\text{te}}\text{-NEXT} \qquad \frac{\Gamma \vdash^{\text{te}}_{w} e : \bigcirc\tau}{\Gamma \vdash^{\text{te}}_{w+1} \textbf{prev}\ e : \tau} \ \text{TP}^{\text{te}}\text{-PREV}$$

where $\boxed{\Gamma(x) = \tau^{w}}$ means that $x : \tau^{w}$ is present within Γ.

As one would expect, the world number only changes in the two rules TP$^{\text{te}}$-NEXT and TP$^{\text{te}}$-PREV indicating a forward and backward jump respectively.

With the typing rules above the expression

$$\textbf{next}\ (\lambda x : \tau\,.\ (\textbf{prev}\ (\textbf{next}\ x)))$$

is obviously a well-typed temporal expression and it perfectly illustrates how the linear ordering of the worlds makes variable assumptions survive across world jumps. The corresponding expression in Mini-ML$^{\Box}$ would look like

$$\textbf{box}\ (\lambda x : \tau\,.\ (\textbf{unbox}\ (\textbf{pop}\ (\textbf{box}\ x))))$$

and would not be valid according to the typing rules defined there. The expression within a box-constructor has to be valid in any of the worlds reachable from the current world and thus only code variables are allowed to occur freely.

5.3 Representing well-typed Mini-ML$^{\bigcirc}$ expressions in LF

Below we shall see how an adequate representation of Mini-ML$^{\bigcirc}$ can be achieved by a straightforward adaption of the adequate representation developed for the implicit language in the previous chapter.

As opposed to what was the case in the previous chapter, the world topology prevailing in the current chapter can simply be represented as a sequence of numbers. These world numbers can be recognized as wumbers in the following.

The LF signature Σ_{te} to be used in representation of Mini-ML$^{\bigcirc}$ has the following contents:

$$\begin{aligned}
\textbf{wumber} &: \textbf{type}\\
\textbf{wumber_0} &: \textbf{wumber}\\
\textbf{wumber_n} &: \textbf{wumber} \rightarrow \textbf{wumber}
\end{aligned}$$

$$\begin{aligned}
\textbf{tp} &: \textbf{type}\\
\textbf{nat} &: \textbf{tp}\\
\textbf{arrow} &: \textbf{tp} \rightarrow \textbf{tp} \rightarrow \textbf{tp}\\
\textbf{product} &: \textbf{tp} \rightarrow \textbf{tp} \rightarrow \textbf{tp}\\
\textbf{circle} &: \textbf{tp} \rightarrow \textbf{tp}
\end{aligned}$$

$$\begin{aligned}
\textbf{exp} &: \textbf{wumber} \rightarrow \textbf{tp} \rightarrow \textbf{type}\\
\textbf{lam} &: \Pi W : \textbf{wumber}\,.\ \Pi T_1 : \textbf{tp}\,.\ \Pi T_2 : \textbf{tp}\,.\\
&\qquad (\textbf{exp}\ W\ T_1 \rightarrow \textbf{exp}\ W\ T_2) \rightarrow \textbf{exp}\ W\ (\textbf{arrow}\ T_1\ T_2)\\
\textbf{app} &: \Pi W : \textbf{wumber}\,.\ \Pi T_1 : \textbf{tp}\,.\ \Pi T_2 : \textbf{tp}\,.
\end{aligned}$$

$$
\begin{array}{rcl}
 & & \mathbf{exp}\ W\ (\mathbf{arrow}\ T_1\ T_2) \rightarrow \mathbf{exp}\ W\ T_1 \rightarrow \mathbf{exp}\ W\ T_2 \\
\mathbf{fix} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T : \mathbf{tp}\,.\,(\mathbf{exp}\ W\ T \rightarrow \mathbf{exp}\ W\ T) \rightarrow \mathbf{exp}\ W\ T \\
\mathbf{pair} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ W\ T_1 \rightarrow \mathbf{exp}\ W\ T_2 \rightarrow \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \\
\mathbf{fst} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \rightarrow \mathbf{exp}\ W\ T_1 \\
\mathbf{snd} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T_1 : \mathbf{tp}\,.\,\Pi T_2 : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ W\ (\mathbf{product}\ T_1\ T_2) \rightarrow \mathbf{exp}\ W\ T_2 \\
\mathbf{z} & : & \Pi W : \mathbf{wumber}\,.\,\mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{s} & : & \Pi W : \mathbf{wumber}\,.\,\mathbf{exp}\ W\ \mathbf{nat} \rightarrow \mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{case} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ W\ \mathbf{nat} \rightarrow \mathbf{exp}\ W\ T \rightarrow (\mathbf{exp}\ W\ \mathbf{nat} \rightarrow \mathbf{exp}\ W\ T) \rightarrow \\
 & & \quad \mathbf{exp}\ W\ T \\
\mathbf{next} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ (\mathbf{wumber\_n}\ W)\ T \rightarrow \mathbf{exp}\ W\ (\mathbf{circle}\ T) \\
\mathbf{prev} & : & \Pi W : \mathbf{wumber}\,.\,\Pi T : \mathbf{tp}\,. \\
 & & \mathbf{exp}\ W\ (\mathbf{circle}\ T) \rightarrow \mathbf{exp}\ (\mathbf{wumber\_n}\ W)\ T
\end{array}
$$

The corresponding LF representation functions working on types, contexts and typed expressions respectively are defined in the following way:

The function $\boxed{[\![\tau]\!]^{\mathrm{te}}}$ mapping a type τ into an LF object T:

$$
\begin{array}{rcl}
[\![\mathbf{nat}]\!]^{\mathrm{te}} & = & \mathbf{nat} \\
[\![\tau_1 \rightarrow \tau_2]\!]^{\mathrm{te}} & = & \mathbf{arrow}\ [\![\tau_1]\!]^{\mathrm{te}}\ [\![\tau_2]\!]^{\mathrm{te}} \\
[\![\tau_1 \times \tau_2]\!]^{\mathrm{te}} & = & \mathbf{product}\ [\![\tau_1]\!]^{\mathrm{te}}\ [\![\tau_2]\!]^{\mathrm{te}} \\
[\![\bigcirc \tau]\!]^{\mathrm{te}} & = & \mathbf{circle}\ [\![\tau]\!]^{\mathrm{te}}
\end{array}
$$

The function $\boxed{[\![w]\!]^{\mathrm{te}}}$ mapping a world number w into an LF object W:

$$
\begin{array}{rcl}
[\![0]\!]^{\mathrm{te}} & = & \mathbf{wumber\_0} \\
[\![w+1]\!]^{\mathrm{te}} & = & \mathbf{wumber\_n}\ [\![w]\!]^{\mathrm{te}}
\end{array}
$$

The function $\boxed{[\![\Gamma]\!]^{\mathrm{te}}}$ mapping a context Γ into an LF context Λ:

$$\begin{array}{rcl} \llbracket \cdot \rrbracket^{\mathrm{te}} & = & \cdot \\ \llbracket \Gamma,\ x : \tau^w \rrbracket^{\mathrm{te}} & = & \llbracket \Gamma \rrbracket^{\mathrm{te}},\ x_w : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}} \end{array}$$

Finally, the function $\boxed{\llbracket \mathcal{F} \rrbracket^{\mathrm{te}}}$ mapping the typing derivation for a temporal expression e into an LF object M such that $\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} M : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}$ when $\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_w e : \tau$:

$$\left\llbracket \frac{x : \tau^w \in \Gamma}{\Gamma \vdash^{\mathrm{te}}_w x : \tau} \right\rrbracket^{\mathrm{te}} = x_w$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma\{x \mapsto \tau_1^w\} \vdash^{\mathrm{te}}_w e : \tau_2}{\Gamma \vdash^{\mathrm{te}}_w \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \right\rrbracket^{\mathrm{te}} =$$

$$\mathbf{lam}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}}\ \llbracket \tau_2 \rrbracket^{\mathrm{te}}\ (\lambda x : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}} . \llbracket \mathcal{F} \rrbracket^{\mathrm{te}})$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Gamma \vdash^{\mathrm{te}}_w e_1 : \tau_1 \to \tau_2 \qquad \mathcal{F}_2 :: \Gamma \vdash^{\mathrm{te}}_w e_2 : \tau_1}{\Gamma \vdash^{\mathrm{te}}_w e_1\ e_2 : \tau_2} \right\rrbracket^{\mathrm{te}} =$$

$$\mathbf{app}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}}\ \llbracket \tau_2 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F}_1 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F}_2 \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma\{x \mapsto \tau^w\} \vdash^{\mathrm{te}}_w e : \tau}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{fix}\ x : \tau . e : \tau} \right\rrbracket^{\mathrm{te}} =$$

$$\mathbf{fix}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\ (\lambda x : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}} . \llbracket \mathcal{F} \rrbracket^{\mathrm{te}})$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Gamma \vdash^{\mathrm{te}}_w e_1 : \tau_1 \qquad \mathcal{F}_2 :: \Gamma \vdash^{\mathrm{te}}_w e_2 : \tau_2}{\Gamma \vdash^{\mathrm{te}}_w \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \right\rrbracket^{\mathrm{te}} =$$

$$\mathbf{pair}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}}\ \llbracket \tau_2 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F}_1 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F}_2 \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_w e : \tau_1 \times \tau_2}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{fst}\ e : \tau_1} \right\rrbracket^{\mathrm{te}} = \mathbf{fst}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}}\ \llbracket \tau_2 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_w e : \tau_1 \times \tau_2}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{fst}\ e : \tau_2} \right\rrbracket^{\mathrm{te}} = \mathbf{snd}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau_1 \rrbracket^{\mathrm{te}}\ \llbracket \tau_2 \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{z} : \mathbf{nat}} \right\rrbracket^{\mathrm{te}} = \mathbf{z}\ \llbracket w \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_w e : \mathbf{nat}}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{s}\ e : \mathbf{nat}} \right\rrbracket^{\mathrm{te}} = \mathbf{s}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_w e : \mathbf{nat} \qquad \mathcal{F}_1 :: \Gamma \vdash^{\mathrm{te}}_w e_1 : \tau \qquad \mathcal{F}_2 :: \Gamma\{x \mapsto \mathbf{nat}^w\} \vdash^{\mathrm{te}}_w e_2 : \tau}{\Gamma \vdash^{\mathrm{te}}_w \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 : \tau} \right\rrbracket^{\mathrm{te}}$$

$$= \mathbf{case}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F}_1 \rrbracket^{\mathrm{te}}\ (\lambda x : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \mathbf{nat}\,.\ \llbracket \mathcal{F}_2 \rrbracket^{\mathrm{te}})$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_{w+1} e : \tau}{\Gamma \vdash^{\mathrm{te}}_{w} \mathbf{next}\ e : \bigcirc \tau} \right\rrbracket^{\mathrm{te}} = \mathbf{next}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_{w} e : \bigcirc \tau}{\Gamma \vdash^{\mathrm{te}}_{w+1} \mathbf{prev}\ e : \tau} \right\rrbracket^{\mathrm{te}} = \mathbf{prev}\ \llbracket w+1 \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\ \llbracket \mathcal{F} \rrbracket^{\mathrm{te}}$$

Note how the world number attached to the typing judgement plays the role of the k used in the previous chapter.

The type and world number representation can be proved adequate by straightforward induction on the structure of the type τ and world number w respectively. Also lemmas similar to **Lemma 3.4** can be proved. The adequacy of the representation of well-typed temporal expressions is stated and proved more formally below.

Theorem 5.1 (Expression representation adequacy $\rightarrow$)**.** If $\mathcal{F} :: \Gamma \vdash^{\mathrm{te}}_{w} e : \tau$ then

$$\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} \llbracket \mathcal{F} \rrbracket^{\mathrm{te}} : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\,.$$

Proof. The proof is done by induction on the structure of the derivation $\mathcal{F}$. We only write down the cases of TP$^{\mathrm{te}}$-NEXT and TP$^{\mathrm{te}}$-PREV being the last rule applied in $\mathcal{F}$, as the rest of the cases are very similar to cases in adequacy proofs in earlier chapters.

1. TP$^{\mathrm{te}}$-NEXT. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}' :: \Gamma \vdash^{\mathrm{te}}_{w+1} e' : \tau'}{\Gamma \vdash^{\mathrm{te}}_{w} \mathbf{next}\ e' : \bigcirc \tau'}$$

and applying the induction hypothesis to $\mathcal{F}'$ we get that

$$\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} \left\llbracket \mathcal{F}' \right\rrbracket^{\mathrm{te}} : \mathbf{exp}\ \llbracket w+1 \rrbracket^{\mathrm{te}}\ \left\llbracket \tau' \right\rrbracket^{\mathrm{te}}.$$

As $\llbracket w+1 \rrbracket^{\mathrm{te}} = \mathbf{wumber\_n}\ \llbracket w \rrbracket^{\mathrm{te}}$ and as

$$\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} \llbracket w \rrbracket^{\mathrm{te}} : \mathbf{wumber}$$

and

$$\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} \left\llbracket \tau' \right\rrbracket^{\mathrm{te}} : \mathbf{tp}$$

due to adequacy of type and world number representation respectively, we can now apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\llbracket \Gamma \rrbracket^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} \mathbf{next}\ \llbracket w \rrbracket^{\mathrm{te}}\ \left\llbracket \tau' \right\rrbracket^{\mathrm{te}}\ \left\llbracket \mathcal{F}' \right\rrbracket^{\mathrm{te}} : \mathbf{exp}\ \llbracket w \rrbracket^{\mathrm{te}}\ (\mathbf{circle}\ \left\llbracket \tau' \right\rrbracket^{\mathrm{te}}).$$

As $\llbracket \tau \rrbracket^{\mathrm{te}} = \llbracket \bigcirc \tau' \rrbracket^{\mathrm{te}} = \mathbf{circle}\ \left\llbracket \tau' \right\rrbracket^{\mathrm{te}}$ and $\left\llbracket \mathcal{F} \right\rrbracket^{\mathrm{te}} = \mathbf{next}\ \llbracket w \rrbracket^{\mathrm{te}}\ \llbracket \tau \rrbracket^{\mathrm{te}}\ \left\llbracket \mathcal{F}' \right\rrbracket^{\mathrm{te}}$, we are done.

2. TP$^{\text{te}}$-PREV. In this case $\mathcal{F}$ is of the form

$$\frac{\mathcal{F}' :: \Gamma \vdash^{\text{te}}_{w} e' : \bigcirc\tau}{\Gamma \vdash^{\text{te}}_{w+1} \textbf{prev}\ e' : \tau}$$

and applying the induction hypothesis to $\mathcal{F}'$ we get that

$$\llbracket\Gamma\rrbracket^{\text{te}} \vdash^{\text{LF}}_{\Sigma_{\text{te}}} \llbracket\mathcal{F}'\rrbracket^{\text{te}} : \textbf{exp}\ \llbracket w\rrbracket^{\text{te}}\ \llbracket\bigcirc\tau\rrbracket^{\text{te}}.$$

As $\llbracket\bigcirc\tau\rrbracket^{\text{te}} = \textbf{circle}\ \llbracket\tau\rrbracket^{\text{te}}$ and as

$$\llbracket\Gamma\rrbracket^{\text{te}} \vdash^{\text{LF}}_{\Sigma_{\text{te}}} \llbracket w\rrbracket^{\text{te}} : \textbf{wumber}$$

and

$$\llbracket\Gamma\rrbracket^{\text{te}} \vdash^{\text{LF}}_{\Sigma_{\text{te}}} \llbracket\tau\rrbracket^{\text{te}} : \textbf{tp}$$

due to adequacy of type and world number representation respectively, we can now apply the TP$^{\text{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\llbracket\Gamma\rrbracket^{\text{te}} \vdash^{\text{LF}}_{\Sigma_{\text{te}}} \textbf{prev}\ \llbracket w\rrbracket^{\text{te}}\ \llbracket\tau\rrbracket^{\text{te}}\ \llbracket\mathcal{F}'\rrbracket^{\text{te}} : \textbf{exp}\ (\textbf{wumber\_n}\ \llbracket w\rrbracket^{\text{te}})\ \llbracket\tau\rrbracket^{\text{te}}.$$

As $\llbracket w+1\rrbracket^{\text{te}} = \textbf{wumber\_n}\ \llbracket w\rrbracket^{\text{te}}$ and $\llbracket\mathcal{F}\rrbracket^{\text{te}} = \textbf{prev}\ \llbracket w\rrbracket^{\text{te}}\ \llbracket\tau\rrbracket^{\text{te}}\ \llbracket\mathcal{F}'\rrbracket^{\text{te}}$, we are done. □

To prove the second adequacy theorem we will need the following lemma:

Lemma 5.1. For a Mini-ML$^{\bigcirc}$ context Γ the LF context $\llbracket\Gamma\rrbracket^{\text{te}}$ will only contain declarations of the form $x : \textbf{exp}\ W\ T$ and for any such declaration we can always find a w, τ, and y such that $W = \llbracket w\rrbracket^{\text{te}}$, $T = \llbracket\tau\rrbracket^{\text{im}}$, $\Gamma(y) = \tau^w$, and $x = y_w$.

Proof. The proof can be done by straightforward induction on the structure of Γ. □

Theorem 5.2 (Expression representation adequacy $\leftarrow$)**.** If M is a canonical LF object and if

$$\llbracket\Gamma\rrbracket^{\text{te}} \vdash^{\text{LF}}_{\Sigma_{\text{te}}} M : \textbf{exp}\ \llbracket w\rrbracket^{\text{te}}\ \llbracket\tau\rrbracket^{\text{te}}$$

then we can find a typing derivation $\mathcal{F} :: \Gamma \vdash^{\text{te}}_{w} e : \tau$ such that $\llbracket\mathcal{F}\rrbracket^{\text{te}} = M$.

Proof. The proof is done by induction on the structure of the canonical LF object M. According to lemma **Lemma 5.1** the context $\llbracket\Gamma\rrbracket^{\text{te}}$ cannot contain any variables of function type, and thus we will have no proof cases concerning variable applications. The cases below are the proof cases which are not completely similar to cases in earlier adequacy proofs.

1. $M = x$. According to the rule TP$^{\text{LF}}$-TERM-VAR we must have that $\llbracket\Gamma\rrbracket^{\text{te}}(x) = \textbf{exp}\ \llbracket w\rrbracket^{\text{te}}\ \llbracket\tau\rrbracket^{\text{te}}$ and due to **Lemma 5.1** we then know that M must be the variable y_w where $\Gamma(y) = \tau$. Choosing $\mathcal{F}$ to be

$$\frac{\Gamma(y) = \tau^w}{\Gamma \vdash^{\text{te}}_{w} y : \tau}$$

we have that $[\![\mathcal{F}]\!]^{\mathrm{te}} = y_w = M$ and we are done.

2. $M = \mathbf{next}\ W\ T\ M'$. From the rules for typing of LF objects we can derive that $W = [\![w]\!]^{\mathrm{te}}$ and $\mathbf{circle}\ T = [\![\tau]\!]^{\mathrm{te}}$ and that

$$[\![\Gamma]\!]^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} M' : \mathbf{exp}\ (\mathbf{wumber\_n}\ [\![w]\!]^{\mathrm{te}})\ T.$$

Due to a suitable variant of **Lemma 3.4** we can find τ' such that $\tau = \bigcirc\tau'$ and such that $T = [\![\tau']\!]^{\mathrm{te}}$ and as also $\mathbf{wumber\_n}\ [\![w]\!]^{\mathrm{te}} = [\![w+1]\!]^{\mathrm{te}}$, we rewrite the last judgement into

$$[\![\Gamma]\!]^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} M' : \mathbf{exp}\ [\![w+1]\!]^{\mathrm{te}}\ [\![\tau']\!]^{\mathrm{te}}.$$

Now we can apply the induction hypothesis to M' and get a derivation $\mathcal{F}' :: \Gamma \vdash^{\mathrm{te}}_{w+1} e' : \tau'$ such that $[\![\mathcal{F}']\!]^{\mathrm{te}} = M'$ and then by application of the TP$^{\mathrm{te}}$-NEXT rule we get a derivation of $\Gamma \vdash^{\mathrm{te}}_{w} \mathbf{next}\ e' : \bigcirc\tau'$. As $\bigcirc\tau' = \tau$ and as the derivation is represented by the LF object $\mathbf{next}\ [\![w]\!]^{\mathrm{te}}\ [\![\tau']\!]^{\mathrm{te}}\ [\![\mathcal{F}']\!]^{\mathrm{te}}$ which is α-equivalent to M, we are done.

3. $M = \mathbf{prev}\ W\ T\ M'$. From the rules for typing of LF objects we can derive that $\mathbf{wumber\_n}\ W = [\![w]\!]^{\mathrm{te}}$ and $T = [\![\tau]\!]^{\mathrm{te}}$ and that

$$[\![\Gamma]\!]^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} M' : \mathbf{exp}\ W\ (\mathbf{circle}\ [\![\tau]\!]^{\mathrm{te}}).$$

Due to a suitable variant of **Lemma 3.4** we can find w' such that $w = w'+1$ and such that $W = [\![w']\!]^{\mathrm{te}}$ and as also $\mathbf{circle}\ [\![\tau]\!]^{\mathrm{te}} = [\![\bigcirc\tau]\!]^{\mathrm{te}}$, we can rewrite the last judgement into

$$[\![\Gamma]\!]^{\mathrm{te}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{te}}} M' : \mathbf{exp}\ [\![w']\!]^{\mathrm{te}}\ [\![\bigcirc\tau]\!]^{\mathrm{te}}.$$

Now we can apply the induction hypothesis to M' and get a derivation $\mathcal{F}' :: \Gamma \vdash^{\mathrm{te}}_{w'} e' : \bigcirc\tau$ such that $[\![\mathcal{F}']\!]^{\mathrm{te}} = M'$ and then by application of the TP$^{\mathrm{te}}$-PREV rule we get a derivation of $\Gamma \vdash^{\mathrm{te}}_{w'+1} \mathbf{prev}\ e' : \tau$. As $w'+1 = w$ and as the derivation is represented by the LF object $\mathbf{next}\ [\![w]\!]^{\mathrm{te}}\ [\![\tau]\!]^{\mathrm{te}}\ [\![\mathcal{F}']\!]^{\mathrm{te}}$ which is α-equivalent to M, we are done. □

5.4 Operational semantics for Mini-ML$^{\bigcirc}$

The values among the expressions in Mini-ML$^{\bigcirc}$ are given by

$$\begin{array}{lrcl}
\text{Values:} & v^0 & ::= & \lambda x:\tau\,.\,e \mid \langle v_1^0, v_2^0\rangle \mid \mathbf{z} \mid \mathbf{s}\ v^0 \mid \mathbf{next}\ v^1 \\
& v^{w+1} & ::= & x \mid \lambda x:\tau\,.\,v^{w+1} \mid v_1^{w+1}\ v_2^{w+1} \mid \\
& & & \mathbf{fix}\ x:\tau\,.\,v^{w+1} \mid \\
& & & \langle v_1^{w+1}, v_2^{w+1}\rangle \mid \mathbf{fst}\ v^{w+1} \mid \mathbf{snd}\ v^{w+1} \mid \\
& & & \mathbf{z} \mid \mathbf{s}\ v^{w+1} \mid \mathbf{case}\ v^{w+1}\ \mathbf{of}\ \mathbf{z} \Rightarrow v_1^{w+1} \mid \mathbf{s}\ x \Rightarrow v_2^{w+1} \mid \\
& & & \mathbf{next}\ v^{w+2} \mid \mathbf{prev}\ v^{w}\ (\text{when } w > 0)
\end{array}$$

Big-step evaluation

The big-step evaluation relation $\boxed{e \hookrightarrow^{\mathrm{te}}_{w} v}$ evaluating a temporal expression e to a temporal value v in the wth world is defined by the following inference rules:

$$\frac{}{x \hookrightarrow^{\mathrm{te}}_{w+1} x}\ \text{EVAL}^{\mathrm{te}}\text{-VAR-W+1}$$

$$\frac{}{\lambda x : \tau \,.\, e \hookrightarrow^{\mathrm{te}}_{0} \lambda x : \tau \,.\, e}\ \text{EVAL}^{\mathrm{te}}\text{-ABS-0} \qquad \frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\lambda x : \tau \,.\, e \hookrightarrow^{\mathrm{te}}_{w+1} \lambda x : \tau \,.\, v}\ \text{EVAL}^{\mathrm{te}}\text{-ABS-W+1}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{0} \lambda x : \tau \,.\, e_1' \qquad e_2 \hookrightarrow^{\mathrm{te}}_{0} v_2 \qquad \{v_2/x\}\, e_1' \hookrightarrow^{\mathrm{te}}_{0} v}{e_1\ e_2 \hookrightarrow^{\mathrm{te}}_{0} v}\ \text{EVAL}^{\mathrm{te}}\text{-APP-0}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{w+1} v_1 \qquad e_2 \hookrightarrow^{\mathrm{te}}_{w+1} v_2}{e_1\ e_2 \hookrightarrow^{\mathrm{te}}_{w+1} v_1\ v_2}\ \text{EVAL}^{\mathrm{te}}\text{-APP-W+1}$$

$$\frac{\{\mathbf{fix}\ x : \tau \,.\, e/x\}\, e \hookrightarrow^{\mathrm{te}}_{0} v}{\mathbf{fix}\ x : \tau \,.\, e \hookrightarrow^{\mathrm{te}}_{0} v}\ \text{EVAL}^{\mathrm{te}}\text{-FIX-0}$$

$$\frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\mathbf{fix}\ x : \tau \,.\, e \hookrightarrow^{\mathrm{te}}_{w+1} \mathbf{fix}\ x : \tau \,.\, v}\ \text{EVAL}^{\mathrm{te}}\text{-FIX-W+1}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{w} v_1 \qquad e_2 \hookrightarrow^{\mathrm{te}}_{w} v_2}{\langle e_1,\, e_2\rangle \hookrightarrow^{\mathrm{te}}_{w} \langle v_1,\, v_2\rangle}\ \text{EVAL}^{\mathrm{te}}\text{-PAIR-W}$$

$$\frac{e \hookrightarrow^{\mathrm{te}}_{0} \langle v_1,\, v_2\rangle}{\mathbf{fst}\ e \hookrightarrow^{\mathrm{te}}_{0} v_1}\ \text{EVAL}^{\mathrm{te}}\text{-FST-0} \qquad \frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\mathbf{fst}\ e \hookrightarrow^{\mathrm{te}}_{w+1} \mathbf{fst}\ v}\ \text{EVAL}^{\mathrm{te}}\text{-FST-W+1}$$

$$\frac{e \hookrightarrow^{\mathrm{te}}_{0} \langle v_1,\, v_2\rangle}{\mathbf{snd}\ e \hookrightarrow^{\mathrm{te}}_{0} v_2}\ \text{EVAL}^{\mathrm{te}}\text{-SND-0} \qquad \frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\mathbf{snd}\ e \hookrightarrow^{\mathrm{te}}_{w+1} \mathbf{snd}\ v}\ \text{EVAL}^{\mathrm{te}}\text{-SND-W+1}$$

$$\frac{}{\mathbf{z} \hookrightarrow^{\mathrm{te}}_{w} \mathbf{z}}\ \text{EVAL}^{\mathrm{te}}\text{-ZERO-W} \qquad \frac{e \hookrightarrow^{\mathrm{te}}_{w} v}{\mathbf{s}\ e \hookrightarrow^{\mathrm{te}}_{w} \mathbf{s}\ v}\ \text{EVAL}^{\mathrm{te}}\text{-SUCC-W}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{0} \mathbf{z} \qquad e_2 \hookrightarrow^{\mathrm{te}}_{0} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\mathrm{te}}_{0} v}\ \text{EVAL}^{\mathrm{te}}\text{-CASE-Z-0}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{0} \mathbf{s}\ v_1 \qquad \{v_1/x\}\, e_3 \hookrightarrow^{\mathrm{te}}_{0} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\mathrm{te}}_{0} v}\ \text{EVAL}^{\mathrm{te}}\text{-CASE-S-0}$$

$$\frac{e_1 \hookrightarrow^{\mathrm{te}}_{w+1} v_1 \qquad e_2 \hookrightarrow^{\mathrm{te}}_{w+1} v_2 \qquad e_3 \hookrightarrow^{\mathrm{te}}_{w+1} v_3}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\mathrm{te}}_{w+1} \mathbf{case}\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow v_2 \mid \mathbf{s}\ x \Rightarrow v_3}\ \textsc{eval}^{\mathrm{te}}\textsc{-case-w+1}$$

$$\frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\mathbf{next}\ e \hookrightarrow^{\mathrm{te}}_{w} \mathbf{next}\ v}\ \textsc{eval}^{\mathrm{te}}\textsc{-next-w}$$

$$\frac{e \hookrightarrow^{\mathrm{te}}_{0} \mathbf{next}\ v}{\mathbf{prev}\ e \hookrightarrow^{\mathrm{te}}_{1} v}\ \textsc{eval}^{\mathrm{te}}\textsc{-prev-1} \qquad \frac{e \hookrightarrow^{\mathrm{te}}_{w+1} v}{\mathbf{prev}\ e \hookrightarrow^{\mathrm{te}}_{w+2} \mathbf{prev}\ v}\ \textsc{eval}^{\mathrm{te}}\textsc{-prev-w+2}$$

Having the big-step operational semantics in place we are now ready to consider a Mini-ML$^{\bigcirc}$ version of the power programs considered in the last chapter:

$$\begin{array}{lll} \mathit{power}^{\mathrm{te}} & \equiv & \lambda n : \mathbf{nat}\,.\,\mathbf{next}\ (\lambda x : \mathbf{nat}\,.\,\mathbf{prev}\,(\\ & & \quad (\mathbf{fix}\ p : \mathbf{nat} \to \bigcirc\,\mathbf{nat}\,. \\ & & \qquad \lambda n' : \mathbf{nat}\,. \\ & & \qquad\quad \mathbf{case}\ n'\ \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{next}\ (\mathbf{s}\ \mathbf{z}) \\ & & \qquad\qquad\quad \mid\ \mathbf{s}\ m \Rightarrow \mathbf{next}\ (\mathit{times}\ x\ (\mathbf{prev}\ (p\ m))))\ n)) \end{array}$$

For $n = 2$ we can derive the following evaluation due to the operational semantics given above:

$$\mathit{power}^{\mathrm{te}}\ (\mathbf{s}\ (\mathbf{s}\ \mathbf{z})) \hookrightarrow^{\mathrm{te}}_{0} \mathbf{next}\ (\lambda x : \mathbf{nat}\,.\,\mathit{time}\ x\ (\mathit{times}\ x\ (\mathbf{s}\ \mathbf{z})))$$

As expected, the superfluous β-redexes which occurred in the previous chapter can no longer be spotted here.

5.5 Representing the Mini-ML$^{\bigcirc}$ operational semantics in LF

Similar to what we did in the Mini-ML$^{\Box}_{\mathrm{ex}}$ case, the value sub-grammar defined above is translated into a value judgement before representation within LF. Again using v to range over expressions expected to be values, we define a Mini-ML$^{\bigcirc}$ value judgement $\boxed{\mathrm{value}^{\mathrm{te}}_{w}(v)}$ by the following inference rules:

$$\frac{}{\mathrm{value}^{\mathrm{te}}_{0}(\lambda x : \tau\,.\,e)}\ \textsc{val}^{\mathrm{te}}\textsc{-abs-0} \qquad \frac{\mathrm{value}^{\mathrm{te}}_{w+1}(v)}{\mathrm{value}^{\mathrm{te}}_{w+1}(\lambda x : \tau\,.\,v)}\ \textsc{val}^{\mathrm{te}}\textsc{-abs-w+1}$$

$$\frac{\mathrm{value}^{\mathrm{te}}_{w}(v_1) \qquad \mathrm{value}^{\mathrm{te}}_{w}(v_2)}{\mathrm{value}^{\mathrm{te}}_{w}(\langle v_1,\, v_2\rangle)}\ \textsc{val}^{\mathrm{te}}\textsc{-pair-w}$$

$$\frac{}{\text{value}^{\text{te}}_w(\mathbf{z})}\ \text{VAL}^{\text{te}}\text{-ZERO-W} \qquad \frac{\text{value}^{\text{te}}_w(v)}{\text{value}^{\text{te}}_w(\mathbf{s}\ v)}\ \text{VAL}^{\text{te}}\text{-SUCC-W}$$

$$\frac{}{\text{value}^{\text{te}}_{w+1}(x)}\ \text{VAL}^{\text{te}}\text{-VAR-W+1}$$

$$\frac{\text{value}^{\text{te}}_{w+1}(v_1) \qquad \text{value}^{\text{te}}_{w+1}(v_2)}{\text{value}^{\text{te}}_{w+1}(v_1\ v_2)}\ \text{VAL}^{\text{te}}\text{-APP-W+1}$$

$$\frac{\text{value}^{\text{te}}_{w+1}(v_1)}{\text{value}^{\text{te}}_{w+1}(\mathbf{fix}\ x : \tau\ .\ v)}\ \text{VAL}^{\text{te}}\text{-FIX-W+1}$$

$$\frac{\text{value}^{\text{te}}_{w+1}(v)}{\text{value}^{\text{te}}_{w+1}(\mathbf{fst}\ v)}\ \text{VAL}^{\text{te}}\text{-FST-W+1} \qquad \frac{\text{value}^{\text{te}}_{w+1}(v)}{\text{value}^{\text{te}}_{w+1}(\mathbf{snd}\ v)}\ \text{VAL}^{\text{te}}\text{-SND-W+1}$$

$$\frac{\text{value}^{\text{te}}_{w+1}(v) \qquad \text{value}^{\text{te}}_{w+1}(v_1) \qquad \text{value}^{\text{te}}_{w+1}(v_2)}{\text{value}^{\text{te}}_{w+1}(\mathbf{case}\ v\ \mathbf{of}\ \mathbf{z} \Rightarrow v_1 \mid \mathbf{s}\ x \Rightarrow v_2)}\ \text{VAL}^{\text{te}}\text{-CASE-W+1}$$

$$\frac{\text{value}^{\text{te}}_{w+1}(v)}{\text{value}^{\text{te}}_{w}(\mathbf{next}\ v)}\ \text{VAL}^{\text{te}}\text{-NEXT-W} \qquad \frac{\text{value}^{\text{te}}_{w}(v)}{\text{value}^{\text{te}}_{w+1}(\mathbf{prev}\ v)}\ \text{VAL}^{\text{te}}\text{-PREV-W+1}$$

In the Twelf code in Section **A.5** the type family `val_t` is the one used for the representation of Mini-ML$^{\bigcirc}$ value judgements. The family is indexed by a wumber and a temporal expression. Note that the VAL$^{\text{te}}$-VAR-W+1 rule does not have its own object constant declared but only shows up as a local assumption in the term constants representing the variable-binding rules VAL$^{\text{te}}$-ABS-W+1, VAL$^{\text{te}}$-CASE-W+1, and VAL$^{\text{te}}$-FIX-W+1.

Similar to what happened in the Mini-ML$^{\Box}_{\text{ex}}$ case, the big-step evaluation relation $e \hookrightarrow^{\text{te}}_w v$ is now defined as a subset of the expression product set, and the fact that the second expression is actually a value now has to be explicitly proved. This value soundness proof can be done by straightforward induction on the structure of the evaluation derivation and in our Twelf code the type family `val_sound_t` is the one representing this. The type family used for the evaluation relation itself is the one named `eval_t`.

5.6 Metatheory about Mini-ML$^{\bigcirc}$

Besides from value soundness which was considered above we will also machine-verify determinacy and type preservation of the big-step operational semantics for Mini-ML$^{\bigcirc}$:

Theorem 5.3 (Evaluation determinacy)**.** If $e \hookrightarrow^{\text{te}}_w v$ and $e \hookrightarrow^{\text{te}}_w v'$ then $v = v'$.

Proof. The theorem can be proved by straightforward induction on the structure of the evaluation derivation. □

Theorem 5.4 (Type preservation)**.** If $e \hookrightarrow^{\text{te}}_{w} v$ and $\Gamma \vdash^{\text{te}}_{w} e : \tau$ then also $\Gamma \vdash^{\text{te}}_{w} v : \tau$.

Proof. The theorem can be proved by straightforward induction on the structure of the evaluation derivation using an appropriate substitution lemma. □

5.7 Representing the Mini-ML$^{\Box}_{\text{ex}}$ metatheory in LF

The representation of evaluation determinacy proceeds similar to the explicit case and the main type family used for this is the one named `eval_t_det` in Section **A.5**.

The type preservation theorem follows for free when using LF syntax indexed with world number and type.

Chapter 6

Staged Computation and Modal Logic of Contextual Necessity

The subject of this chapter is the third and last metaprogramming system to be considered in this project. The system is described by Nanevski *et al.* in [6] and is a system based on a so called contextual modal logic. This contextual modal logic can be seen as a generalization of the logic of necessity on which the system in **Chapter 4** was based, and an important objective of its development was to achieve a basis for a metaprogramming language comprising the best of the previous two systems: fewer superfluous β-redexes in generated code and possible instant execution of generated code.

In this chapter computation stages are to be compared to the worlds of a Kripke model of the same structure as in **Chapter 4**, but the necessity operator $\Box$ is here replaced by a version to which a list of types can be attached. Variables of these types are then allowed to occur freely within the universally valid formula in question.

Below we will consider the extension of Mini-ML resulting from an extension of the Curry-Howard isomorphism to include the contextual modal logic presented in [6]. This functional metaprogramming language will be referred to as Mini-$\mathrm{ML}_{\mathrm{co}}$.

6.1 Syntax of Mini-$\mathrm{ML}_{\mathrm{co}}$

As we will not consider dependent types at any point, we have chosen to simplify the notation given in [6] in the following way: wherever reasonable we will use position association instead of name association of variables occurring freely within code. The syntax categories of Mini-$\mathrm{ML}_{\mathrm{co}}$ are then given by

$$
\begin{array}{rrcl}
\text{Types:} & \tau & ::= & \mathbf{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid [\Sigma]\tau \\
\text{Expressions:} & e & ::= & x \mid \lambda x : \tau \,.\, e \mid e_1\ e_2 \mid \mathbf{fix}\ x : \tau \,.\, e \mid \langle e_1, e_2 \rangle \mid \\
 & & & \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \mid \\
 & & & \mathbf{box}\ b \mid \mathbf{let}\ \mathbf{box}\ u = e_1\ \mathbf{in}\ e_2 \mid \mathbf{clo}\ (u, S) \\
\text{Box-arguments:} & b & ::= & e \mid x : \tau \,.\, b
\end{array}
$$

$$\begin{array}{rrcl}
\text{Expression lists:} & S & ::= & \cdot \mid S,\, e \\
\text{Type lists:} & \Sigma & ::= & \cdot \mid \Sigma,\, \tau \\
\\
\text{Modal contexts:} & \Delta & ::= & \cdot \mid \Delta,\ u :: \tau[\Sigma] \\
\text{Non-modal contexts:} & \Gamma & ::= & \cdot \mid \Gamma,\ x : \tau
\end{array}$$

This syntax is very similar to the syntax of Mini-ML$^{\Box}_{\text{ex}}$. The main difference is that the code within a box-constructor now has a list of variable declarations in front where each of the listed variables might occur freely within the code, and then, instead of writing just u, we now have to write **clo** (u, S) where S is a list of expressions to be substituted for the variables occurring freely in the code expression referred to by u. The code type constructor $\Box$ used in **Chapter 4** is here replaced by the constructor $[\Sigma]$ where Σ is the types of the free variables occurring in the code being classified. A modal variable u in Mini-ML$^{\Box}_{\text{ex}}$ corresponds to **clo** $(u, \cdot)$ in Mini-ML$_{\text{co}}$, and a code type $\Box\,\tau$ corresponds to $[\cdot]\tau$.

We will again use $\boxed{\Gamma_1,\ \Gamma_2}$ to mean the concatenation of the non-modal contexts Γ_1 and Γ_2 and similarly $\boxed{\Delta_1,\ \Delta_2}$ to mean the concatenation of the modal contexts Δ_1 and Δ_2.

Also the functions $\boxed{\Gamma\{x \mapsto \tau\}}$ and $\boxed{\Delta\{u \mapsto \tau[\Sigma]\}}$ updating a non-modal and a modal context with a new assumption respectively are defined just as in the previous chapters.

In the following we presume validity for modal and non-modal contexts in the sense of unique variable names unless otherwise explicitly mentioned.

6.2 Typing rules for Mini-ML$_{\text{co}}$

The typing judgement for Mini-ML$_{\text{co}}$ has the form $\boxed{\Delta \mid \Gamma \vdash^{\text{co}} e : \tau}$ and with the assumption overwriting methodology incorporated it is defined by the following inference rules:

$$\frac{\Gamma(x) = \tau}{\Delta \mid \Gamma \vdash^{\text{co}} x : \tau}\ \text{TP}^{\text{co}}\text{-VAR-X} \qquad \frac{\Delta(u) = \tau[\Sigma] \qquad \Delta \mid \Gamma \vdash^{\text{co,S}} S : \Sigma}{\Delta \mid \Gamma \vdash^{\text{co}} \textbf{clo}\ (u, S) : \tau}\ \text{TP}^{\text{co}}\text{-VAR-U}$$

$$\frac{\Delta \mid \Gamma\{x \mapsto \tau_1\} \vdash^{\text{co}} e : \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \lambda x : \tau_1 \,.\, e : \tau_1 \to \tau_2}\ \text{TP}^{\text{co}}\text{-ABS}$$

$$\frac{\Delta \mid \Gamma \vdash^{\text{co}} e_1 : \tau_1 \to \tau_2 \qquad \Delta \mid \Gamma \vdash^{\text{co}} e_2 : \tau_1}{\Delta \mid \Gamma \vdash^{\text{co}} e_1\ e_2 : \tau_2}\ \text{TP}^{\text{co}}\text{-APP}$$

$$\frac{\Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\text{co}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{co}} \textbf{fix}\ x : \tau \,.\, e : \tau}\ \text{TP}^{\text{co}}\text{-FIX}$$

$$\frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e_1 \,:\, \tau_1 \qquad \Delta \mid \Gamma \vdash^{\mathrm{co}} e_2 \,:\, \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \langle e_1, e_2\rangle \,:\, \tau_1 \times \tau_2}\ \mathrm{TP}^{\mathrm{co}}\text{-PAIR}$$

$$\frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e \,:\, \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{fst}\ e \,:\, \tau_1}\ \mathrm{TP}^{\mathrm{co}}\text{-FST} \qquad \frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e \,:\, \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{snd}\ e \,:\, \tau_2}\ \mathrm{TP}^{\mathrm{co}}\text{-SND}$$

$$\frac{}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{z} \,:\, \mathbf{nat}}\ \mathrm{TP}^{\mathrm{co}}\text{-ZERO} \qquad \frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e \,:\, \mathbf{nat}}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{s}\ e \,:\, \mathbf{nat}}\ \mathrm{TP}^{\mathrm{co}}\text{-SUCC}$$

$$\frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e \,:\, \mathbf{nat} \qquad \Delta \mid \Gamma \vdash^{\mathrm{co}} e_1 \,:\, \tau \qquad \Delta \mid \Gamma\{x \mapsto \mathbf{nat}\} \vdash^{\mathrm{co}} e_2 \,:\, \tau}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \,:\, \tau}\ \mathrm{TP}^{\mathrm{co}}\text{-CASE}$$

$$\frac{\Delta \mid \cdot \vdash^{\mathrm{co,b}} b \,:\, [\Sigma]\tau}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{box}\ b \,:\, [\Sigma]\tau}\ \mathrm{TP}^{\mathrm{co}}\text{-BOX}$$

$$\frac{\Delta \mid \Gamma \vdash^{\mathrm{co}} e_1 \,:\, [\Sigma]\tau_1 \qquad \Delta\{u \mapsto \tau_1[\Sigma]\} \mid \Gamma \vdash^{\mathrm{co}} e_2 \,:\, \tau_2}{\Delta \mid \Gamma \vdash^{\mathrm{co}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \,:\, \tau_2}\ \mathrm{TP}^{\mathrm{co}}\text{-LET-BOX}$$

Here $\boxed{\Gamma(x) = \tau}$ means that $x : \tau$ is present in the non-modal context Γ and $\boxed{\Delta(u) = \tau[\Sigma]}$ means that $u :: \tau[\Sigma]$ is present in the modal context Δ.

Also the judgement $\boxed{\Delta \mid \Gamma \vdash^{\mathrm{co,b}} b \,:\, [\Sigma]\tau}$ typing contextual box-arguments is defined by

$$\frac{\Delta \mid \cdot \vdash^{\mathrm{co}} e \,:\, \tau}{\Delta \mid \Gamma \vdash^{\mathrm{co,b}} e \,:\, [\cdot]\tau}\ \mathrm{TP}^{\mathrm{co,b}}\text{-0} \qquad \frac{\Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\mathrm{co,b}} b \,:\, [\Sigma]\tau'}{\Delta \mid \Gamma \vdash^{\mathrm{co,b}} x : \tau \,.\, b \,:\, [\Sigma, \tau]\tau'}\ \mathrm{TP}^{\mathrm{co,b}}\text{-N}$$

and the judgement $\boxed{\Delta \mid \Gamma \vdash^{\mathrm{co,S}} S \,:\, \Sigma}$ typing lists of contextual expressions is defined by

$$\frac{}{\Delta \mid \Gamma \vdash^{\mathrm{co,S}} \cdot \,:\, \cdot}\ \mathrm{TP}^{\mathrm{co,S}}\text{-0} \qquad \frac{\Delta \mid \Gamma \vdash^{\mathrm{co,S}} S \,:\, \Sigma \qquad \Delta \mid \Gamma \vdash^{\mathrm{co}} e \,:\, \tau}{\Delta \mid \Gamma \vdash^{\mathrm{co,S}} (S, e) \,:\, (\Sigma, \tau)}\ \mathrm{TP}^{\mathrm{co,S}}\text{-N}$$

It should be rather clear that typing rules equivalent to the typing rules of Mini-ML$^{\Box}_{\mathrm{ex}}$ can be obtained by setting $\Sigma = \cdot$ and $S = \cdot$.

6.2.1 Representing well-typed Mini-ML$_{\mathrm{co}}$ expressions in LF

As we also did in the previous chapters we use an LF syntax where types and world scopes are intrinsically attached. The LF signature to be used, Σ_{co}, contains the following declarations:

$$
\begin{array}{rcl}
\textbf{world} & : & \textbf{type} \\
\\
\textbf{tp} & : & \textbf{type} \\
\textbf{ctx} & : & \textbf{type} \\
\textbf{sequence} & : & \textbf{world} \to \textbf{ctx} \to \textbf{type} \\
\\
\textbf{nat} & : & \textbf{tp} \\
\textbf{arrow} & : & \textbf{tp} \to \textbf{tp} \to \textbf{tp} \\
\textbf{product} & : & \textbf{tp} \to \textbf{tp} \to \textbf{tp} \\
\textbf{ccode} & : & \textbf{ctx} \to \textbf{tp} \to \textbf{tp} \\
\\
\textbf{ctx\_0} & : & \textbf{tp} \\
\textbf{ctx\_n} & : & \textbf{tp} \to \textbf{ctx} \to \textbf{tp} \\
\\
\textbf{sequence\_0} & : & \Pi\, W : \textbf{world}\,.\, \textbf{sequence}\ W\ \textbf{ctx\_0} \\
\textbf{sequence\_n} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, C : \textbf{ctx}\,.\, \Pi\, T : \textbf{tp}\,. \\
 & & \quad \textbf{exp}\ W\ T \to \textbf{sequence}\ W\ C \to \textbf{sequence}\ W\ (\textbf{ctx\_n}\ T\ C) \\
\\
\textbf{uvar} & : & \textbf{ctx} \to \textbf{tp} \to \textbf{type} \\
\textbf{exp} & : & \textbf{world} \to \textbf{tp} \to \textbf{type} \\
\textbf{boxarg} & : & \textbf{world} \to \textbf{ctx} \to \textbf{tp} \to \textbf{type} \\
\\
\textbf{lam} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T_1 : \textbf{tp}\,.\, \Pi\, T_2 : \textbf{tp}\,. \\
 & & \quad (\textbf{exp}\ W\ T_1 \to \textbf{exp}\ W\ T_2) \to \textbf{exp}\ W\ (\textbf{arrow}\ T_1\ T_2) \\
\textbf{app} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T_1 : \textbf{tp}\,.\, \Pi\, T_2 : \textbf{tp}\,. \\
 & & \quad \textbf{exp}\ W\ (\textbf{arrow}\ T_1\ T_2) \to \textbf{exp}\ W\ T_1 \to \textbf{exp}\ W\ T_2 \\
\textbf{fix} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T : \textbf{tp}\,. \\
 & & \quad (\textbf{exp}\ W\ T \to \textbf{exp}\ W\ T) \to \textbf{exp}\ W\ T \\
\textbf{pair} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T_1 : \textbf{tp}\,.\, \Pi\, T_2 : \textbf{tp}\,. \\
 & & \quad \textbf{exp}\ W\ T_1 \to \textbf{exp}\ W\ T_2 \to \textbf{exp}\ W\ (T_1 \times T_2) \\
\textbf{fst} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T_1 : \textbf{tp}\,.\, \Pi\, T_2 : \textbf{tp}\,. \\
 & & \quad \textbf{exp}\ W\ (T_1 \times T_2) \to \textbf{exp}\ W\ T_1 \\
\textbf{snd} & : & \Pi\, W : \textbf{world}\,.\, \Pi\, T_1 : \textbf{tp}\,.\, \Pi\, T_2 : \textbf{tp}\,. \\
 & & \quad \textbf{exp}\ W\ (T_1 \times T_2) \to \textbf{exp}\ W\ T_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{z} & : & \Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{s} & : & \Pi\, W : \mathbf{world}\,.\ \mathbf{exp}\ W\ \mathbf{nat} \ \rightarrow\ \mathbf{exp}\ W\ \mathbf{nat} \\
\mathbf{case} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, T : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ \mathbf{nat} \ \rightarrow\ \mathbf{exp}\ W\ T \ \rightarrow \\
 & & \quad\quad (\mathbf{exp}\ W\ \mathbf{nat} \ \rightarrow\ \mathbf{exp}\ W\ T) \ \rightarrow\ \mathbf{exp}\ W\ T \\
\mathbf{box} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, C : \mathbf{ctx}\,.\ \Pi\, T : \mathbf{tp}\,. \\
 & & \quad (\Pi\, W' : \mathbf{world}\,.\ \mathbf{boxarg}\ W'\ C\ T) \ \rightarrow \\
 & & \quad\quad \mathbf{exp}\ W\ (\mathbf{ccode}\ \Gamma\ T) \\
\mathbf{let\_box} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, C : \mathbf{ctx}\,.\ \Pi\, T_1 : \mathbf{tp}\,.\ \Pi\, T_2 : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ (\mathbf{ccode}\ C\ T_1) \ \rightarrow \\
 & & \quad\quad (\mathbf{uvar}\ C\ T_1 \ \rightarrow\ \mathbf{exp}\ W\ T_2) \ \rightarrow\ \mathbf{exp}\ W\ T_2 \\
\mathbf{clo} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, C : \mathbf{ctx}\,.\ \Pi\, T : \mathbf{tp}\,. \\
 & & \quad \mathbf{uvar}\ C\ T \ \rightarrow\ \mathbf{sequence}\ W\ C \ \rightarrow\ \mathbf{exp}\ W\ T
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{boxarg\_0} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, T : \mathbf{tp}\,. \\
 & & \quad \mathbf{exp}\ W\ T \ \rightarrow\ \mathbf{boxarg}\ W\ \mathbf{ctx\_0}\ T \\
\mathbf{boxarg\_n} & : & \Pi\, W : \mathbf{world}\,.\ \Pi\, C : \mathbf{ctx}\,.\ \Pi\, T' : \mathbf{tp}\,.\ \Pi\, T : \mathbf{tp}\,. \\
 & & \quad (\mathbf{exp}\ W\ T' \ \rightarrow\ \mathbf{boxarg}\ W\ C\ T) \ \rightarrow \\
 & & \quad\quad \mathbf{boxarg}\ W\ (\mathbf{ctx\_n}\ T'\ C)\ T
\end{array}
$$

These constant declarations are easiest understood by inspection of the corresponding representation functions defined below.

The function $\boxed{\llbracket \tau \rrbracket^{\text{co}}}$ mapping a type τ into an LF object T is defined by:

$$
\begin{array}{rcl}
\llbracket \mathbf{nat} \rrbracket^{\text{co}} & = & \mathbf{nat} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket^{\text{co}} & = & \mathbf{arrow}\ \llbracket \tau_1 \rrbracket^{\text{co}}\ \llbracket \tau_2 \rrbracket^{\text{co}} \\
\llbracket \tau_1 \times \tau_2 \rrbracket^{\text{co}} & = & \mathbf{product}\ \llbracket \tau_1 \rrbracket^{\text{co}}\ \llbracket \tau_2 \rrbracket^{\text{co}} \\
\llbracket [\Sigma]\tau \rrbracket^{\text{co}} & = & \mathbf{ccode}\ \llbracket \Sigma \rrbracket^{\text{co}}\ \llbracket \tau \rrbracket^{\text{co}}
\end{array}
$$

The function $\boxed{\llbracket \Sigma \rrbracket^{\text{co}}}$ mapping a contextual type list Σ into an LF object C is defined by:

$$
\begin{array}{rcl}
\llbracket \cdot \rrbracket^{\text{co}} & = & \mathbf{ctx\_0} \\
\llbracket \Sigma, \tau \rrbracket^{\text{co}} & = & \mathbf{ctx\_n}\ \llbracket \tau \rrbracket^{\text{co}}\ \llbracket \Sigma \rrbracket^{\text{co}}
\end{array}
$$

The function $\boxed{\llbracket \Delta \rrbracket^{\text{co}}}$ mapping a modal context Δ into an LF context Λ is defined by:

$$\begin{aligned}
\llbracket \cdot \rrbracket^{\text{co}} &= \cdot \\
\llbracket \Delta,\ u :: \tau[\Sigma] \rrbracket^{\text{co}} &= \llbracket \Delta \rrbracket^{\text{co}},\ u : \mathbf{uvar}\ \llbracket \Sigma \rrbracket^{\text{co}}\ \llbracket \tau \rrbracket^{\text{co}}
\end{aligned}$$

The function $\boxed{\llbracket \Gamma \rrbracket^{\text{co}}}$ mapping a non-modal context Γ into an LF context Λ is defined exactly as in the Mini-ML$^{\Box}_{\text{ex}}$ case:

$$\begin{aligned}
\llbracket \cdot \rrbracket^{\text{co}} &= \cdot,\ \mathbf{w} : \mathbf{world} \\
\llbracket \Gamma,\ x : \tau \rrbracket^{\text{co}} &= \llbracket \Gamma \rrbracket^{\text{co}},\ x : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\text{co}}
\end{aligned}$$

The function $\boxed{\llbracket \mathcal{Q} \rrbracket^{\text{co}}}$ mapping a typing derivation $\mathcal{Q}$ for a contextual expression list S into an LF object S such that $\llbracket \Delta \rrbracket^{\text{co}},\ \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} S : \mathbf{sequence}\ \mathbf{w}\ \llbracket \Sigma \rrbracket^{\text{co}}$ when $\mathcal{Q} :: \Delta \mid \Gamma \vdash^{\text{co,S}} S : \Sigma$ is defined by:

$$\left\llbracket \frac{}{\Delta \mid \Gamma \vdash^{\text{co,S}} \cdot : \cdot} \right\rrbracket^{\text{co}} = \mathbf{sequence\_0}\ \mathbf{w}$$

$$\left\llbracket \frac{\mathcal{Q} :: \Delta \mid \Gamma \vdash^{\text{co,S}} S : \Sigma \qquad \mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{co,S}} (S, e) : (\Sigma, \tau)} \right\rrbracket^{\text{co}} =$$

$$\mathbf{sequence\_n}\ \mathbf{w}\ \llbracket \Sigma \rrbracket^{\text{co}}\ \llbracket \tau \rrbracket^{\text{co}}\ \llbracket \mathcal{Q} \rrbracket^{\text{co}}\ \llbracket \mathcal{F} \rrbracket^{\text{co}}$$

The function $\boxed{\llbracket \mathcal{F} \rrbracket^{\text{co}}}$ mapping a typing derivation $\mathcal{F}$ for a contextual expression e into an LF object M such that $\llbracket \Delta \rrbracket^{\text{co}},\ \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} M : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau \rrbracket^{\text{co}}$ when $\mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \tau$ is defined by:

$$\left\llbracket \frac{\Gamma(x) = \tau}{\Delta \mid \Gamma \vdash^{\text{co}} x : \tau} \right\rrbracket^{\text{co}} = x$$

$$\left\llbracket \frac{\Delta(u) = \tau[\Sigma] \qquad \mathcal{Q} :: \Delta \mid \Gamma \vdash^{\text{co,S}} S : \Sigma}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{clo}\ (u, S) : \tau} \right\rrbracket^{\text{co}} = \mathbf{clo}\ \mathbf{w}\ \llbracket \Sigma \rrbracket^{\text{co}}\ \llbracket \tau \rrbracket^{\text{co}}\ u\ \llbracket \mathcal{Q} \rrbracket^{\text{co}}$$

$$\left\llbracket \frac{\mathcal{F} :: \Delta \mid \Gamma\{x \mapsto \tau_1\} \vdash^{\text{co}} e : \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \lambda x : \tau_1 \,.\, e : \tau_1 \to \tau_2} \right\rrbracket^{\text{co}} =$$

$$\mathbf{lam}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\text{co}}\ \llbracket \tau_2 \rrbracket^{\text{co}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\text{co}} \,.\, \llbracket \mathcal{F} \rrbracket^{\text{co}})$$

$$\left\llbracket \frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\text{co}} e_1 : \tau_1 \to \tau_2 \qquad \mathcal{F}_2 :: \Delta \mid \Gamma \vdash^{\text{co}} e_2 : \tau_1}{\Delta \mid \Gamma \vdash^{\text{co}} e_1\ e_2 : \tau_2} \right\rrbracket^{\text{co}} =$$

$$\mathbf{app}\ \mathbf{w}\ \llbracket \tau_1 \rrbracket^{\text{co}}\ \llbracket \tau_2 \rrbracket^{\text{co}}\ \llbracket \mathcal{F}_1 \rrbracket^{\text{co}}\ \llbracket \mathcal{F}_2 \rrbracket^{\text{co}}$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \Gamma\{x \mapsto \tau\} \vdash^{\text{co}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{fix}\, x : \tau\,.\, e : \tau} \right]\!\!\right^{\text{co}} = \mathbf{fix}\ \mathbf{w}\ [\![\tau]\!]^{\text{co}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}\ [\![\tau]\!]^{\text{co}}\,.\,[\![\mathcal{F}]\!]^{\text{co}})$$

$$\left[\!\!\left[\frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\text{co}} e_1 : \tau_1 \qquad \mathcal{F}_2 :: \Delta \mid \Gamma \vdash^{\text{co}} e_2 : \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \right]\!\!\right^{\text{co}} =$$

$$\mathbf{pair}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{co}}\ [\![\tau_2]\!]^{\text{co}}\ [\![\mathcal{F}_1]\!]^{\text{co}}\ [\![\mathcal{F}_2]\!]^{\text{co}}$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{fst}\ e : \tau_1} \right]\!\!\right^{\text{co}} = \mathbf{fst}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{co}}\ [\![\tau_2]\!]^{\text{co}}\ [\![\mathcal{F}]\!]^{\text{co}}$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \tau_1 \times \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{fst}\ e : \tau_2} \right]\!\!\right^{\text{co}} = \mathbf{snd}\ \mathbf{w}\ [\![\tau_1]\!]^{\text{co}}\ [\![\tau_2]\!]^{\text{co}}\ [\![\mathcal{F}]\!]^{\text{co}}$$

$$\left[\!\!\left[\frac{}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{z} : \mathbf{nat}} \right]\!\!\right^{\text{co}} = \mathbf{z}\ \mathbf{w}$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \mathbf{nat}}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{s}\ e : \mathbf{nat}} \right]\!\!\right^{\text{co}} = \mathbf{s}\ \mathbf{w}\ [\![\mathcal{F}]\!]^{\text{co}}$$

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \mathbf{nat} \quad \mathcal{F}' :: \Delta \mid \Gamma \vdash^{\text{co}} e_1 : \tau \quad \mathcal{F}'' :: \Delta \mid \Gamma\{x \mapsto \mathbf{nat}\} \vdash^{\text{co}} e_2 : \tau}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 : \tau} \right]\!\!\right^{\text{co}}$$

$$= \mathbf{case}\ \mathbf{w}\ [\![\tau]\!]^{\text{co}}\ [\![\mathcal{F}]\!]^{\text{co}}\ [\![\mathcal{F}']\!]^{\text{co}}\ (\lambda x : \mathbf{exp}\ \mathbf{w}\ \mathbf{nat}\,.\,[\![\mathcal{F}'']\!]^{\text{co}})$$

$$\left[\!\!\left[\frac{\mathcal{R} :: \Delta \mid \cdot \vdash^{\text{co,b}} b : [\Sigma]\tau}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{box}\ b : [\Sigma]\tau} \right]\!\!\right^{\text{co}} = \mathbf{box}\ \mathbf{w}\ [\![\Sigma]\!]^{\text{co}}\ [\![\tau]\!]^{\text{co}}\ (\lambda \mathbf{w} : \mathbf{world}\,.\,[\![\mathcal{R}]\!]^{\text{co}})$$

$$\left[\!\!\left[\frac{\mathcal{F}_1 :: \Delta \mid \Gamma \vdash^{\text{co}} e_1 : [\Sigma]\tau_1 \qquad \mathcal{F}_2 :: \Delta\{u \mapsto \tau_1[\Sigma]\} \mid \Gamma \vdash^{\text{co}} e_2 : \tau_2}{\Delta \mid \Gamma \vdash^{\text{co}} \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : \tau_2} \right]\!\!\right^{\text{co}} =$$

$$\mathbf{let\_box}\ \mathbf{w}\ [\![\Sigma]\!]^{\text{co}}\ [\![\tau_1]\!]^{\text{co}}\ [\![\tau_2]\!]^{\text{co}}\ [\![\mathcal{F}_1]\!]^{\text{co}}\ (\lambda u : \mathbf{uvar}\ [\![\Sigma]\!]^{\text{co}}\ [\![\tau_1]\!]^{\text{co}}\,.\,[\![\mathcal{F}_2]\!]^{\text{co}})$$

Finally the function $\boxed{[\![\mathcal{R}]\!]^{\text{co}}}$ mapping a typing derivation $\mathcal{R}$ for a contextual box-argument b into an LF object B such that $[\![\Delta]\!]^{\text{co}}$, $[\![\Gamma]\!]^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} B : \mathbf{boxarg}\ \mathbf{w}\ [\![\Sigma]\!]^{\text{co}}\ [\![\tau]\!]^{\text{co}}$ when $\mathcal{R} :: \Delta \mid \Gamma \vdash^{\text{co,b}} b : [\Sigma]\tau$ is defined by:

$$\left[\!\!\left[\frac{\mathcal{F} :: \Delta \mid \cdot \vdash^{\text{co}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{co,b}} e : [\cdot]\tau} \right]\!\!\right^{\text{co}} = \mathbf{boxarg\_0}\ \mathbf{w}\ [\![\tau]\!]^{\text{co}}\ [\![\mathcal{F}]\!]^{\text{co}}$$

$$\left[\!\!\left[\frac{\mathcal{R}' :: \Delta \mid \Gamma\{x \mapsto \tau'\} \vdash^{\text{co,b}} b : [\Sigma]\tau}{\Delta \mid \Gamma \vdash^{\text{co,b}} x : \tau'\,.\,b : [\Sigma, \tau']\tau} \right]\!\!\right^{\text{co}} =$$

$$\mathbf{boxarg\_n}\ \mathbf{w}\ [\![\Sigma]\!]^{\text{co}}\ [\![\tau']\!]^{\text{co}}\ [\![\tau]\!]^{\text{co}}\ [\![\mathcal{R}']\!]^{\text{co}}$$

The type and type list representations can be proved adequate by straightforward mutual induction on the structure of τ and Σ respectively. The rest of the adequacy properties are stated in more detail below.

Adequacy of the representation function $\llbracket \mathcal{Q} \rrbracket^{\text{co}}$ and $\llbracket \mathcal{F} \rrbracket^{\text{co}}$:

Theorem 6.1 (Expression list/Expression representation adequacy $\rightarrow$)**.** The following two statements are true:

a) If $\mathcal{Q} :: \Gamma \mid \Delta \vdash^{\text{co,S}} S : \Sigma$ then

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \mathcal{Q} \rrbracket^{\text{co}} : \mathbf{sequence\ w}\ \llbracket \Sigma \rrbracket^{\text{co}}.$$

b) If $\mathcal{F} :: \Gamma \mid \Delta \vdash^{\text{co}} e : \tau$ then

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \mathcal{F} \rrbracket^{\text{co}} : \mathbf{exp\ w}\ \llbracket \tau \rrbracket^{\text{co}}.$$

Proof. The proof is done by mutual induction on the structure of the derivations $\mathcal{Q}$ and $\mathcal{F}$. We only write down the proof cases for a) as the rest of the cases are very similar to cases of the proof of **Theorem 4.1**.

1. TP$^{\text{co,S}}$-0. In this case $\mathcal{Q}$ is of the form

$$\frac{}{\Delta \mid \Gamma \vdash^{\text{co,S}} \cdot : \cdot}.$$

We have that

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \mathbf{sequence\_0} : \mathbf{sequence\ w\ ctx\_0}$$

and as $\llbracket \mathcal{Q} \rrbracket^{\text{co}} = \mathbf{sequence\_0}$ and $\llbracket \cdot \rrbracket^{\text{co}} = \mathbf{ctx\_0}$, we are done.

2. TP$^{\text{co,S}}$-N. In this case $\mathcal{Q}$ is of the form

$$\frac{\mathcal{Q}' :: \Delta \mid \Gamma \vdash^{\text{co,S}} S : \Sigma \qquad \mathcal{F} :: \Delta \mid \Gamma \vdash^{\text{co}} e : \tau}{\Delta \mid \Gamma \vdash^{\text{co,S}} (S, e) : (\Sigma, \tau)}.$$

Applying the induction hypothesis of the proof of b) we get that

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \mathcal{F} \rrbracket^{\text{co}} : \mathbf{exp\ w}\ \llbracket \tau \rrbracket^{\text{co}}$$

and by application of the other induction hypothesis to $\mathcal{Q}'$ we get that

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \mathcal{Q}' \rrbracket^{\text{co}} : \mathbf{sequence\ w}\ \llbracket \Sigma \rrbracket^{\text{co}}.$$

As also

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \tau \rrbracket^{\text{co}} : \mathbf{tp}$$

and

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \llbracket \Sigma \rrbracket^{\text{co}} : \mathbf{ctx}$$

due to adequacy of type and type list representation, we can now apply the TP$^{\text{LF}}$-TERM-APP rule of the logical framework five times and get that

$$\llbracket \Delta \rrbracket^{\text{co}}, \llbracket \Gamma \rrbracket^{\text{co}} \vdash^{\text{LF}}_{\Sigma_{\text{co}}} \mathbf{sequence\_n\ w}\ \llbracket \Sigma \rrbracket^{\text{co}}\ \llbracket \tau \rrbracket^{\text{co}}\ \llbracket \mathcal{F} \rrbracket^{\text{co}}\ \llbracket \mathcal{Q}' \rrbracket^{\text{co}} :$$

$$\textbf{sequence w } (\textbf{ctx\_n } \llbracket\tau\rrbracket^{\mathrm{co}} \ \llbracket\Sigma\rrbracket^{\mathrm{co}}).$$

As $\llbracket\Sigma, \tau\rrbracket^{\mathrm{co}} = \textbf{ctx\_n } \llbracket\tau\rrbracket^{\mathrm{co}} \ \llbracket\Sigma\rrbracket^{\mathrm{co}}$ and $\llbracket\mathcal{Q}\rrbracket^{\mathrm{co}} = \textbf{sequence\_n w } \llbracket\Sigma\rrbracket^{\mathrm{co}} \ \llbracket\tau\rrbracket^{\mathrm{co}} \ \llbracket\mathcal{F}\rrbracket^{\mathrm{co}} \ \llbracket\mathcal{Q}'\rrbracket^{\mathrm{co}}$, we are done. □

Theorem 6.2 (Expression list/Expression representation adequacy ←)**.** The following two statements are true:

a) If S is a canonical LF object with

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} S \ : \ \textbf{sequence w } \llbracket\Sigma\rrbracket^{\mathrm{co}}.$$

then we can find a typing derivation $\mathcal{Q} :: \Gamma \mid \Delta \vdash^{\mathrm{co,S}} S \ : \ \Sigma$ such that $\llbracket\mathcal{Q}\rrbracket^{\mathrm{co}} = S$.

b) If M is a canonical LF object with

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} M \ : \ \textbf{exp w } \llbracket\tau\rrbracket^{\mathrm{co}}.$$

then we can find a typing derivation $\mathcal{F} :: \Gamma \mid \Delta \vdash^{\mathrm{co}} e \ : \ \tau$ such that $\llbracket\mathcal{F}\rrbracket^{\mathrm{co}} = M$.

Proof. The theorem can be proved by mutual induction on the structure of the canonical object S and M. As the proof is very similar to what we have seen earlier, it is left out. □

Adequacy of the representation function $\llbracket\mathcal{R}\rrbracket^{\mathrm{co}}$:

Theorem 6.3 (Box-argument representation adequacy →)**.** If $\mathcal{R} :: \Gamma \mid \Delta \vdash^{\mathrm{co,b}} b \ : \ [\Sigma]\tau$ then

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} \llbracket\mathcal{R}\rrbracket^{\mathrm{co}} \ : \ \textbf{boxarg w } \llbracket\Sigma\rrbracket^{\mathrm{co}} \ \llbracket\tau\rrbracket^{\mathrm{co}}.$$

Proof. The proof is done by induction on the derivation $\mathcal{R}$. Below, each of the two typing rules is in turn considered the last rule applied in the derivation.

1. TP$^{\mathrm{co,b}}$-0. In this case $\mathcal{R}$ is of the form

$$\frac{\mathcal{F} :: \Delta \mid \cdot \vdash^{\mathrm{co}} e \ : \ \tau}{\Delta \mid \Gamma \vdash^{\mathrm{co,b}} e \ : \ [\cdot]\tau}.$$

From **Theorem 6.1** b) we get that

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} \llbracket\mathcal{F}\rrbracket^{\mathrm{co}} \ : \ \textbf{exp w } \llbracket\tau\rrbracket^{\mathrm{co}}$$

and as $(\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}})(\mathbf{w}) = \textbf{world}$ and as also

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} \llbracket\tau\rrbracket^{\mathrm{co}} \ : \ \textbf{tp}$$

due to adequacy of the type representation, we can apply the TP$^{\mathrm{LF}}$-TERM-APP rule of the logical framework three times and get that

$$\llbracket\Delta\rrbracket^{\mathrm{co}}, \ \llbracket\Gamma\rrbracket^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} \textbf{boxarg\_0 w } \llbracket\tau\rrbracket^{\mathrm{co}} \ \llbracket\mathcal{F}\rrbracket^{\mathrm{co}} \ : \ \textbf{boxarg w ctx\_0 } \llbracket\tau\rrbracket^{\mathrm{co}}.$$

As $[\![\cdot]\!]^{\mathrm{co}} = \mathbf{ctx\_0}$ and $[\![\mathcal{R}]\!]^{\mathrm{co}} = \mathbf{boxarg\_0\ w}\ [\![\tau]\!]^{\mathrm{co}}\ [\![\mathcal{F}]\!]^{\mathrm{co}}$, we are done.

2. TP$^{\mathrm{co},\,\mathrm{b}}$-N. In this case $\mathcal{R}$ is of the form

$$\frac{\mathcal{R}' :: \Delta \mid \Gamma\{x \mapsto \tau'\} \vdash^{\mathrm{co},\,\mathrm{b}} b : [\Sigma]\tau}{\Delta \mid \Gamma \vdash^{\mathrm{co},\,\mathrm{b}} x : \tau' . b : [\Sigma, \tau']\tau}\,.$$

and we can apply the induction hypothesis to $\mathcal{R}'$ and get that

$$[\![\Delta]\!]^{\mathrm{co}},\ [\![\Gamma\{x \mapsto \tau'\}]\!]^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} [\![\mathcal{R}']\!]^{\mathrm{co}} : \mathbf{boxarg\ w}\ [\![\Sigma]\!]^{\mathrm{co}}\ [\![\tau]\!]^{\mathrm{co}}\,.$$

By in turn considering the cases where x is and is not already present in Γ and exploiting a suitable variant of **Lemma 3.6** as well as the weakening property described in **Theorem 2.2** we reach that

$$[\![\Delta]\!]^{\mathrm{co}},\ [\![\Gamma]\!]^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} \mathbf{boxarg\_n\ w}\ [\![\Sigma]\!]^{\mathrm{co}}\ [\![\tau']\!]^{\mathrm{co}}\ [\![\tau]\!]^{\mathrm{co}}\ [\![\mathcal{R}']\!]^{\mathrm{co}} : \mathbf{boxarg\ w}\ (\mathbf{ctx\_n}\ [\![\tau']\!]^{\mathrm{co}}\ [\![\Sigma]\!]^{\mathrm{co}})\ [\![\tau]\!]^{\mathrm{co}}.$$

As $[\![\Sigma, \tau']\!]^{\mathrm{co}} = \mathbf{ctx\_n}\ [\![\tau']\!]^{\mathrm{co}}\ [\![\Sigma]\!]^{\mathrm{co}}$ and $[\![\mathcal{R}]\!]^{\mathrm{co}} = \mathbf{boxarg\_n\ w}\ [\![\Sigma]\!]^{\mathrm{co}}\ [\![\tau']\!]^{\mathrm{co}}\ [\![\tau]\!]^{\mathrm{co}}\ [\![\mathcal{R}']\!]^{\mathrm{co}}$, we are done. □

Theorem 6.4 (Box-argument representation adequacy ←)**.** If B is a canonical LF object with

$$[\![\Delta]\!]^{\mathrm{co}},\ [\![\Gamma]\!]^{\mathrm{co}} \vdash^{\mathrm{LF}}_{\Sigma_{\mathrm{co}}} B : \mathbf{boxarg\ w}\ [\![\Sigma]\!]^{\mathrm{co}}\ [\![\tau]\!]^{\mathrm{co}}\,.$$

then we can find a typing derivation $\mathcal{R} :: \Gamma \mid \Delta \vdash^{\mathrm{co},\,\mathrm{b}} b : [\Sigma]\tau$ such that $[\![\mathcal{R}]\!]^{\mathrm{co}} = B$.

Proof. The theorem can be proved by induction on the structure of the canonical object B. As the proof is very similar to what we have seen earlier, it is left out. □

6.3 Operational semantics for Mini-ML$_{\mathrm{co}}$

Nanevski *et al.* do not provide any concrete operational semantics for Mini-ML$_{\mathrm{co}}$. They only define relevant substitutions and point out that Mini-ML$_{\mathrm{co}}$ is a straightforward call-by-value functional language. Below we will define both big-step and small-step operational semantics for Mini-ML$_{\mathrm{co}}$ and prove them equivalent. By definition of big-step operational semantics it is substantiated that Mini-ML$_{\mathrm{co}}$ generalizes Mini-ML$^{\Box}_{\mathrm{ex}}$. The small-step operational semantics will be used in proving type soundness.

Adapting the substitutions defined by Nanevski *et al.* to our syntax we get the following definition of the function $\boxed{^{\mathrm{b}}\{b/u\}e}$ substituting the box-argument b for the modal variable u in the expression e:

$$
\begin{array}{rcl}
{}^{\mathrm{b}}\{b/u\}x & = & x \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{clo}\ (u, S)) & = & {}^{S}\Big\{{}^{\mathrm{b,S}}\{b/u\}S\Big\}\, b \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{clo}\ (u', S)) & = & \mathbf{clo}\ (u', {}^{\mathrm{b,S}}\{b/u\}S), \text{ when } u \neq u' \\
{}^{\mathrm{b}}\{b/u\}(\lambda x : \tau\,.\ e) & = & \lambda x : \tau\,.\ ({}^{\mathrm{b}}\{b/u\}e) \\
{}^{\mathrm{b}}\{b/u\}(e_1\ e_2) & = & ({}^{\mathrm{b}}\{b/u\}e_1)\ ({}^{\mathrm{b}}\{b/u\}e_2) \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{fix}\ x : \tau\,.\ e) & = & \mathbf{fix}\ x : \tau\,.\ ({}^{\mathrm{b}}\{b/u\}e) \\
{}^{\mathrm{b}}\{b/u\}(\langle e_1, e_2\rangle) & = & \langle {}^{\mathrm{b}}\{b/u\}e_1, {}^{\mathrm{b}}\{b/u\}e_2\rangle \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{fst}\ e) & = & \mathbf{fst}\ ({}^{\mathrm{b}}\{b/u\}e) \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{snd}\ e) & = & \mathbf{snd}\ ({}^{\mathrm{b}}\{b/u\}e) \\
{}^{\mathrm{b}}\{b/u\}\mathbf{z} & = & \mathbf{z} \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{s}\ e) & = & \mathbf{s}\ ({}^{\mathrm{b}}\{b/u\}e) \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2) & = & \mathbf{case}\ ({}^{\mathrm{b}}\{b/u\}e)\ \mathbf{of}\ \mathbf{z} \Rightarrow ({}^{\mathrm{b}}\{b/u\}e_1) \\
 & & \qquad \mid\ \mathbf{s}\ x \Rightarrow ({}^{\mathrm{b}}\{b/u\}e_2) \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{box}\ b') & = & \mathbf{box}\ ({}^{\mathrm{b,b}}\{b/u\}b') \\
{}^{\mathrm{b}}\{b/u\}(\mathbf{let\ box}\ u' = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let\ box}\ u' = {}^{\mathrm{b}}\{b/u\}e_1\ \mathbf{in}\ {}^{\mathrm{b}}\{b/u\}e_2, \\
 & & \qquad \text{when } u \neq u' \text{ and } u' \notin \mathrm{FV}(b)
\end{array}
$$

where $\boxed{{}^{\mathrm{b,S}}\{b/u\}S}$ substituting the box-argument b for the modal variable u in the expression list S is defined by

$$
\begin{array}{rcl}
{}^{\mathrm{b,S}}\{b/u\}\cdot & = & \cdot \\
{}^{\mathrm{b,S}}\{b/u\}(S, e) & = & {}^{\mathrm{b,S}}\{b/u\}S, {}^{\mathrm{b}}\{b/u\}e
\end{array}
$$

and $\boxed{{}^{\mathrm{b,b}}\{b/u\}b'}$ substituting the box-argument b for the modal variable u in the box-argument b' is defined by

$$
\begin{array}{rcl}
{}^{\mathrm{b,b}}\{b/u\}e & = & {}^{\mathrm{b}}\{b/u\}e \\
{}^{\mathrm{b,b}}\{b/u\}(x : \tau\,.\ b') & = & x : \tau\,.\ ({}^{\mathrm{b,b}}\{b/u\}b')
\end{array}
$$

and finally where $\boxed{{}^{S}\{S\}\, b}$ substituting the expressions of the list S into the variables bound by the box-argument b is defined by

$$
\begin{aligned}
{}^{S}\{\cdot\}\, e &= e \\
{}^{S}\{S,\, e\}\, (x : \tau \,.\, b) &= {}^{S}\{S\}\, (\{e/x\}\, b)
\end{aligned}
$$

Before proceeding to the operational semantics we also need to define the values among the expressions in Mini-ML$_{\text{co}}$. These are defined by the following sub-grammar:

$$\text{Values:} \quad v \quad ::= \quad \lambda x : \tau \,.\, e \mid \langle v_1,\, v_2 \rangle \mid \mathbf{z} \mid \mathbf{s}\ v \mid \mathbf{box}\ b$$

Big-step evaluation

The big-step evaluation relation $\boxed{e \hookrightarrow^{\text{co}} v}$ evaluating a contextual expression e to a contextual value v is defined by the following inference rules:

$$\frac{}{\lambda x : \tau \,.\, e \hookrightarrow^{\text{co}} \lambda x : \tau \,.\, e}\ \text{EVAL}^{\text{co}}\text{-ABS}$$

$$\frac{e_1 \hookrightarrow^{\text{co}} \lambda x : \tau \,.\, e_1' \qquad e_2 \hookrightarrow^{\text{co}} v_2 \qquad \{v_2/x\}\, e_1' \hookrightarrow^{\text{co}} v}{e_1\ e_2 \hookrightarrow^{\text{co}} v}\ \text{EVAL}^{\text{co}}\text{-APP}$$

$$\frac{\{\mathbf{fix}\ x : \tau \,.\, e/x\}\, e \hookrightarrow^{\text{co}} v}{\mathbf{fix}\ x : \tau \,.\, e \hookrightarrow^{\text{co}} v}\ \text{EVAL}^{\text{co}}\text{-FIX}$$

$$\frac{e_1 \hookrightarrow^{\text{co}} v_1 \qquad e_2 \hookrightarrow^{\text{co}} v_2}{\langle e_1,\, e_2 \rangle \hookrightarrow^{\text{co}} \langle v_1,\, v_2 \rangle}\ \text{EVAL}^{\text{co}}\text{-PAIR}$$

$$\frac{e \hookrightarrow^{\text{co}} \langle v_1,\, v_2 \rangle}{\mathbf{fst}\ e \hookrightarrow^{\text{co}} v_1}\ \text{EVAL}^{\text{co}}\text{-FST} \qquad \frac{e \hookrightarrow^{\text{co}} \langle v_1,\, v_2 \rangle}{\mathbf{snd}\ e \hookrightarrow^{\text{co}} v_2}\ \text{EVAL}^{\text{co}}\text{-SND}$$

$$\frac{}{\mathbf{z} \hookrightarrow^{\text{co}} \mathbf{z}}\ \text{EVAL}^{\text{co}}\text{-ZERO} \qquad \frac{e \hookrightarrow^{\text{co}} v}{\mathbf{s}\ e \hookrightarrow^{\text{co}} \mathbf{s}\ v}\ \text{EVAL}^{\text{co}}\text{-SUCC}$$

$$\frac{e_1 \hookrightarrow^{\text{co}} \mathbf{z} \qquad e_2 \hookrightarrow^{\text{co}} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\text{co}} v}\ \text{EVAL}^{\text{co}}\text{-CASE-Z}$$

$$\frac{e_1 \hookrightarrow^{\text{co}} \mathbf{s}\ v_1 \qquad \{v_1/x\}\, e_3 \hookrightarrow^{\text{co}} v}{\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 \hookrightarrow^{\text{co}} v}\ \text{EVAL}^{\text{co}}\text{-CASE-S}$$

$$\frac{}{\mathbf{box}\ b \hookrightarrow^{\text{co}} \mathbf{box}\ b}\ \text{EVAL}^{\text{co}}\text{-BOX}$$

$$\frac{e_1 \hookrightarrow^{\text{co}} \mathbf{box}\ b \qquad {}^{\text{b}}\{b/u\} e_2 \hookrightarrow^{\text{co}} v}{\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \hookrightarrow^{\text{co}} v}\ \text{EVAL}^{\text{co}}\text{-LET-BOX}$$

These big-step evaluation rules look exactly the same as the rules we saw for Mini-ML$^{\Box}_{\text{ex}}$ except from the box-arguments in TP$^{\text{co}}$-BOX and TP$^{\text{co}}$-LET-BOX which are just expressions in the rules TP$^{\text{ex}}$-BOX and TP$^{\text{ex}}$-LET-BOX.

Having the big-step operational semantics in place it is now appropriate to consider the contextual version of the explicit program *power*$^{\text{ex}}$:

$$
\begin{array}{lcl}
\mathit{power}^{\text{co}} & \equiv & \mathbf{fix}\ p : \mathbf{nat} \to [\cdot, \mathbf{nat}](\mathbf{nat} \to \mathbf{nat})\,. \\
& & \quad \lambda n : \mathbf{nat}\,. \\
& & \quad\quad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \quad\ \Rightarrow \mathbf{box}\ (x : \mathbf{nat}\,.\ \mathbf{s}\ \mathbf{z}) \\
& & \quad\quad\quad\ \mid\ \mathbf{s}\ m \Rightarrow \mathbf{let}\ \mathbf{box}\ q = p\ m \\
& & \quad\quad\quad\quad\quad\ \mathbf{in}\ \ \mathbf{box}\ (x : \mathbf{nat}\,.\ \mathit{times}\ x\ (\mathbf{clo}\ (q, (\cdot, x))))
\end{array}
$$

For $n = 2$ we can derive the following evaluation due to the operational semantics given above:

$$
\mathit{power}^{\text{co}}\ (\mathbf{s}\ (\mathbf{s}\ \mathbf{z})) \hookrightarrow^{\text{co}} \mathbf{box}\ (x : \mathbf{nat}\,.\ \mathit{time}\ x\ (\mathit{times}\ x\ (\mathbf{s}\ \mathbf{z})))
$$

We see that the superfluous β-redexes which occurred when *power*$^{\text{ex}}$ was applied to 2 are no longer an issue here. They are prevented due to substitutions performed on evaluation of closure-constructions.

Small-step evaluation

As already mentioned in Section **4.1.6**, we find it relevant to verify whether the defined type system is strong enough to ensure that no well-typed expression gets stuck during evaluation. For this purpose we find it most convenient to transform the big-step evaluation rules given above into a list of equivalent small-step evaluation rules and then use the soundness proof approach presented by Wright and Felleisen in [13].

We define the small-step evaluation relation $\boxed{e \mapsto^{\text{co}} e'}$ by the following inference rules:

$$
\frac{}{(\lambda x : \tau\,.\ e_1)\ v_2 \mapsto^{\text{co}} \{v_2/x\}\, e_1}\ \text{STEP}^{\text{co}}\text{-APP-1}
$$

$$
\frac{e_2 \mapsto^{\text{co}} e_2'}{e_1\ e_2 \mapsto^{\text{co}} e_1\ e_2'}\ \text{STEP}^{\text{co}}\text{-APP-2} \qquad \frac{e_1 \mapsto^{\text{co}} e_1'}{e_1\ e_2 \mapsto^{\text{co}} e_1'\ e_2}\ \text{STEP}^{\text{co}}\text{-APP-3}
$$

$$
\frac{}{\mathbf{fix}\ x : \tau\,.\ e \mapsto^{\text{co}} \{\mathbf{fix}\ x : \tau\,.\ e/x\}\, e}\ \text{STEP}^{\text{co}}\text{-FIX}
$$

$$\frac{e_2 \;\mapsto^{\text{co}}\; e_2'}{\langle v_1,\, e_2\rangle \;\mapsto^{\text{co}}\; \langle v_1,\, e_2'\rangle}\ \text{STEP}^{\text{co}}\text{-PAIR-1} \qquad \frac{e_1 \;\mapsto^{\text{co}}\; e_1'}{\langle e_1,\, e_2\rangle \;\mapsto^{\text{co}}\; \langle e_1',\, e_2\rangle}\ \text{STEP}^{\text{co}}\text{-PAIR-2}$$

$$\frac{}{\mathbf{fst}\ \langle v_1,\, v_2\rangle \;\mapsto^{\text{co}}\; v_1}\ \text{STEP}^{\text{co}}\text{-FST-1} \qquad \frac{e \;\mapsto^{\text{co}}\; e'}{\mathbf{fst}\ e \;\mapsto^{\text{co}}\; \mathbf{fst}\ e'}\ \text{STEP}^{\text{co}}\text{-FST-2}$$

$$\frac{}{\mathbf{snd}\ \langle v_1,\, v_2\rangle \;\mapsto^{\text{co}}\; v_2}\ \text{STEP}^{\text{co}}\text{-SND-1} \qquad \frac{e \;\mapsto^{\text{co}}\; e'}{\mathbf{snd}\ e \;\mapsto^{\text{co}}\; \mathbf{snd}\ e'}\ \text{STEP}^{\text{co}}\text{-SND-2}$$

$$\frac{e \;\mapsto^{\text{co}}\; e'}{\mathbf{s}\ e \;\mapsto^{\text{co}}\; \mathbf{s}\ e'}\ \text{STEP}^{\text{co}}\text{-SUCC}$$

$$\frac{}{\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \;\mapsto^{\text{co}}\; e_1}\ \text{STEP}^{\text{co}}\text{-CASE-1}$$

$$\frac{}{\mathbf{case}\ (\mathbf{s}\ v)\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \;\mapsto^{\text{co}}\; \{v/x\}\, e_2}\ \text{STEP}^{\text{co}}\text{-CASE-2}$$

$$\frac{e \;\mapsto^{\text{co}}\; e'}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \;\mapsto^{\text{co}}\; \mathbf{case}\ e'\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2}\ \text{STEP}^{\text{co}}\text{-CASE-3}$$

$$\frac{}{\mathbf{let\ box}\ u = (\mathbf{box}\ b)\ \mathbf{in}\ e \;\mapsto^{\text{co}}\; {}^{\text{b}}\{b/u\}e}\ \text{STEP}^{\text{co}}\text{-LET-BOX-1}$$

$$\frac{e_1 \;\mapsto^{\text{co}}\; e_1'}{\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \;\mapsto^{\text{co}}\; \mathbf{let\ box}\ u = e_1'\ \mathbf{in}\ e_2}\ \text{STEP}^{\text{co}}\text{-LET-BOX-2}$$

The multi-step evaluation relation $\boxed{e \;\mapsto^{*\,\text{co}}\; e'}$ is defined as the reflexive-transitive closure of the small-step evaluation relation $e \;\mapsto^{\text{co}}\; e'$:

$$\frac{}{e \;\mapsto^{*\,\text{co}}\; e}\ \text{STEPS}^{\text{co}}\text{-0} \qquad \frac{e \;\mapsto^{\text{co}}\; e' \qquad e' \;\mapsto^{*\,\text{co}}\; e''}{e \;\mapsto^{*\,\text{co}}\; e''}\ \text{STEPS}^{\text{co}}\text{-M}$$

That is $e \;\mapsto^{*\,\text{co}}\; e'$ iff e evaluates to e' in zero, one or several steps.

Equivalence of big- and small-step evaluation

To prove equivalence of the big- and small-step evaluation relation we need the following lemmas:

Lemma 6.1. If $e \mapsto^{\text{co}} e'$ and $e' \hookrightarrow^{\text{co}} v$ then $e \hookrightarrow^{\text{co}} v$.

Proof. The lemma can be proved by straightforward induction on the structure of the derivation of $e \mapsto^{\text{co}} e'$. □

Lemma 6.2. If $e \mapsto^{*\text{co}} e'$ and $e' \hookrightarrow^{\text{co}} v$ then $e \hookrightarrow^{\text{co}} v$.

Proof. The lemma can be proved by induction on the structure of the derivation of $e \mapsto^{*\text{co}} e'$ using **Lemma 6.1** in the STEPS$^{\text{co}}$-M case. □

Lemma 6.3. For any value v we have that $v \hookrightarrow^{\text{co}} v$.

Proof. The lemma follows by straightforward induction on the structure of the value v. □

Lemma 6.4. If $e \mapsto^{*\text{co}} e'$ and $e' \mapsto^{*\text{co}} e''$ then $e \mapsto^{*\text{co}} e''$.

Proof. The lemma can be proved by straightforward induction on the structure of the derivation of $e \mapsto^{*\text{co}} e'$. □

Lemma 6.5. If $e \mapsto^{*\text{co}} e'$ then also

- $e_1\ e \mapsto^{*\text{co}} e_1\ e'$
- $e\ e_2 \mapsto^{*\text{co}} e'\ e_2$
- $\langle v_1, e\rangle \mapsto^{*\text{co}} \langle v_1, e'\rangle$
- $\langle e, e_2\rangle \mapsto^{*\text{co}} \langle e', e_2\rangle$
- $\mathbf{fst}\ e \mapsto^{*\text{co}} \mathbf{fst}\ e'$
- $\mathbf{snd}\ e \mapsto^{*\text{co}} \mathbf{snd}\ e'$
- $\mathbf{s}\ e \mapsto^{*\text{co}} \mathbf{s}\ e'$
- $\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \mapsto^{*\text{co}} \mathbf{case}\ e'\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2$
- $\mathbf{let\ box}\ u = e\ \mathbf{in}\ e_2 \mapsto^{*\text{co}} \mathbf{let\ box}\ u = e'\ \mathbf{in}\ e_2$

Proof. The lemma can be proved by straightforward induction on the structure of the derivation of $e \mapsto^{*\text{co}} e'$. □

We are now ready to prove the following equivalence theorem:

Theorem 6.5 (Equivalence of small-steps and big-step)**.** We have that $e \mapsto^{*\text{co}} v$ if and only if $e \hookrightarrow^{\text{co}} v$.

Proof.

⇒: This case follows immediately from **Lemma 6.2** and **Lemma 6.3**.

⇐: This case can be proved by induction on the structure of the derivation of $e \hookrightarrow^{\text{co}} v$ and appropriate use of **Lemma 6.4** and **Lemma 6.5**.

□

6.3.1 Representing Mini-ML$_{\text{co}}$ operational semantics in LF

The substitutions defined above can all be represented straightforwardly within LF. In the Twelf code in Section **A.6** the type families concerning these functions are `subst_boxarg`, `subst_boxarg_stack`, `subst_boxarg_boxarg`, and `eval_clo` respectively.

Similar to how we approached the representation of operational semantics for both Mini-ML$^{\Box}_{\text{ex}}$ and Mini-ML$^{\bigcirc}$ the above sub-grammar for contextual values is translated into a value judgement before representation. The inference rules defining the value judgement $\boxed{\text{value}^{\text{co}}(v)}$ are all the same as in the explicit case except from the box-rule:

$$\frac{}{\text{value}^{\text{co}}(\textbf{box}\ b)}\ \text{VAL}^{\text{co}}\text{-BOX}$$

Again, with this value judgement, big-step evaluation becomes a mapping from expressions into other expressions, and then the later can be explicitly proved to be a values. The Twelf type families in Section **A.6** concerning the value judgement, big-step evaluation and value soundness for Mini-ML$_{\text{co}}$ are `val_c`, `eval_c`, and `val_sound_c` respectively.

The verification of the equivalence between big- and small-step evaluation within Twelf is also straightforward. In Section **A.6** the type families `cat_step_eval_c`, `cat_steps_eval_c`, and `eval_val_c` correspond to **Lemma 6.1**, **Lemma 6.2**, and **Lemma 6.3** respectively and the $\Rightarrow$-part of **Theorem 6.5** is represented by use of `small_to_big_c`. The type family `cat_steps_c` corresponds to **Lemma 6.4**, the type families whose name begins with `inject_steps_c_` corresponds to **Lemma 6.5**, and finally the $\Leftarrow$-part of **Theorem 6.5** is represented by use of `big_to_small_c`.

6.3.2 Metatheory about Mini-ML$_{\text{co}}$

Just as in the case of Mini-ML$^{\Box}_{\text{ex}}$ we can formulate a substitution lemma for the contextual language. Also, the big-step operational semantics defined for Mini-ML$_{\text{co}}$ can be proved to be determinant and type preserving. We will not state these properties explicitly here.

Type soundness

The properties just mentioned do not ensure that we are not able to find some well-typed explicit expression, which will get stuck during evaluation and thus will not reach a value. We find it most relevant to prove that the type-system of Mini-ML$_{\text{co}}$ is actually strong enough to prevent such incidents. That will also be a nice opportunity to have the capability of our LF representation more thoroughly tested.

Soundness of the Mini-ML$_{\text{co}}$ system is verified by proving that the small-step evaluation relation is type preserving and that the progress property is fulfilled:

Theorem 6.6 (Type preservation)**.** If we have that $\Delta \mid \Gamma \vdash^{\text{co}} e : \tau$ and that $e \mapsto^{\text{co}} e'$ then also $\Delta \mid \Gamma \vdash^{\text{co}} e' : \tau$.

Proof. The proof can be done by induction on the structure of the derivation of $\Delta \mid \Gamma \vdash^{\text{co}} e : \tau$. □

Theorem 6.7 (Progress)**.** If we have that $\Delta \mid \Gamma \vdash^{\text{co}} e : \tau$ then at least one of the following two cases is true:

a) The expression e is a value.

b) We can find an expression e' such that $e \mapsto^{\text{co}} e'$.

Proof. The proof can be done by induction on the structure of the derivation of $\Delta \mid \Gamma \vdash^{\text{co}} e : \tau$. □

Mini-ML$^{\Box}_{\text{ex}}$ contained in Mini-ML$_{\text{co}}$

To make the fact that Mini-ML$^{\Box}_{\text{ex}}$ corresponds to the subset of Mini-ML$_{\text{co}}$ where Σ is constrained to $\cdot$, we formulate a type preserving translation from Mini-ML$^{\Box}_{\text{ex}}$ to Mini-ML$_{\text{co}}$. For this we use $\bar{\tau}$, $\bar{e}$, $\bar{\Delta}$, and $\bar{\Gamma}$ to range over contextual types, expressions, modal contexts, and non-modal contexts respectively.

The relation $\boxed{\tau \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\tau}}$ mapping a Mini-ML$^{\Box}_{\text{ex}}$ type τ into a Mini-ML$_{\text{co}}$ type $\bar{\tau}$ is trivially defined for the standard types, whereas the rule for code types looks like:

$$\frac{\tau \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\tau}}{\Box\tau \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} [\cdot]\bar{\tau}} \ \text{TR}^{\text{ex,co}}\text{-TP-CODE}$$

The relation $\boxed{\Gamma \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\Gamma}}$ mapping a Mini-ML$^{\Box}_{\text{ex}}$ non-modal context Γ into a Mini-ML$_{\text{co}}$ non-modal context $\bar{\Gamma}$ is trivially defined using the defined type translation. The relation $\boxed{\Delta \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\Delta}}$ mapping a Mini-ML$^{\Box}_{\text{ex}}$ modal context Δ into a Mini-ML$_{\text{co}}$ modal context $\bar{\Delta}$ is defined by the following two rules:

$$\frac{}{\cdot \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \cdot} \ \text{TR}^{\text{ex,co}}\text{-MCTX-0} \qquad \frac{\Delta \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\Delta} \qquad \tau \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\tau}}{(\Delta,\ u :: \tau) \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} (\bar{\Delta},\ u :: \bar{\tau}[\cdot])} \ \text{TR}^{\text{ex,co}}\text{-MCTX-N}$$

Finally we write down the inference rules concerning modal variables and box constructions for the relation $\boxed{e \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{e}}$ mapping a Mini-ML$^{\Box}_{\text{ex}}$ expression e into a Mini-ML$_{\text{co}}$ expression $\bar{e}$:

$$\frac{}{u \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \mathbf{clo}\ (u, \cdot)} \ \text{TR}^{\text{ex,co}}\text{-VAR-U} \qquad \frac{e \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{e}}{\mathbf{box}\ e \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \mathbf{box}\ \bar{e}} \ \text{TR}^{\text{ex,co}}\text{-BOX}$$

Here the $\bar{e}$ appearing in TR$^{\text{ex,co}}$-BOX is in fact a box-argument with an empty preceding list of variable assumptions.

Now we are ready to formulate the type preservation theorem:

Theorem 6.8. If we have that $\Delta \mid \Gamma \vdash^{\text{ex}} e : \tau$ then we can find $\bar{\Delta}$, $\bar{\Gamma}$, $\bar{\tau}$, and $\bar{e}$ such that $\Delta \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\Delta}$, $\Gamma \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\Gamma}$, $e \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{e}$, and $\tau \overset{\text{ex}}{\rightsquigarrow}{}^{\text{co}} \bar{\tau}$ and such that $\bar{\Delta} \mid \bar{\Gamma} \vdash^{\text{co}} \bar{e} : \bar{\tau}$.

Proof. The theorem can be proved by induction on the structure of the derivation of $\Delta \mid \Gamma \vdash^{\text{ex}} e : \tau$. □

6.3.3 Representing the Mini-ML$_{\mathrm{co}}$ metatheory in LF

As we use intrinsically typed representation syntax we get the formalization of substitution and type preservation properties for free.

The representation and verification of big-step evaluation determinacy proceeds just like in the Mini-ML$^{\Box}_{\mathrm{ex}}$-case. The corresponding Twelf type family in Section **A.6** is the one named `eval_det_c`.

Concerning verification of the progress theorem, **Theorem 6.7**, we define a type family `not_stuck_c` comprising the characteristics of an expression not being stuck, and then we prove that an object of not-stuck-type can be established for any well-typed expression. This takes lemmas about how for instance **s** e will not be stuck if e is not. These lemmas correspond to Twelf type families whose name is prefixed `not_stuck_c_` and the final progress proof is represented by use of the type family `progress_c`.

The Twelf-verification of **Theorem 6.8** does not involve any problems. The key type families are `tr_tp_e~>c`, `tr_tp_e~>c_total`, `tr_tp_e~>c_unq`, and `tr_e~>c`.

Chapter 7

Conclusion

In the previous chapters we succeeded in adequately representing three metaprogramming systems based on different modal logics within the logical framework LF, a framework normally used for the representation of single-stage systems. We showed that the syntax representations did not prevent representation of operational semantics or Twelf verifications of relevant metatheories. More specifically:

- First we verified that the ordinary single-stage Mini-ML could in fact be adequately represented when implicit α-renaming was replaced by an explicit context update methodology.
- Then we proceeded to the first of the two languages based on modal logic of necessity, the explicit language Mini-ML$^{\Box}_{\text{ex}}$, and proved how an adequate LF representation could be achieved by use of standard higher-order abstract syntax and intrinsic encoding of world-scopes. We also had the big-step evaluation relation of Mini-ML$^{\Box}_{\text{ex}}$ represented in LF, and the machine-verification of its determinacy within Twelf went smoothly. A similar representation strategy turned out to work for the implicit language Mini-ML$^{\Box}$ as well, although we in that case had to also appropriately equip representation functions with a stage counter to distinct cross-stage variables.
- For the type preserving translation of Mini-ML$^{\Box}$ expressions into Mini-ML$^{\Box}_{\text{ex}}$ expressions we reformulated the method described by Davies and Pfenning in [2] into another type preserving method where unbox-sub-terms were replaced by numbered labels during translation. By postponing the insertion of modal variables until later we manged to have the translation represented and machine-verified within Twelf.
- Next we managed to have the temporal logically based metaprogramming language Mini-ML$^{\bigcirc}$ adequately represented using intrinsic world numbers and higher-order-abstract syntax and also for this language we represented the big-step operational semantics and had determinacy Twelf-verified.
- Last we succeeded in applying our representation methodology to the language based on modal logic of contextual necessity, Mini-ML$_{\text{co}}$. By formulation of its big-step operational semantics we made it clear that Mini-ML$^{\Box}_{\text{ex}}$ is indeed a special case of Mini-ML$_{\text{co}}$. Not in this case either did the LF representation of the big-step operational semantics

and Twelf-verification of its determinacy course any problems. To challenge our LF representation we decided to formulate a big-step-equivalent small-step operational semantics for Mini-$\mathrm{ML}_{\mathrm{co}}$ and have progress and type preservation verified within Twelf. The experiment was successful.

We indeed find it an interesting result that even though the type preserving translation from Mini-$\mathrm{ML}^{\Box}$ into Mini-$\mathrm{ML}^{\Box}_{\mathrm{ex}}$ was represented in a first-order style it was capable of doing the translation between two languages that were represented using higher-order abstract syntax. One could have expected that such a translation would have prevented for at least the target language to be represented using higher-order abstract syntax.

Another interesting experience gained was the fact that we managed to have phenomenons such as staged contexts represented within LF. Whereas the type systems of the metaprogramming languages Mini-$\mathrm{ML}^{\Box}_{\mathrm{ex}}$, Mini-$\mathrm{ML}^{\Box}$, and Mini-$\mathrm{ML}_{\mathrm{co}}$ all contain rules removing assumptions from the surrounding context, it is not possible to literally remove assumptions from an LF context once they have been added. Our solution to this was to make assumptions appropriately invisible in the LF context by intrinsically encoding of well-scopedness on translation to LF. Here considering the Kripke semantics of the underlying modal logic provided a good understanding of how worlds should be attached to expressions in order to make the representation work.

Prior to our intrinsic encoding of worlds we in fact tried to handle well-scopedness by use of an extrinsic judgement. However we did not manage to have the translation from Mini-$\mathrm{ML}^{\Box}$ into Mini-$\mathrm{ML}^{\Box}_{\mathrm{ex}}$ adequately represented using this approach. Once the intrinsical staged encoding was established, the translation problem was solved pretty easily though. This suggests that intrinsic encoding of well-scopedness is indeed a good methodology when dealing with staged computation.

Having succeeded in finding adequate and reasonable representations of different modal logically motivated metaprogramming systems within the logical framework LF, it could be rather interesting to investigate on how well the gained experience could be applied to a metaprogramming system like for example MetaOCaml (see [14]) which is a language inspired by practical needs and thus might not be logically substantiated.

Bibliography

[1] Frank Pfenning and Hao-Chi Wong. *On a Modal λ-calculus for S4*. Electronic Notes in Computer Science, Vol. 1, 1995.

[2] Rowan Davies and Frank Pfenning. *A Modal Analysis of Staged Computation*. In 23rd Annual ACM Symposium on Principles of Programming Languages, January 1996.

[3] Rowan Davies and Frank Pfenning. *A Modal Analysis of Staged Computation*. Journal of the ACM, Vol. 48, No. 3, pp. 555–604, May 2001.

[4] Rowan Davies. *A Temporal-Logic Approach to Binding-Time Analysis*. Logic in Computer Science, 1996.

[5] Frank Pfenning and Rowan Davies. *A Judgmental Reconstruction of Modal Logic*. Mathematical Structures in Computer Science, Vol. 11, No. 4, pp. 511–540, 2001.

[6] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. *Contextual Modal Theory*. ACM Transactions on Computational Logic, Vol. V, No. N, pp. 1–48, February 2007.

[7] Michael Huth and Mark Ryan. *Logic in Computer Science*. Published by Cambridge University Press, 2004.

[8] Frank Pfenning. *Computation and Deduction*. Notes draft, March 2001.

[9] Frank Pfenning. *Logical Frameworks - a Brief Introduction*. H. Schwichtenberg and R. Steinbrüggen (editors), Proof and System-Reliability, Vol. 62 of NATO Science Series II, pp. 137–166, 2002.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. *A Framework for Defining Logics*. Journal of the ACM, Vol. 40, No. 1, pp. 143–184, January 1993.

[11] Frank Pfenning and Carsten Schuermann. *Twelf User's Guide*. Version 1.4, December 2002.

[12] Adam Poswolsky and Carsten Schürmann. *Practical Programming with Higher-Order Encodings and Dependent Types*. S. Drossopoulou (editor), ESOP 2008, LNCS 4960, pp. 93–107, 2008.

[13] Andrew K. Wright and Matthias Felleisen. *A Syntactic Approach to Type Soundness*. Information and Computation, Vol. 115, No. 1, pp. 38–94, 1994.

[14] The home page of MetaOCaml: `http://www.metaocaml.org/`.

Appendix A

Twelf code

A.1 sources.cfg

In sources.cfg the .elf files of the project are listed in dependency order.

```
shared.elf
box_explicit.elf
box_implicit.elf
circle.elf
contextual_box.elf
examples.elf
```

A.2 shared.elf

The file shared.elf contains declarations used by multiple .elf files. The type `world` declared in shared.elf is used by both box_explicit.elf, box_implicit.elf, and contextual_box.elf, and the type `type` is used by box_explicit.elf and box_implicit.elf.

```
void : type. %freeze void.

% --- Worlds

world : type. %name world W.

world_0 : world.

% --- Types

tp : type. %name tp T.

int : tp.
product : tp -> tp -> tp.
arrow : tp -> tp -> tp.
code : tp -> tp.
```

A.3 box_explicit.elf

The file box_explicit.elf contains the Twelf codes concerning the language Mini-$\mathrm{ML}^{\Box}_{\mathrm{ex}}$.

```
%%% Modal Mini-ML: explicit formulation

% --- Explicit expressions

ee : world -> tp -> type. %name ee EE.

z_e : ee W int.
s_e : ee W int -> ee W int.

pair_e : ee W T1 -> ee W T2 -> ee W (product T1 T2).
fst_e : ee W (product T1 T2) -> ee W T1.
snd_e : ee W (product T1 T2) -> ee W T2.

lam_e : (ee W T1 -> ee W T2) -> ee W (arrow T1 T2).
app_e : ee W (arrow T1 T2) -> ee W T1 -> ee W T2.

case_e : ee W int -> ee W T -> (ee W int -> ee W T) -> ee W T.

fix_e : (ee W T -> ee W T) -> ee W T.

box_e : ({w} ee w T) -> ee W (code T).
let_box_e : ee W (code T1) -> (({w} ee w T1) -> ee W T2) -> ee W T2.

% --- Equality of explicit expressions

eq_e : ee W T -> ee W' T' -> type. %name eq_e QP.
eq_e_r : eq_e EE EE.

eq_e_sym : eq_e EE EE' -> eq_e EE' EE -> type. %name eq_e_sym QP.
%mode eq_e_sym +QP -QP'.
eq_e_sym_r : eq_e_sym eq_e_r eq_e_r.
%worlds () (eq_e_sym _ _).
%total {} (eq_e_sym _ _).

eq_e_sym_u : ({W} eq_e ((EE : {w} ee w T) W) ((EE' : {w} ee w T) W)) ->
             ({W} eq_e (EE' W) (EE W)) ->
             type.
%name eq_e_sym_u QP.
%mode eq_e_sym_u +QP -QP'.
eq_e_sym_u_r : eq_e_sym_u ([w : world] eq_e_r) ([w : world] eq_e_r).
%worlds () (eq_e_sym_u _ _).
%total {} (eq_e_sym_u _ _).

eq_e_hole : eq_e EE EE' ->
            {H : ee W T -> ee W' T'}
            eq_e (H EE) (H EE') ->
            type.
%name eq_e_hole QP.
%mode eq_e_hole +QP +H -QP'.
eq_e_hole_r : eq_e_hole eq_e_r H eq_e_r.
%worlds () (eq_e_hole _ _ _).
%total {} (eq_e_hole _ _ _).

eq_e_hole_u : ({W : world} eq_e (EE W) (EE' W)) ->
              {H : ({W'} ee W' T) -> ee W'' T'} eq_e (H EE) (H EE') ->
              type.
%name eq_e_hole_u QP.
%mode eq_e_hole_u +QP +H -QP'.
eq_e_hole_u_r : eq_e_hole_u ([w] eq_e_r) H eq_e_r.
%worlds () (eq_e_hole_u _ _ _).
%total {} (eq_e_hole_u _ _ _).

eq_e_s : eq_e EE EE' -> eq_e (s_e EE) (s_e EE') -> type. %name eq_e_s QP.
%mode eq_e_s +QP -QP'.
eq_e_s_r : eq_e_s eq_e_r eq_e_r.
%worlds () (eq_e_s _ _).
%total {} (eq_e_s _ _).

eq_e_s_b : eq_e (s_e EE) (s_e EE') -> eq_e EE EE' -> type. %name eq_e_s_b QP.
%mode eq_e_s_b +QP -QP'.
eq_e_s_b_r : eq_e_s_b eq_e_r eq_e_r.
%worlds () (eq_e_s_b _ _).
```

```
%total {} (eq_e_s_b _ _).

neq_e_zs : eq_e (z_e : ee W int) ((s_e EE) : ee W int) ->
           eq_e EE1 EE2 ->
           type.
%name neq_e_zs QP.
%mode +{W : world} +{EE : ee W int}
      +{QP : eq_e z_e (s_e EE)}
      +{W1 : world} +{T1 : tp} +{EE1 : ee W1 T1}
      +{W2 : world} +{T2 : tp} +{EE2 : ee W2 T2}
      -{QP' : eq_e EE1 EE2}
      neq_e_zs QP QP'.
%worlds () (neq_e_zs _ _).
%total {} (neq_e_zs _ _).

eq_e_pair : eq_e EE1 EE1' ->
            eq_e EE2 EE2' ->
            eq_e (pair_e EE1 EE2) (pair_e EE1' EE2') ->
            type.
%name eq_e_pair QP.
%mode eq_e_pair +QP1 +QP2 -QP.
eq_e_pair_r : eq_e_pair eq_e_r eq_e_r eq_e_r.
%worlds () (eq_e_pair _ _ _).
%total {} (eq_e_pair _ _ _).

eq_e_pair_1 : eq_e (pair_e EE1 EE2) (pair_e EE1' EE2') ->
              eq_e EE1 EE1' ->
              type.
%name eq_e_pair_1 QP.
%mode eq_e_pair_1 +QP -QP1.
eq_e_pair_1_r : eq_e_pair_1 eq_e_r eq_e_r.
%worlds () (eq_e_pair_1 _ _).
%total {} (eq_e_pair_1 _ _).

eq_e_pair_2 : eq_e (pair_e EE1 EE2) (pair_e EE1' EE2') ->
              eq_e EE2 EE2' ->
              type.
%name eq_e_pair_2 QP.
%mode eq_e_pair_2 +QP -QP2.
eq_e_pair_2_r : eq_e_pair_2 eq_e_r eq_e_r.
%worlds () (eq_e_pair_2 _ _).
%total {} (eq_e_pair_2 _ _).

eq_e_lam_b : eq_e (lam_e EE) (lam_e EE') ->
             eq_e X X' ->
             eq_e (EE X) (EE' X') ->
             type.
%name eq_e_lam_b QP.
%mode eq_e_lam_b +QP1 +QP2 -QP3.
eq_e_lam_b_r : eq_e_lam_b eq_e_r eq_e_r eq_e_r.
%worlds () (eq_e_lam_b _ _ _).
%total {} (eq_e_lam_b _ _ _).

eq_e_box_b : eq_e (box_e EE) (box_e EE') -> ({W} eq_e (EE W) (EE' W)) -> type.
%name eq_e_box_b QP.
%mode eq_e_box_b +QP -QP'.
eq_e_box_b_r : eq_e_box_b eq_e_r ([w : world] eq_e_r).
%worlds () (eq_e_box_b _ _).
%total {} (eq_e_box_b _ _).

% --- Explicit values

val_e : ee W T -> type. %name val_e VEP.
%mode val_e +EE.
val_e_z : val_e z_e.
val_e_s : val_e (s_e VE)
           <- val_e VE.
val_e_pair : val_e (pair_e VE1 VE2)
             <- val_e VE1
             <- val_e VE2.
val_e_lam : val_e (lam_e EE).
val_e_box : val_e (box_e EE).
```

```
%worlds () (val_e _).

val_e_pair_split : val_e (pair_e VE1 VE2) -> val_e VE1 -> val_e VE2 -> type.
%mode val_e_pair_split +VEP -VEP1 -VEP2.
val_e_pair_split_proof : val_e_pair_split (val_e_pair VEP2 VEP1) VEP1 VEP2.

% --- Operational semantics for explicit expressions

% -- Big-step evaluation rules for explicit expressions:
eval_e : ee W T -> ee W T -> type. %name eval_e EEP.
%mode eval_e +EE -VE.

eval_e_z : eval_e z_e z_e.

eval_e_s : eval_e (s_e EE) (s_e VE)
            <- eval_e EE VE.

eval_e_pair : eval_e (pair_e EE1 EE2) (pair_e VE1 VE2)
               <- eval_e EE1 VE1
               <- eval_e EE2 VE2.

eval_e_fst : eval_e (fst_e EE) VE1
              <- eval_e EE (pair_e VE1 VE2).

eval_e_snd : eval_e (snd_e EE) VE2
              <- eval_e EE (pair_e VE1 VE2).

eval_e_lam : eval_e (lam_e EE) (lam_e EE).

eval_e_app : eval_e (app_e EE1 EE2) VE1
              <- eval_e EE1 (lam_e EE1')
              <- eval_e EE2 VE2
              <- eval_e (EE1' VE2) VE1.

eval_e_case_z : eval_e (case_e EE EE1 EE2) VE
                 <- eval_e EE z_e
                 <- eval_e EE1 VE.

eval_e_case_s : eval_e (case_e EE EE1 EE2) VE'
                 <- eval_e EE (s_e VE)
                 <- eval_e (EE2 VE) VE'.

eval_e_fix : eval_e (fix_e EE) VE
              <- eval_e (EE (fix_e EE)) VE.

eval_e_box : eval_e (box_e EE) (box_e EE).

eval_e_let_box : eval_e (let_box_e EE1 EE2) VE
                  <- eval_e EE1 (box_e EE1')
                  <- eval_e (EE2 EE1') VE.

%worlds () (eval_e _ _).
%covers eval_e +EE -VE.

% -- Value soundness for evaluation of explicit expressions:
val_sound_e : eval_e EE VE -> val_e VE -> type. %name val_sound_e VSEP.
%mode val_sound_e +EEP -VEP.

val_sound_e_z : val_sound_e eval_e_z val_e_z.

val_sound_e_s : val_sound_e (eval_e_s EEP) (val_e_s VEP)
                 <- val_sound_e EEP VEP.

val_sound_e_pair : val_sound_e (eval_e_pair EEP2 EEP1) (val_e_pair VEP2 VEP1)
                    <- val_sound_e EEP1 VEP1
                    <- val_sound_e EEP2 VEP2.

val_sound_e_fst : val_sound_e (eval_e_fst EEP) VEP1
                   <- val_sound_e EEP (val_e_pair VEP2 VEP1).

val_sound_e_snd : val_sound_e (eval_e_snd EEP) VEP2
                   <- val_sound_e EEP (val_e_pair VEP2 VEP1).
```

```
val_sound_e_lam : val_sound_e eval_e_lam val_e_lam.

val_sound_e_app : val_sound_e (eval_e_app EEP3 EEP2 EEP1) VEP
                   <- val_sound_e EEP3 VEP.

val_sound_e_case_z : val_sound_e (eval_e_case_z EEP1 EEP) VEP1
                      <- val_sound_e EEP1 VEP1.

val_sound_e_case_s : val_sound_e (eval_e_case_s EEP2 EEP) VEP2
                      <- val_sound_e EEP2 VEP2.

val_sound_e_fix : val_sound_e (eval_e_fix EEP) VEP
                   <- val_sound_e EEP VEP.

val_sound_e_box : val_sound_e eval_e_box val_e_box.

val_sound_e_let_box : val_sound_e (eval_e_let_box EEP2 EEP1) VEP
                       <- val_sound_e EEP2 VEP.

%worlds () (val_sound_e _ _).
%total EEP (val_sound_e EEP _).

% -- Equality of evaluation of explicit expressions:
eq_e_eval : eq_e EE EE' -> eval_e EE VE -> eval_e EE' VE -> type.
%name eq_e_eval QP.
%mode eq_e_eval +QP +EEP -EEP'.
eq_e_eval_r : eq_e_eval eq_e_r EEP EEP.
%worlds () (eq_e_eval _ _ _).
%total {} (eq_e_eval _ _ _).

% -- Determinacy for evaluation of explicit expressions:
eval_det_e : eval_e EE VE -> eval_e EE VE' -> eq_e VE VE' -> type.
%mode eval_det_e +EEP +EEP' -QP.

eval_det_e_z : eval_det_e eval_e_z eval_e_z eq_e_r.

eval_det_e_s : eval_det_e (eval_e_s EEP) (eval_e_s EEP') QP'
                <- eval_det_e EEP EEP' QP
                <- eq_e_s QP QP'.

eval_det_e_pair : eval_det_e (eval_e_pair EEP2 EEP1)
                             (eval_e_pair EEP2' EEP1')
                             QP
                   <- eval_det_e EEP1 EEP1' QP1
                   <- eval_det_e EEP2 EEP2' QP2
                   <- eq_e_pair QP1 QP2 QP.

eval_det_e_fst : eval_det_e (eval_e_fst EEP) (eval_e_fst EEP') QP1
                  <- eval_det_e EEP EEP' QP
                  <- eq_e_pair_1 QP QP1.

eval_det_e_snd : eval_det_e (eval_e_snd EEP) (eval_e_snd EEP') QP2
                  <- eval_det_e EEP EEP' QP
                  <- eq_e_pair_2 QP QP2.

eval_det_e_lam : eval_det_e eval_e_lam eval_e_lam eq_e_r.

eval_det_e_app : eval_det_e (eval_e_app EEP3 EEP2 EEP1)
                            (eval_e_app EEP3' EEP2' EEP1')
                            QP''
                  <- eval_det_e EEP1 EEP1' QP1
                  <- eval_det_e EEP2 EEP2' QP2
                  <- eq_e_lam_b QP1 QP2 QP
                  <- eq_e_sym QP QP'
                  <- eq_e_eval QP' EEP3' EEP3''
                  <- eval_det_e EEP3 EEP3'' QP''.

eval_det_e_case_zz : eval_det_e (eval_e_case_z EEP1 EEP)
                                (eval_e_case_z EEP1' EEP')
                                QP1
                      <- eval_det_e EEP1 EEP1' QP1.
```

```
eval_det_e_case_zs : eval_det_e (eval_e_case_z EEP1 EEP)
                                (eval_e_case_s EEP1' EEP')
                                QP'
                     <- eval_det_e EEP EEP' QP
                     <- neq_e_zs QP QP'.

eval_det_e_case_sz : eval_det_e (eval_e_case_s EEP1 EEP)
                                (eval_e_case_z EEP1' EEP')
                                QP''
                     <- eval_det_e EEP EEP' QP
                     <- eq_e_sym QP QP'
                     <- neq_e_zs QP' QP''.

eval_det_e_case_ss :
  eval_det_e ((eval_e_case_s EEP1 EEP) : eval_e (case_e _ _ EE2) _)
             (eval_e_case_s EEP1' EEP')
             QP''''
   <- eval_det_e EEP EEP' QP
   <- eq_e_s_b QP QP'
   <- eq_e_sym QP' QP''
   <- eq_e_hole QP'' EE2 QP'''
   <- eq_e_eval QP''' EEP1' EEP1''
   <- eval_det_e EEP1 EEP1'' QP''''.

eval_det_e_fix : eval_det_e (eval_e_fix EEP) (eval_e_fix EEP') QP
                  <- eval_det_e EEP EEP' QP.

eval_det_e_box : eval_det_e eval_e_box eval_e_box eq_e_r.

eval_det_e_let_box :
  eval_det_e ((eval_e_let_box EEP2 EEP1) : (eval_e (let_box_e _ EE2) _))
             (eval_e_let_box EEP2' EEP1')
             QP'''
   <- eval_det_e EEP1 EEP1' QP1
   <- eq_e_box_b QP1 QP
   <- eq_e_sym_u QP QP'
   <- eq_e_hole_u QP' EE2 QP''
   <- eq_e_eval QP'' EEP2' EEP2''
   <- eval_det_e EEP2 EEP2'' QP'''.

%worlds () (eval_det_e _ _ _).
%total EEP (eval_det_e EEP _ _).
```

A.4 box_implicit.elf

The file box_implicit.elf contains the Twelf codes concerning the language Mini-ML$^{\Box}$. This includes the translation from Mini-ML$^{\Box}$ into Mini-ML$^{\Box}_{\text{ex}}$.

```
%%% Modal Mini-ML: implicit formulation

% --- World jumps

world_jump : world -> world -> type. %name world_jump WJP.
%mode world_jump +W +W'.

% -- Normal worlds:
world_norm : world -> type. %name world_norm WNP.
world_norm_0 : world_norm world_0.
world_norm_n : world_norm W
                  <- world_norm W'
                  <- world_jump W' W.

% -- Equality of worlds:
eq_world : world -> world -> type. %name eq_world WQP.
eq_world_r : eq_world W W.
```

```
% -- Equality of world normality:
eq_world_norm : eq_world W W' -> world_norm W -> world_norm W' -> type.
%mode eq_world_norm +EWP +WNP -WNP'.
eq_world_norm_r: eq_world_norm EWP WNP WNP.

% -- Strip lambda wjp from world jump:
elim_world_jump : ({w} world_jump W' w -> world_jump W0 W1) ->
                  world_jump W0 W1 ->
                  type.
%mode elim_world_jump +WJP -WJP'.
elim_world_jump_r : elim_world_jump ([w] [wjp] WJP) WJP.

% -- Each world has at most one predecessor:
world_pred_unq : world_jump W' W ->
                 world_jump W'' W ->
                 eq_world W' W'' ->
                 type.
%mode world_pred_unq +WJP +WJP' -EWP.
world_pred_unq_r : world_pred_unq WJP WJP' eq_world_r.

% -- Predecessing or equal worlds:
world_pred_eq : world -> world -> type.
world_pred_eq_0 : world_pred_eq W W.
world_pred_eq_n : world_pred_eq W W'
                   <- world_pred_eq W W''
                   <- world_jump W'' W'.

% -- Front extension of predecessing or equal worlds:
world_pred_eq_front : world_jump W W' ->
                      world_pred_eq W' W'' ->
                      world_pred_eq W W'' ->
                      type.
%mode world_pred_eq_front +WJP +WPEP -WPEP'.
world_pred_eq_front_0 :
  world_pred_eq_front WJP
                      world_pred_eq_0
                      (world_pred_eq_n WJP world_pred_eq_0).
world_pred_eq_front_n : world_pred_eq_front WJP
                                            (world_pred_eq_n WJP' WPEP)
                                            (world_pred_eq_n WJP' WPEP')
                         <- world_pred_eq_front WJP WPEP WPEP'.

% -- Predecessing worlds:
world_pred : world -> world -> type.
world_pred_n : world_pred W W'
                <- world_pred_eq W W''
                <- world_jump W'' W'.

% -- Front extension of predecessing worlds:
world_pred_front : world_jump W W' ->
                   world_pred_eq W' W'' ->
                   world_pred W W'' ->
                   type.
%mode world_pred_front +WJP +WPEP -WPP.
world_pred_front_0 : world_pred_front WJP
                                      world_pred_eq_0
                                      (world_pred_n WJP world_pred_eq_0).
world_pred_front_n : world_pred_front WJP
                                      (world_pred_eq_n WJP' WPEP)
                                      (world_pred_n WJP' WPEP')
                      <- world_pred_eq_front WJP WPEP WPEP'.

% -- Equality of world predecessing:
eq_world_pred : eq_world W W' -> world_pred W W' -> world_pred W W -> type.
%mode eq_world_pred +EWP +WPP -WPP'.
eq_world_pred_r : eq_world_pred EWP WPP WPP.

% -- Irreflexivity of world predecessing:
world_pred_irrefl : world_norm W -> world_pred W W -> void -> type.
%mode world_pred_irrefl +WNP +WPP -V.
world_pred_irrefl_n :
  world_pred_irrefl (world_norm_n WJP WNP) (world_pred_n WJP' WPEP) V
```

```
   <- world_pred_unq WJP WJP' EWP
   <- world_pred_front WJP WPEP WPP
   <- eq_world_pred EWP WPP WPP'
   <- world_pred_irrefl WNP WPP' V.

% --- Implicit expressions

pop : world -> tp -> type. %name pop P.

ei : world -> tp -> type. %name ei EI.

z_i : ei W int.
s_i : ei W int -> ei W int.

pair_i : ei W T1 -> ei W T2 -> ei W (product T1 T2).
fst_i : ei W (product T1 T2) -> ei W T1.
snd_i : ei W (product T1 T2) -> ei W T2.

lam_i : (ei W T1 -> ei W T2) -> ei W (arrow T1 T2).
app_i : ei W (arrow T1 T2) -> ei W T1 -> ei W T2.

case_i : ei W int -> ei W T -> (ei W int -> ei W T) -> ei W T.

fix_i : (ei W T -> ei W T) -> ei W T.

box_i : ({w} world_jump W w -> ei w T) -> ei W (code T).

unbox :  pop W (code T) -> ei W T.

pop_0 : ei W T -> pop W T.
pop_m : world_jump W' W -> pop W' (code T) -> pop W (code T).

% --- Translation from implicit into explicit language

% -- Type list:
ctx : type. %name ctx C.
ctx_0 : ctx.
ctx_n : tp -> ctx -> ctx.

% -- Type list prefix:
ctx_prefix : ctx -> ctx -> type. %name ctx_prefix CPP.
ctx_prefix_0 : ctx_prefix ctx_0 C.
ctx_prefix_n : ctx_prefix (ctx_n T C) (ctx_n T C')
                <- ctx_prefix C C'.

% -- Type list prefix - transitivity:
ctx_prefix_trans : ctx_prefix C1 C2 ->
                   ctx_prefix C2 C3 ->
                   ctx_prefix C1 C3 ->
                   type.
%mode ctx_prefix_trans +CPP1 +CPP2 -CPP3.
ctx_prefix_trans_0 : ctx_prefix_trans ctx_prefix_0 CPP ctx_prefix_0.
ctx_prefix_trans_n : ctx_prefix_trans (ctx_prefix_n CPP1)
                                      (ctx_prefix_n CPP2)
                                      (ctx_prefix_n CPP3)
                      <- ctx_prefix_trans CPP1 CPP2 CPP3.

% -- Type list prefix - reflexivity:
ctx_prefix_refl : {C} ctx_prefix C C -> type.
%mode ctx_prefix_refl +C -CPP.
ctx_prefix_refl_0 : ctx_prefix_refl ctx_0 ctx_prefix_0.
ctx_prefix_refl_n : ctx_prefix_refl (ctx_n T C)
                                    (ctx_prefix_n CPP)
                     <- ctx_prefix_refl C CPP.

% -- List of type lists:
ktx : type. %name ktx K.
ktx_0 : ktx.
ktx_n : ktx -> ctx -> ktx.

% -- List of type lists prefix:
ktx_prefix : ktx -> ktx -> type. %name ktx_prefix KPP.
```

```
ktx_prefix_0 : ktx_prefix ktx_0 ktx_0.
ktx_prefix_n : ktx_prefix (ktx_n K C) (ktx_n K' C')
                <- ktx_prefix K K'
                <- ctx_prefix C C'.

% -- List of type lists prefix - transitivity:
ktx_prefix_trans : ktx_prefix K K' ->
                   ktx_prefix K' K'' ->
                   ktx_prefix K K'' ->
                   type.
%mode ktx_prefix_trans +KPP1 +KPP2 -KPP3.
ktx_prefix_trans_0 : ktx_prefix_trans ktx_prefix_0 ktx_prefix_0 ktx_prefix_0.
ktx_prefix_trans_n : ktx_prefix_trans (ktx_prefix_n CTP1 KTP1)
                                      (ktx_prefix_n CTP2 KTP2)
                                      (ktx_prefix_n CTP3 KTP3)
                      <- ktx_prefix_trans KTP1 KTP2 KTP3
                      <- ctx_prefix_trans CTP1 CTP2 CTP3.

% -- List of type lists prefix - reflexivity:
ktx_prefix_refl : {K} ktx_prefix K K -> type.
%mode ktx_prefix_refl +K -KPP.
ktx_prefix_refl_0 : ktx_prefix_refl ktx_0 ktx_prefix_0.
ktx_prefix_refl_n : ktx_prefix_refl (ktx_n K C)
                                    (ktx_prefix_n CPP KPP)
                     <- ktx_prefix_refl K KPP
                     <- ctx_prefix_refl C CPP.

% -- Labels:
lab_c : ctx -> tp -> type. %name lab_c LC.
lab_c_0 : {C} lab_c (ctx_n T0 C) T0.
lab_c_n : {T0} lab_c C T -> lab_c (ctx_n T0 C) T.

lab_k : ktx -> tp -> type. %name lab_k LK.
lab_k_0 : {K} lab_c C T -> lab_k (ktx_n K C) T.
lab_k_m : {C} lab_k K T -> lab_k (ktx_n K C) T.

% -- Explicit expressions containing labels:
eek : world -> ktx -> tp -> type. %name eek EEK.

z_ek : eek W K int.
s_ek : eek W K int -> eek W K int.

pair_ek : eek W K T1 -> eek W K T2 -> eek W K (product T1 T2).
fst_ek : eek W K (product T1 T2) -> eek W K T1.
snd_ek : eek W K (product T1 T2) -> eek W K T2.

lam_ek : (({k} eek W k T1) -> eek W K T2) -> eek W K (arrow T1 T2).
app_ek : eek W K (arrow T1 T2) -> eek W K T1 -> eek W K T2.

case_ek : eek W K int ->
          eek W K T ->
          (({k} eek W k int) -> eek W K T) ->
          eek W K T.

fix_ek : (({k} eek W k T) -> eek W K T) -> eek W K T.

box_ek : ({w} eek w K T) -> eek W K (code T).
let_box_ek : eek W K (code T1) ->
             (({w} {k} eek w k T1) -> eek W K T2) ->
             eek W K T2.

lab_exp : lab_k K T -> {w} eek w K T.

% -- Any explicit expression can be derived from void:
void_eek : void -> ({k} {t} {w} eek w k t) -> type.
%mode void_eek +V -EEK.

% -- Stripping lambda w from expression:
elim_world : (world -> eek W K T) -> eek W K T -> type.
%mode elim_world +EEK -EEK'.

elim_world_z : elim_world ([w] z_ek) z_ek.
```

```
elim_world_s : elim_world ([w] (s_ek (EEK* w))) (s_ek EEK)
               <- elim_world EEK* EEK.

elim_world_pair : elim_world ([w] (pair_ek (EEK1* w) (EEK2* w)))
                             (pair_ek EEK1 EEK2)
                  <- elim_world EEK1* EEK1
                  <- elim_world EEK2* EEK2.

elim_world_fst : elim_world ([w] (fst_ek (EEK* w))) (fst_ek EEK)
                  <- elim_world EEK* EEK.

elim_world_snd : elim_world ([w] (snd_ek (EEK* w))) (snd_ek EEK)
                  <- elim_world EEK* EEK.

elim_world_lam : elim_world ([w] (lam_ek ([x] EEK* x w))) (lam_ek EEK)
                  <- ({x}
                      ({k} elim_world ([w] x k) (x k)) ->
                      elim_world (EEK* x) (EEK x)).

elim_world_app : elim_world ([w] (app_ek (EEK1* w) (EEK2* w)))
                             (app_ek EEK1 EEK2)
                  <- elim_world EEK1* EEK1
                  <- elim_world EEK2* EEK2.

elim_world_case : elim_world ([w] (case_ek (EEK* w) (EEK1* w) ([x] EEK2* x w)))
                             (case_ek EEK EEK1 EEK2)
                   <- elim_world EEK* EEK
                   <- elim_world EEK1* EEK1
                   <- ({x}
                       ({k} elim_world ([w] x k) (x k)) ->
                       elim_world (EEK2* x) (EEK2 x)).

elim_world_fix : elim_world ([w] (fix_ek ([x] EEK* x w)))
                             (fix_ek EEK)
                  <- ({x}
                      ({k} elim_world ([w] x k) (x k)) ->
                      elim_world (EEK* x) (EEK x)).

elim_world_box : elim_world ([w] box_ek ([w'] EEK* w' w )) (box_ek EEK)
                  <- ({w'} elim_world ([w] EEK* w' w) (EEK w')).

elim_world_let_box : elim_world ([w] let_box_ek (EEK1* w) ([u] EEK2* u w))
                                (let_box_ek EEK1 EEK2)
                     <- elim_world EEK1* EEK1
                     <- ({u}
                           ({w'} {k} elim_world ([w] u w' k) (u w' k)) ->
                         elim_world ([w] EEK2* u w) (EEK2 u)).

elim_world_lab_exp : elim_world ([w] ((lab_exp LK) W)) ((lab_exp LK) W).

%block elw_b1 : some {W : world} {T : tp}
                block {x : {k : ktx} eek W k T}
                      {elwp : {k : ktx} elim_world ([w] x k) (x k)}.
%block elw_b2 : block {w' : world}.
%block elw_b3 : some {T : tp}
                block  {u : {w : world} {k : ktx} eek w k T}
                       {elwp : {w' : world} {k : ktx}
                               elim_world ([w] u w' k) (u w' k)}.

% -- Stripping lambda x from expression:
elim_x : world_norm W ->
         world_pred W W' ->
         (({k : ktx} eek W' k T') -> eek W K T) ->
         eek W K T ->
         type.
%mode elim_x +WNP +WPP +EEK -EEK'.

elim_x_z : elim_x WNP WPP ([x] z_ek) z_ek.

elim_x_s : elim_x WNP WPP ([x] (s_ek (EEK* x))) (s_ek EEK)
            <- elim_x WNP WPP EEK* EEK.
```

```
elim_x_pair : elim_x WNP
                     WPP
                     ([x] (pair_ek (EEK1* x) (EEK2* x)))
                     (pair_ek EEK1 EEK2)
               <- elim_x WNP WPP EEK1* EEK1
               <- elim_x WNP WPP EEK2* EEK2.

elim_x_fst : elim_x WNP WPP ([x] (fst_ek (EEK* x))) (fst_ek EEK)
              <- elim_x WNP WPP EEK* EEK.

elim_x_snd : elim_x WNP WPP ([x] (snd_ek (EEK* x))) (snd_ek EEK)
              <- elim_x WNP WPP EEK* EEK.

elim_x_x : elim_x WNP WPP ([x] x K) (EEK K T W)
            <- world_pred_irrefl WNP WPP V
            <- void_eek V EEK.

elim_x_e : elim_x WNP WPP ([x] E) E.

elim_x_lam : elim_x WNP WPP ([x] (lam_ek ([x'] EEK* x' x))) (lam_ek EEK)
              <- ({x'}
                  ({w'} {wpp : world_pred W w'} {k}
                   elim_x WNP wpp ([x : {k} eek w' k T] (x' k)) (x' k)) ->
                  elim_x WNP WPP ([x] EEK* x' x) (EEK x')).

elim_x_app : elim_x WNP WPP
                    ([x] (app_ek (EEK1* x) (EEK2* x)))
                    (app_ek EEK1 EEK2)
              <- elim_x WNP WPP EEK1* EEK1
              <- elim_x WNP WPP EEK2* EEK2.

elim_x_case : elim_x WNP
                     WPP
                     ([x] (case_ek (EEK* x) (EEK1* x) ([x'] EEK2* x' x)))
                     (case_ek EEK EEK1 EEK2)
               <- elim_x WNP WPP EEK* EEK
               <- elim_x WNP WPP EEK1* EEK1
               <- ({x'}
                   ({w'} {wpp : world_pred W w'} {k}
                    elim_x WNP wpp ([x : {k} eek w' k T] (x' k)) (x' k)) ->
                   elim_x WNP WPP ([x] EEK2* x' x) (EEK2 x')).

elim_x_fix : elim_x WNP WPP ([x] (fix_ek ([x'] EEK* x' x))) (fix_ek EEK)
              <- ({x'}
                  ({w'} {wpp : world_pred W w'} {k}
                   elim_x WNP wpp ([x : {k} eek w' k T] (x' k)) (x' k)) ->
                  elim_x WNP WPP ([x] EEK* x' x) (EEK x')).

elim_x_box : elim_x WNP WPP ([x] box_ek ([w] (EEK* w x))) (box_ek EEK)
              <- ({w} {wnp : world_norm w} {wpp : world_pred w W'}
                  elim_x wnp wpp ([x] EEK* w x) (EEK w)).

elim_x_let_box : elim_x WNP WPP
                        ([x] let_box_ek (EEK1* x) ([u] EEK2* u x))
                        (let_box_ek EEK1 EEK2)
                  <- elim_x WNP WPP EEK1* EEK1
                  <- ({u}
                      ({w} {wnp : world_norm w} {w'} {wpp : world_pred w w'}
                       {k}
                       elim_x wnp wpp ([x : {k} eek w' k T] u w k) (u w k)) ->
                      elim_x WNP WPP ([x] EEK2* u x) (EEK2 u)).

elim_x_lab_exp : elim_x WNP WPP ([x] ((lab_exp LK) W)) ((lab_exp LK) W).

%block elx_b1 :
  some {W : world} {WNP : world_norm W} {T : tp} {T1 : tp}
  block {x' : {k : ktx} eek W k T1}
        {elxp : {w' : world} {wpp : world_pred W w'} {k : ktx}
                elim_x WNP wpp ([x : {k : ktx} eek w' k T] x' k) (x' k)}.
%block elx_b2 : some {W' : world}
                block {w : world} {wnp : world_norm w} {wpp : world_pred w W'}.
```

```
%block elx_b3 :
  some {T : tp} {T' : tp}
  block  {u : {w : world} {k : ktx} eek w k T'}
         {elxp : {w : world} {wnp : world_norm w}
                 {w' : world} {wpp : world_pred w w'}
                 {k : ktx}
                 elim_x wnp wpp ([x : {k' : ktx} eek w' k' T] u w k) (u w k)}.

% -- Replacement of label 0 with var and decrementing of other labels with one:
repl_lab : eek W (ktx_n K (ctx_n T0 C)) T ->
           (({w} {k} eek w k T0) -> eek W (ktx_n K C) T) ->
           type.
%mode repl_lab +EEK -EEK'.

repl_lab_z : repl_lab z_ek ([u] z_ek).

repl_lab_s : repl_lab (s_ek EEK) ([u] (s_ek (EEK' u)))
              <- repl_lab EEK EEK'.

repl_lab_pair : repl_lab (pair_ek EEK1 EEK2) ([u] (pair_ek (EEK1' u) (EEK2' u)))
                 <- repl_lab EEK1 EEK1'
                 <- repl_lab EEK2 EEK2'.

repl_lab_fst : repl_lab (fst_ek EEK) ([u] (fst_ek (EEK' u)))
                <- repl_lab EEK EEK'.

repl_lab_snd : repl_lab (snd_ek EEK) ([u] (snd_ek (EEK' u)))
                <- repl_lab EEK EEK'.

repl_lab_lam : repl_lab (lam_ek EEK) ([u] (lam_ek ([x] ((EEK' x) u))))
                <- ({x} ({t0} {c} {k} repl_lab (x (ktx_n k (ctx_n t0 c)))
                                               ([u] (x (ktx_n k c)))) ->
                        repl_lab (EEK x) (EEK' x)).

repl_lab_app : repl_lab (app_ek EEK1 EEK2) ([u] (app_ek (EEK1' u) (EEK2' u)))
                <- repl_lab EEK1 EEK1'
                <- repl_lab EEK2 EEK2'.

repl_lab_case : repl_lab (case_ek EEK EEK1 EEK2)
                         ([u] case_ek (EEK' u) (EEK1' u) ([x] EEK2' x u))
                 <- repl_lab EEK EEK'
                 <- repl_lab EEK1 EEK1'
                 <- ({x} ({t0} {c} {k} repl_lab (x (ktx_n k (ctx_n t0 c)))
                                                ([u] (x (ktx_n k c)))) ->
                          repl_lab (EEK2 x) (EEK2' x)).

repl_lab_fix : repl_lab (fix_ek EEK) ([u] fix_ek ([x] EEK' x u))
                <- ({x} ({t0} {c} {k} repl_lab (x (ktx_n k (ctx_n t0 c)))
                                               ([u] (x (ktx_n k c)))) ->
                        repl_lab (EEK x) (EEK' x)).

repl_lab_box : repl_lab (box_ek EEK) ([u] (box_ek ([w] (EEK' w u))))
                <- ({w} repl_lab (EEK w) (EEK' w)).

repl_lab_let_box :
  repl_lab (let_box_ek EEK1 EEK2)
           ([u] (let_box_ek (EEK1' u) ([u'] EEK2' u' u)))
   <- repl_lab EEK1 EEK1'
   <- ({u} ({w} {t0} {c} {k} repl_lab (u w (ktx_n k (ctx_n t0 c)))
                                      ([u'] (u w (ktx_n k c)))) ->
           repl_lab (EEK2 u) (EEK2' u)).

repl_lab_lab_exp_0_0 : repl_lab ((lab_exp (lab_k_0 K (lab_c_0 C))) W)
                                ([u] (u W (ktx_n K C))).

repl_lab_lab_exp_0_n : repl_lab ((lab_exp (lab_k_0 K (lab_c_n T0 LC))) W)
                                ([u] ((lab_exp (lab_k_0 K LC)) W)).

repl_lab_lab_exp_m_n : repl_lab ((lab_exp (lab_k_m (ctx_n T0 C) LK)) W)
                                ([u] ((lab_exp (lab_k_m C LK)) W)).

%block rl_b1 :
```

```
  some {W : world} {T : tp}
  block {x : {k : ktx} eek W k T}
        {rlp : {t0 : tp} {c : ctx} {k : ktx}
               repl_lab (x (ktx_n k (ctx_n t0 c)))
                        ([u : {w : world} {k : ktx} eek w k t0]
                           (x (ktx_n k c)))}.
%block rl_b2 : block {w : world}.
%block rl_b3 :
  some {T1 : tp}
  block {u : {w : world} {k : ktx} eek w k T1}
        {rlp : {w : world} {t0 : tp} {c : ctx} {k : ktx}
               repl_lab (u w (ktx_n k (ctx_n t0 c)))
                        ([u' : {w : world} {k : ktx} eek w k t0]
                           (u w (ktx_n k c)))}.

% -- List of explicit expressions:
sequence : world -> ktx -> ctx -> type. %name sequence S.
sequence_0 : sequence W K ctx_0.
sequence_n : eek W K (code T) -> sequence W K C -> sequence W K (ctx_n T C).
%freeze sequence.

% -- End extension of explicit expression list:
sequence_end : sequence W K C ->
               eek W K (code T) ->
               sequence W K C' ->
               ctx_prefix C C' ->
               lab_c C' T ->
               type.
%name sequence_end SEP.
%mode sequence_end +S +EEK -S' -CPP -LC.
sequence_end_0 : sequence_end sequence_0
                              EEK
                              (sequence_n EEK sequence_0)
                              ctx_prefix_0
                              (lab_c_0 ctx_0).
sequence_end_n : sequence_end (sequence_n EEK S)
                              EEK'
                              (sequence_n EEK S')
                              (ctx_prefix_n CPP) (lab_c_n T0 LC)
                 <- sequence_end S EEK' S' CPP LC.

% -- Stripping lambda w from explicit expression list:
elim_sequence_world : (world -> sequence W K C) -> sequence W K C -> type.
%mode elim_sequence_world +S* -S.
elim_sequence_world_0 : elim_sequence_world ([w] sequence_0) sequence_0.
elim_sequence_world_n : elim_sequence_world ([w] sequence_n (EEK* w) (S* w))
                                            (sequence_n EEK S)
                         <- elim_world EEK* EEK
                         <- elim_sequence_world S* S.

% -- Stripping lambda x from explicit expression list:
elim_sequence_x : world_norm W ->
                  world_pred W W' ->
                  (({k} eek W' k T) -> sequence W K C) ->
                  sequence W K C ->
                  type.
%mode elim_sequence_x +WNP +WPP +S* -S.
elim_sequence_x_0 : elim_sequence_x WNP WPP ([x] sequence_0) sequence_0.
elim_sequence_x_n : elim_sequence_x WNP
                                    WPP
                                    ([x] sequence_n (EEK* x) (S* x))
                                    (sequence_n EEK S)
                  <- elim_x WNP WPP EEK* EEK
                  <- elim_sequence_x WNP WPP S* S.

world_eq_sequence : eq_world W W' -> sequence W' K C -> sequence W K C -> type.
%mode world_eq_sequence +WQP +S -S'.
world_eq_sequence_r : world_eq_sequence eq_world_r S S.

weak_lab_c : ctx_prefix C C' ->
             lab_c C T ->
             lab_c C' T ->
```

```
            type.
%mode weak_lab_c +CPP +LC -LC'.
weak_lab_c_0 : weak_lab_c (ctx_prefix_n CPP) (lab_c_0 C) (lab_c_0 C').
weak_lab_c_n : weak_lab_c (ctx_prefix_n CPP) (lab_c_n T0 LC) (lab_c_n T0 LC')
               <- weak_lab_c CPP LC LC'.

weak_lab_k : ktx_prefix K K' ->
             lab_k K T ->
             lab_k K' T ->
             type.
%mode weak_lab_k +KPP +LK -LK'.
weak_lab_k_0 : weak_lab_k (ktx_prefix_n CPP KPP)
                          (lab_k_0 K LC)
                          (lab_k_0 K' LC')
               <- weak_lab_c CPP LC LC'.
weak_lab_k_m : weak_lab_k (ktx_prefix_n CPP KPP)
                          (lab_k_m C LK)
                          (lab_k_m C' LK')
               <- weak_lab_k KPP LK LK'.

weak_eek : ktx_prefix K K' -> eek W K T -> eek W K' T -> type.
%name weak_eek WEP.
%mode weak_eek +KPP +EEK -EEK'.

weak_eek_z : weak_eek KPP z_ek z_ek.

weak_eek_s : weak_eek KPP (s_ek EEK) (s_ek EEK')
              <- weak_eek KPP EEK EEK'.

weak_eek_pair : weak_eek KPP (pair_ek EEK1 EEK2) (pair_ek EEK1' EEK2')
                 <- weak_eek KPP EEK1 EEK1'
                 <- weak_eek KPP EEK2 EEK2'.

weak_eek_fst : weak_eek KPP (fst_ek EEK) (fst_ek EEK')
                <- weak_eek KPP EEK EEK'.

weak_eek_snd : weak_eek KPP (snd_ek EEK) (snd_ek EEK')
                <- weak_eek KPP EEK EEK'.

weak_eek_lam : weak_eek KPP (lam_ek EEK) (lam_ek EEK')
                <- ({x} ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
                        weak_eek KPP (EEK x) (EEK' x)).

weak_eek_app : weak_eek KPP (app_ek EEK1 EEK2) (app_ek EEK1' EEK2')
                <- weak_eek KPP EEK1 EEK1'
                <- weak_eek KPP EEK2 EEK2'.

weak_eek_case : weak_eek KPP
                         (case_ek EEK EEK1 EEK2)
                         (case_ek EEK' EEK1' EEK2')
                 <- weak_eek KPP EEK EEK'
                 <- weak_eek KPP EEK1 EEK1'
                 <- ({x} ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
                         weak_eek KPP (EEK2 x) (EEK2' x)).

weak_eek_fix : weak_eek KPP (fix_ek EEK) (fix_ek EEK')
                <- ({x} ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
                        weak_eek KPP (EEK x) (EEK' x)).

weak_eek_box : weak_eek KPP (box_ek EEK) (box_ek EEK')
                <- ({w} weak_eek KPP (EEK w) (EEK' w)).

weak_eek_let_box :
  weak_eek KPP (let_box_ek EEK1 EEK2) (let_box_ek EEK1' EEK2')
   <- weak_eek KPP EEK1 EEK1'
   <- ({u} ({w} {k} {k'} {kpp} weak_eek kpp (u w k) (u w k')) ->
           weak_eek KPP (EEK2 u) (EEK2' u)).

weak_eek_lab_exp : weak_eek KPP ((lab_exp LK) W) ((lab_exp LK') W)
    <- weak_lab_k KPP LK LK'.

%block we_b1 : some {W : world} {T : tp}
```

```
               block {x : {k : ktx} eek W k T}
                     {wep : {k : ktx} {k' : ktx} {kpp : ktx_prefix k k'}
                            weak_eek kpp (x k) (x k')}.
%block we_b2 : block {w : world}.
%block we_b3 : some {T : tp}
               block {u : {w : world} {k : ktx} eek w k T}
                     {wep : {w : world}
                            {k : ktx} {k' : ktx} {kpp : ktx_prefix k k'}
                            weak_eek kpp (u w k) (u w k')}.

weak_sequence : ktx_prefix K K' -> sequence W K C -> sequence W K' C -> type.
%mode weak_sequence +KPP +S -S.
weak_sequence_0 : weak_sequence KPP sequence_0 sequence_0.
weak_sequence_n : weak_sequence KPP (sequence_n EEK S) (sequence_n EEK' S')
                   <- weak_eek KPP EEK EEK'
                   <- weak_sequence KPP S S'.

drop_stage : eek W (ktx_n K ctx_0) T -> eek W K T -> type.
%mode drop_stage +EEK -EEK'.
drop_stage_z : drop_stage z_ek z_ek.

drop_stage_s : drop_stage (s_ek EEK) (s_ek EEK')
                <- drop_stage EEK EEK'.

drop_stage_pair : drop_stage (pair_ek EEK1 EEK2) (pair_ek EEK1' EEK2')
                   <- drop_stage EEK1 EEK1'
                   <- drop_stage EEK2 EEK2'.

drop_stage_fst : drop_stage (fst_ek EEK) (fst_ek EEK')
                  <- drop_stage EEK EEK'.

drop_stage_snd : drop_stage (snd_ek EEK) (snd_ek EEK')
                  <- drop_stage EEK EEK'.

drop_stage_lam : drop_stage (lam_ek EEK) (lam_ek EEK')
                  <- ({x} ({k} drop_stage (x (ktx_n k ctx_0)) (x k)) ->
                          drop_stage (EEK x) (EEK' x)).

drop_stage_app : drop_stage (app_ek EEK1 EEK2) (app_ek EEK1' EEK2')
                  <- drop_stage EEK1 EEK1'
                  <- drop_stage EEK2 EEK2'.

drop_stage_case : drop_stage (case_ek EEK EEK1 EEK2) (case_ek EEK' EEK1' EEK2')
                   <- drop_stage EEK EEK'
                   <- drop_stage EEK1 EEK1'
                   <- ({x} ({k} drop_stage(x (ktx_n k ctx_0)) (x k)) ->
                           drop_stage (EEK2 x) (EEK2' x)).

drop_stage_fix : drop_stage (fix_ek EEK) (fix_ek EEK')
                  <- ({x} ({k} drop_stage (x (ktx_n k ctx_0)) (x k)) ->
                          drop_stage (EEK x) (EEK' x)).

drop_stage_box : drop_stage (box_ek EEK) (box_ek EEK')
                  <- ({w} drop_stage (EEK w) (EEK' w)).

drop_stage_let_box : drop_stage (let_box_ek EEK1 EEK2) (let_box_ek EEK1' EEK2')
                      <- drop_stage EEK1 EEK1'
                      <- ({u} ({w} {k} drop_stage (u w (ktx_n k ctx_0))
                                                  (u w k)) ->
                                drop_stage (EEK2 u) (EEK2' u)).

drop_stage_lab_exp : drop_stage ((lab_exp (lab_k_m ctx_0 LK)) W)
                                ((lab_exp LK) W).

%block ds_b1 : some {W : world} {T : tp}
               block {x : {k : ktx} eek W k T}
                     {dsp : {k : ktx} drop_stage (x (ktx_n k ctx_0)) (x k)}.
%block ds_b2 : block {w : world}.
%block ds_b3 : some {T : tp}
               block {u : {w : world} {k : ktx} eek w k T}
                     {dsp : {w : world} {k : ktx}
                            drop_stage (u w (ktx_n k ctx_0)) (u w k)}.
```

```
% -- Translation of an expression list into let-box-bindings:
repl_sequence : eek W (ktx_n K C) T -> sequence W K C -> eek W K T -> type.
%name repl_sequence RSP.
%mode repl_sequence +EEK +S -EEK'.

repl_sequence_0 : repl_sequence EEK sequence_0 EEK'
                   <- drop_stage EEK EEK'.

repl_sequence_n :
  repl_sequence EEK (sequence_n EEK0 S) (let_box_ek (EEK0 ) EEK'')
   <- repl_lab EEK EEK'
   <- ({u}
       ({w} {k} {k'} {kpp} weak_eek kpp (u w k) (u w k')) ->
       ({w : world} {t0 : tp} {c : ctx} {k : ktx}
        repl_lab (u w (ktx_n k (ctx_n t0 c))) ([u'] (u w (ktx_n k c)))) ->
       ({w} {k} elim_world ([w'] u w k) (u w k)) ->
       ({w} {k} drop_stage (u w (ktx_n k ctx_0)) (u w k)) ->
       repl_sequence (EEK' u) S (EEK'' u)).

%block rs_b :
  some {T0 : tp} {T : tp}
  block {u : {w : world} {k : ktx} eek w k T}
        {wep : {w : world}
               {k : ktx} {k' : ktx} {kpp : ktx_prefix k k'}
               weak_eek kpp (u w k) (u w k')}
        {rlp : {w : world} {t0 : tp} {c : ctx} {k : ktx}
               repl_lab (u w (ktx_n k (ctx_n t0 c)))
                        ([u' : {w : world} {k : ktx} eek w k t0]
                           (u w (ktx_n k c)))}
        {elwp : {w : world} {k : ktx} elim_world ([w'] u w k) (u w k)}
        {dsp : {w : world} {k : ktx} drop_stage (u w (ktx_n k ctx_0)) (u w k)}.

% -- List of type lists:
zequence : world -> ktx -> type. %name zequence Z.
zequence_0 : zequence world_0 ktx_0.
zequence_n : zequence W (ktx_n K C)
              <- world_jump W' W
              <- zequence W' K
              <- sequence W' K C.

% -- Stripping lambda wjp from list of type lists:
elim_zequence_world_jump : ({w} world_jump W' w -> zequence W K) ->
                           zequence W K ->
                           type.
%mode elim_zequence_world_jump +Z' -Z.
elim_zequence_world_jump_0 : elim_zequence_world_jump ([w] [wjp] zequence_0)
                                                      zequence_0.
elim_zequence_world_jump_n :
  elim_zequence_world_jump ([w] [wjp] zequence_n (S* w)
                                                 (Z* w wjp)
                                                 (WJP* w wjp))
                           (zequence_n S Z WJP)
   <- elim_sequence_world S* S
   <- elim_zequence_world_jump Z* Z
   <- elim_world_jump WJP* WJP.

% -- Stripping lambda x from list of type lists:
elim_zequence_x : world_norm W ->
                  world_pred_eq W W' ->
                  (({k} eek W' k T) -> zequence W K) ->
                  zequence W K ->
                  type.
%mode elim_zequence_x +WNP +WPP +Z' -Z.
elim_zequence_x_0 : elim_zequence_x WNP WPEP ([x] zequence_0) zequence_0.
elim_zequence_x_n :
  elim_zequence_x (world_norm_n WJP WNP) WPEP
                  ([x] zequence_n (S* x) (Z* x) WJP')
                  (zequence_n S Z WJP')
   <- world_pred_unq WJP WJP' EWP
   <- eq_world_norm EWP WNP WNP'
   <- world_pred_front WJP' WPEP WPP
```

```
   <- elim_sequence_x WNP' WPP S* S
   <- world_pred_eq_front WJP' WPEP WPEP'
   <- elim_zequence_x WNP' WPEP' Z* Z.

world_eq_zequence : eq_world W W' -> zequence W' K -> zequence W K -> type.
%mode world_eq_zequence +WQ +Z -Z'.
world_eq_zequence_r : world_eq_zequence eq_world_r Z Z.

% -- Finally the translation from implicit into explicit:
tr_i~>e : world_norm W ->
          ei W T ->
          zequence W K ->
          zequence W K' ->
          ktx_prefix K K' ->
          eek W K' T ->
          type.
%name tr_i~>e TRIEP.
%mode tr_i~>e +WNP +EI +Z -Z' -KPP -EEK.

tr_i~>e_pop : world_norm W ->
              world_jump W W' ->
              pop W (code T) ->
              zequence W' K ->
              zequence W' K' ->
              ktx_prefix K K' ->
              lab_k K' T ->
              type.
%mode tr_i~>e_pop +WNP +WJP +P +Z -Z' -KPP -LK.

tr_i~>e_pop_0 : tr_i~>e_pop WNP
                            WJP2
                            (pop_0 EEI)
                            (zequence_n S Z WJP2')
                            (zequence_n S'' Z' WJP2'')
                            (ktx_prefix_n CPP KPP)
                            (lab_k_0 _ LC)
                <- tr_i~>e WNP EEI Z Z' KPP EEK
                <- weak_sequence KPP S S'
                <- sequence_end S' EEK S'' CPP LC.

tr_i~>e_pop_m : tr_i~>e_pop (world_norm_n WJP1 WNP)
                            WJP2
                            (pop_m WJP2' P)
                            (zequence_n (S : sequence W' K C) Z WJP2'')
                            (zequence_n S' Z' WJP2'')
                            (ktx_prefix_n CPP KPP)
                            (lab_k_m C LK)
                <- tr_i~>e_pop WNP WJP1 P Z Z' KPP LK
                <- ctx_prefix_refl C CPP
                <- weak_sequence KPP S S'.

tr_i~>e_z : tr_i~>e WNP z_i Z Z KPP z_ek
             <- ktx_prefix_refl K KPP.

tr_i~>e_s : tr_i~>e WNP (s_i EI) Z Z' KPP (s_ek EEK)
             <- tr_i~>e WNP EI Z Z' KPP EEK.

tr_i~>e_pair : tr_i~>e WNP (pair_i EI1 EI2) Z Z2 KPP (pair_ek EEK1' EEK2)
                <- tr_i~>e WNP EI1 Z Z1 KPP1 EEK1
                <- tr_i~>e WNP EI2 Z1 Z2 KPP2 EEK2
                <- weak_eek KPP2 EEK1 EEK1'
                <- ktx_prefix_trans KPP1 KPP2 KPP.

tr_i~>e_fst : tr_i~>e WNP (fst_i EI) Z Z' KPP (fst_ek EEK)
               <- tr_i~>e WNP EI Z Z' KPP EEK.

tr_i~>e_snd : tr_i~>e WNP (snd_i EI) Z Z' KPP (snd_ek EEK)
               <- tr_i~>e WNP EI Z Z' KPP EEK.

tr_i~>e_lam : tr_i~>e WNP (lam_i EI) (Z : zequence _ K) Z' KPP (lam_ek EEK)
               <- ({y} {x}
                   ({wnp} {k} {z} {kpp}
```

```
                    tr_i~>e wnp y z z kpp (x k)
                     <- ktx_prefix_refl k kpp) ->
                   ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
                   ({t} {c} {k} repl_lab (x (ktx_n k (ctx_n t c)))
                                         ([u] x (ktx_n k c))) ->
                   ({k} elim_world ([w] x k) (x k)) ->
                   ({k} drop_stage (x (ktx_n k ctx_0)) (x k)) ->
                   (tr_i~>e WNP (EI y) Z (Z'* x) KPP (EEK x)))
               <- elim_zequence_x WNP world_pred_eq_0 Z'* Z'.

tr_i~>e_app : tr_i~>e WNP (app_i EI1 EI2) Z Z2 KPP (app_ek EEK1' EEK2)
               <- tr_i~>e WNP EI1 Z Z1 KPP1 EEK1
               <- tr_i~>e WNP EI2 Z1 Z2 KPP2 EEK2
               <- weak_eek KPP2 EEK1 EEK1'
               <- ktx_prefix_trans KPP1 KPP2 KPP.

tr_i~>e_case :
  tr_i~>e WNP (case_i EI EI1 EI2) Z Z3 KPP (case_ek EEK'' EEK1' EEK2)
   <- tr_i~>e WNP EI Z Z1 KPP1 EEK
   <- tr_i~>e WNP EI1 Z1 Z2 KPP2 EEK1
   <- ({y} {x}
       ({wnp} {k} {z} {kpp}
        tr_i~>e wnp y z z kpp (x k) <- ktx_prefix_refl k kpp) ->
       ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
       ({t} {c} {k} repl_lab (x (ktx_n k (ctx_n t c))) ([u] x (ktx_n k c))) ->
       ({k} elim_world ([w] x k) (x k)) ->
       ({k} drop_stage (x (ktx_n k ctx_0)) (x k)) ->
       (tr_i~>e WNP (EI2 y) Z2 (Z3* x) KPP3 (EEK2 x)))
   <- elim_zequence_x WNP world_pred_eq_0 Z3* Z3
   <- ktx_prefix_trans KPP1 KPP2 KPP2'
   <- ktx_prefix_trans KPP2' KPP3 KPP
   <- weak_eek KPP2 EEK EEK'
   <- weak_eek KPP3 EEK' EEK''
   <- weak_eek KPP3 EEK1 EEK1'.

tr_i~>e_fix : tr_i~>e WNP (fix_i EI) Z Z' KPP (fix_ek EEK)
               <- ({y} {x}
                   ({wnp} {k} {z} {kpp}
                    tr_i~>e wnp y z z kpp (x k) <- ktx_prefix_refl k kpp) ->
                   ({k} {k'} {kpp} weak_eek kpp (x k) (x k')) ->
                   ({t} {c} {k} repl_lab (x (ktx_n k (ctx_n t c)))
                                         ([u] x (ktx_n k c))) ->
                   ({k} elim_world ([w] x k) (x k)) ->
                   ({k} drop_stage (x (ktx_n k ctx_0)) (x k)) ->
                   (tr_i~>e WNP (EI y) Z (Z'* x) KPP (EEK x)))
               <- elim_zequence_x WNP world_pred_eq_0 Z'* Z'.

tr_i~>e_box : tr_i~>e WNP (box_i EI) (Z : zequence _ K) Z' KPP EEK'
               <- ({w} {wjp : world_jump W w}
                   tr_i~>e (world_norm_n wjp WNP)
                           (EI w wjp)
                           (zequence_n sequence_0 Z wjp)
                           (zequence_n (S** w) (Z'** w wjp) (WJP' w wjp))
                           (ktx_prefix_n CPP KPP)
                           (EEK w))
               <- ({w} {wjp : world_jump W w}
                    world_pred_unq wjp (WJP' w wjp) WEP)
               <- elim_sequence_world S** S*
               <- elim_zequence_world_jump Z'** Z'*
               <- world_eq_zequence WEP Z'* Z'
               <- world_eq_sequence WEP S* S
               <- repl_sequence (box_ek EEK) S EEK'.

tr_i~>e_unbox_0 : tr_i~>e WNP
                          (unbox (pop_0 EI))
                          Z
                          Z'
                          KPP
                          (let_box_ek EEK ([u] (u W K')))
                   <- tr_i~>e WNP EI Z Z' KPP EEK.

tr_i~>e_unbox_m : tr_i~>e (world_norm_n WJP WNP)
```

```
                        (unbox (pop_m WJP P))
                        Z
                        Z'
                        KPP
                        ((lab_exp LK) W)
                  <- tr_i~>e_pop WNP WJP P Z Z' KPP LK.

%block trie_b1 :
  some {W : world} {WNP : world_norm W} {T : tp}
  block {y : ei W T}
        {x : {k : ktx} eek W k T}
        {triep : {wnp : world_norm W}
                 {k : ktx}
                 {z : zequence W k}
                 {kpp : ktx_prefix k k}
                 tr_i~>e wnp y z z kpp (x k) <- ktx_prefix_refl k kpp}
        {wkp : {k : ktx} {k' : ktx} {kpp : ktx_prefix k k'}
               weak_eek kpp (x k) (x k')}
        {rlp : {t : tp} {c : ctx} {k : ktx}
               repl_lab (x (ktx_n k (ctx_n t c))) ([u] x (ktx_n k c))}
        {elwp : {k : ktx} elim_world ([w] (x k)) (x k)}
        {dsp : {k : ktx} drop_stage (x (ktx_n k ctx_0)) (x k)}.
%block trie_b2 : some {W : world}
                 block {w : world} {wjp : world_jump W w}.

%worlds ()
        (ctx_prefix_refl _ _)
        (ktx_prefix_refl _ _)
        (ctx_prefix_trans _ _ _)
        (ktx_prefix_trans _ _ _).
%worlds (elw_b1 | elw_b2 | elw_b3 | rs_b | trie_b1 | trie_b2)
        (elim_world _ _).
%worlds (we_b1 | we_b2 | we_b3 | rs_b)
        (weak_lab_c _ _ _)
        (weak_lab_k _ _ _)
        (weak_eek _ _ _)
        (weak_sequence _ _ _).
%worlds (rl_b1 | rl_b2 | rl_b3 | rs_b | trie_b1 | trie_b2)
        (repl_lab _ _).
%worlds (rs_b | trie_b1 | trie_b2 | ds_b1 | ds_b2 | ds_b3)
        (drop_stage _ _).
%worlds (rs_b | trie_b1 | trie_b2)
        (repl_sequence _ _ _).
%worlds (elx_b1 | elx_b2 | elx_b3 | trie_b1 | trie_b2)
        (void_eek _ _)
        (elim_x _ _ _ _)
        (world_pred_unq _ _ _)
        (eq_world_pred _ _ _)
        (world_pred_eq_front _ _ _)
        (world_pred_front _ _ _)
        (world_pred_irrefl _ _ _).
%worlds (trie_b1 | trie_b2)
        (eq_world_norm _ _ _)
        (sequence_end _ _ _ _ _)
        (world_eq_sequence _ _ _)
        (world_eq_zequence _ _ _)
        (elim_world_jump _ _)
        (elim_sequence_x _ _ _ _)
        (elim_zequence_x _ _ _ _)
        (elim_sequence_world _ _)
        (elim_zequence_world_jump _ _)
        (tr_i~>e_pop _ _ _ _ _ _ _)
        (tr_i~>e _ _ _ _ _ _).

%total C (ctx_prefix_refl C _).
%total K (ktx_prefix_refl K _).
%total CPP (ctx_prefix_trans CPP _ _).
%total KPP (ktx_prefix_trans KPP _ _).
%total EEK (elim_world EEK _).
%total LC (weak_lab_c _ LC _).
%total LK (weak_lab_k _ LK _).
%total EEK (weak_eek _ EEK _).
```

```
%total S (weak_sequence _ S _).
%total EEK (repl_lab EEK _).
%total EEK (drop_stage EEK _).
%total S (repl_sequence _ S _).
%total {} (void_eek _ _).
%total {} (eq_world_norm _ _ _).
%total {} (eq_world_pred _ _ _).
%total {} (world_pred_unq _ _ _).
%total WPEP (world_pred_eq_front _ WPEP _).
%total WPEP (world_pred_front _ WPEP _).
%total WNP (world_pred_irrefl WNP _ _).
%total EEK (elim_x _ _ EEK _).
%total S (sequence_end S _ _ _ _).
%total {} (world_eq_sequence _ _ _).
%total {} (world_eq_zequence _ _ _).
%total {} (elim_world_jump _ _).
%total S* (elim_sequence_x _ _ S* _).
%total Z* (elim_zequence_x _ _ Z* _).
%total S* (elim_sequence_world S* _).
%total Z* (elim_zequence_world_jump Z* _).
%total (P EI) (tr_i~>e_pop _ _ P _ _ _ _) (tr_i~>e _ EI _ _ _ _).

%query 1 * D : tr_i~>e world_norm_0 z_i zequence_0 Z KP E.
%query 1 * D : tr_i~>e world_norm_0 (lam_i [x] x) zequence_0 Z KP E.

rm_ktx: eek W ktx_0 T -> ee W T -> type.
%name rm_ktx RKP.
%mode rm_ktx +EEK -EE.

rm_ktx_z : rm_ktx z_ek z_e.

rm_ktx_s : rm_ktx (s_ek EEK) (s_e EE)
            <- rm_ktx EEK EE.

rm_ktx_pair : rm_ktx (pair_ek EEK1 EEK2) (pair_e EE1 EE2)
               <- rm_ktx EEK1 EE1
               <- rm_ktx EEK2 EE2.

rm_ktx_fst : rm_ktx (fst_ek EEK) (fst_e EE)
              <- rm_ktx EEK EE.

rm_ktx_snd : rm_ktx (snd_ek EEK) (snd_e EE)
              <- rm_ktx EEK EE.

rm_ktx_lam : rm_ktx (lam_ek EEK) (lam_e EE)
              <- ({y} {x}
                  rm_ktx (y ktx_0) x ->
                  rm_ktx (EEK y) (EE x)).

rm_ktx_app : rm_ktx (app_ek EEK1 EEK2) (app_e EE1 EE2)
              <- rm_ktx EEK1 EE1
              <- rm_ktx EEK2 EE2.

rm_ktx_case : rm_ktx (case_ek EEK EEK1 EEK2) (case_e EE EE1 EE2)
               <- rm_ktx EEK EE
               <- rm_ktx EEK1 EE1
               <- ({y} {x}
                   rm_ktx (y ktx_0) x ->
                   rm_ktx (EEK2 y) (EE2 x)).

rm_ktx_fix : rm_ktx (fix_ek EEK) (fix_e EE)
              <- ({y} {x}
                  rm_ktx (y ktx_0) x ->
                  rm_ktx (EEK y) (EE x)).

rm_ktx_box : rm_ktx (box_ek EEK) (box_e EE)
              <- {w} rm_ktx (EEK w) (EE w).

rm_ktx_let_box : rm_ktx (let_box_ek EEK1 EEK2) (let_box_e EE1 EE2)
                  <- rm_ktx EEK1 EE1
                  <- ({v} {u}
                      ({w} rm_ktx (v w ktx_0) (u w)) ->
```

```
                    rm_ktx (EEK2 v) (EE2 u)).

%block rk_b1 : some {W : world} {T : tp}
               block {y : {k} eek W k T} {x : ee W T}
                     {rkp : rm_ktx (y ktx_0) x}.
%block rk_b2 : block {w : world}.
%block rk_b3 : some {T : tp}
               block {v : {w : world} {k : ktx} eek w k T}
                     {u : {w : world} ee w T}
                     {rkp : {w} rm_ktx (v w ktx_0) (u w)}.
%worlds (rk_b1 | rk_b2 | rk_b3) (rm_ktx _ _).
%total EEK (rm_ktx EEK _).

tr_i~>e_final : ei world_0 T -> ee world_0 T -> type.
%mode tr_i~>e_final +EI -EE.
tr_i~>e_final_0 : tr_i~>e_final EI EE
                  <- tr_i~>e world_norm_0 EI
                                          zequence_0
                                          zequence_0
                                          ktx_prefix_0
                                          EEK
                  <- rm_ktx EEK EE.
%worlds () (tr_i~>e_final _ _).
%total {} (tr_i~>e_final _ _).
```

A.5 circle.elf

The file circle.elf contains the Twelf codes concerning the language Mini-ML$^{\bigcirc}$.

```
%%% Temporal Mini-ML

% --- World number

wumber : type. %name wumber W.

wumber_0 : wumber.
wumber_n : wumber -> wumber.

% --- Temporal types

tp_t : type. %name tp_t TT.

int_t : tp_t.
product_t : tp_t -> tp_t -> tp_t.
arrow_t : tp_t -> tp_t -> tp_t.
circle : tp_t -> tp_t.

% --- Temporal expressions

et : wumber -> tp_t -> type. %name et ET.

z_t : et W int_t.
s_t : et W int_t -> et W int_t.

pair_t : et W TT1 -> et W TT2 -> et W (product_t TT1 TT2).
fst_t : et W (product_t TT1 TT2) -> et W TT1.
snd_t : et W (product_t TT1 TT2) -> et W TT2.

lam_t : (et W TT1 -> et W TT2) -> et W (arrow_t TT1 TT2).
app_t : et W (arrow_t TT1 TT2) -> et W TT1 -> et W TT2.

case_t : et W int_t -> et W TT -> (et W int_t -> et W TT) -> et W TT.

fix_t : (et W TT -> et W TT) -> et W TT.

next : et (wumber_n W) T -> et W (circle T).
prev : et W (circle T) -> et (wumber_n W) T.
```

```
% --- Equality of temporal expressions

eq_t : et W TT -> et W' TT' -> type. %name eq_t QP.
eq_t_r : eq_t ET ET.

eq_t_sym : eq_t EE EE' -> eq_t EE' EE -> type. %name eq_t_sym QP.
%mode eq_t_sym +QP -QP'.
eq_t_sym_r : eq_t_sym eq_t_r eq_t_r.

eq_t_hole : eq_t ET ET' ->
            {H : et W TT -> et W' TT'}
            eq_t (H ET) (H ET') ->
            type.
%name eq_t_hole QP.
%mode eq_t_hole +QP +H -QP'.
eq_t_hole_r : eq_t_hole eq_t_r H eq_t_r.

eq_t_s : eq_t ET ET' -> eq_t (s_t ET) (s_t ET') -> type. %name eq_t_s QP.
%mode eq_t_s +QP -QP'.
eq_t_s_r : eq_t_s eq_t_r eq_t_r.

eq_t_s_b : eq_t (s_t ET) (s_t ET') -> eq_t ET ET' -> type. %name eq_t_s_b QP.
%mode eq_t_s_b +QP -QP'.
eq_t_s_b_r : eq_t_s_b eq_t_r eq_t_r.

neq_t_zs : eq_t (z_t : et W int_t) ((s_t ET) : et W int_t) ->
           eq_t ET1 ET2 ->
           type.
%name neq_t_zs QP.
%mode +{W : wumber} +{ET : et W int_t}
      +{QP : eq_t z_t (s_t ET)}
      +{W1 : wumber} +{TT1 : tp_t} +{ET1 : et W1 TT1}
      +{W2 : wumber} +{TT2 : tp_t} +{ET2 : et W2 TT2}
      -{QP' : eq_t ET1 ET2}
      neq_t_zs QP QP'.

eq_t_pair : eq_t ET1 ET1' ->
            eq_t ET2 ET2' ->
            eq_t (pair_t ET1 ET2) (pair_t ET1' ET2') ->
            type.
%name eq_t_pair QP.
%mode eq_t_pair +QP1 +QP2 -QP.
eq_t_pair_r : eq_t_pair eq_t_r eq_t_r eq_t_r.

eq_t_pair_1 : eq_t (pair_t ET1 ET2) (pair_t ET1' ET2') ->
              eq_t ET1 ET1' ->
              type.
%name eq_t_pair_1 QP.
%mode eq_t_pair_1 +QP -QP1.
eq_t_pair_1_r : eq_t_pair_1 eq_t_r eq_t_r.

eq_t_pair_2 : eq_t (pair_t ET1 ET2) (pair_t ET1' ET2') ->
              eq_t ET2 ET2' ->
              type.
%name eq_t_pair_2 QP.
%mode eq_t_pair_2 +QP -QP2.
eq_t_pair_2_r : eq_t_pair_2 eq_t_r eq_t_r.

eq_t_fst : eq_t ET ET' -> eq_t (fst_t ET) (fst_t ET') -> type.
%name eq_t_fst QP.
%mode eq_t_fst +QP -QP2.
eq_t_fst_r : eq_t_fst eq_t_r eq_t_r.

eq_t_snd : eq_t ET ET' -> eq_t (snd_t ET) (snd_t ET') -> type.
%name eq_t_snd QP.
%mode eq_t_snd +QP -QP'.
eq_t_snd_r : eq_t_snd eq_t_r eq_t_r.

eq_t_lam : ({x} eq_t (ET x) (ET' x)) -> eq_t (lam_t ET) (lam_t ET') -> type.
%mode eq_t_lam +QP -QP'.
eq_t_lam_r : eq_t_lam ([x] eq_t_r) eq_t_r.
```

```
eq_t_lam_b : eq_t (lam_t ET) (lam_t ET') ->
             eq_t X X' ->
             eq_t (ET X) (ET' X') ->
             type.
%name eq_t_lam_b QP.
%mode eq_t_lam_b +QP1 +QP2 -QP3.
eq_t_lam_b_r : eq_t_lam_b eq_t_r eq_t_r eq_t_r.

eq_t_app : eq_t ET1 ET1' ->
           eq_t ET2 ET2' ->
           eq_t (app_t ET1 ET2) (app_t ET1' ET2') ->
           type.
%mode eq_t_app +QP1 +QP2 -QP.
eq_t_app_r : eq_t_app eq_t_r eq_t_r eq_t_r.

eq_t_case : eq_t ET ET' ->
            eq_t ET1 ET1' ->
            ({x} eq_t (ET2 x) (ET2' x)) ->
            eq_t (case_t ET ET1 ET2) (case_t ET' ET1' ET2') ->
            type.
%mode eq_t_case +QP +QP1 +QP2 -QP'.
eq_t_case_r : eq_t_case eq_t_r eq_t_r ([x] eq_t_r) eq_t_r.

eq_t_fix : ({x} eq_t (ET x) (ET' x)) -> eq_t (fix_t ET) (fix_t ET') -> type.
%mode eq_t_fix +QP -QP'.
eq_t_fix_r : eq_t_fix ([x] eq_t_r) eq_t_r.

eq_t_next : eq_t ET ET' -> eq_t (next ET) (next ET') -> type.
%mode eq_t_next +QP -QP'.
eq_t_next_r : eq_t_next eq_t_r eq_t_r.

eq_t_next_b : eq_t (next ET) (next ET') -> eq_t ET ET' -> type.
%mode eq_t_next_b +QP -QP'.
eq_t_next_b_r : eq_t_next_b eq_t_r eq_t_r.

eq_t_prev : eq_t ET ET' -> eq_t (prev ET) (prev ET') -> type.
%mode eq_t_prev +QP -QP'.
eq_t_prev_r : eq_t_prev eq_t_r eq_t_r.

% --- Temporal values

val_t : {W} et W TT -> type. %name val_t VTP.
%mode val_t +W +ET.

val_t_z : val_t W z_t.

val_t_s : val_t W (s_t VT)
          <- val_t W VT.

val_t_pair : val_t W (pair_t VT1 VT2)
             <- val_t W VT1
             <- val_t W VT2.

val_t_fst_>0 : val_t (wumber_n W) (fst_t VT)
               <- val_t (wumber_n W) VT.

val_t_snd_>0 : val_t (wumber_n W) (snd_t VT)
               <- val_t (wumber_n W) VT.

val_t_lam_0 : val_t wumber_0 (lam_t ET).

val_t_lam_>0 : val_t (wumber_n W) (lam_t VT)
               <- ({x} val_t (wumber_n W) x -> val_t (wumber_n W) (VT x)).

val_t_app_>0 : val_t (wumber_n W) (app_t VT1 VT2)
               <- val_t (wumber_n W) VT1
               <- val_t (wumber_n W) VT2.

val_t_case_>0 : val_t (wumber_n W) (case_t VT VT1 VT2)
                <- val_t (wumber_n W) VT
                <- val_t (wumber_n W) VT1
                <- ({x} val_t (wumber_n W) x -> val_t (wumber_n W) (VT2 x)).
```

```
val_t_fix_>0 : val_t (wumber_n W) (fix_t VT)
               <- ({x} val_t (wumber_n W) x -> val_t (wumber_n W) (VT x)).

val_t_next : val_t W (next VT)
             <- val_t (wumber_n W) VT.

val_t_prev_>0 : val_t (wumber_n W) (prev VT)
                <- val_t W VT.

% --- Operational semantics for temporal expressions

% -- Big-step evaluation rules for temporal expressions:
eval_t : {W} et W TT -> et W TT -> type. %name eval_t ETP.
%mode eval_t +W +ET -VT.

eval_t_z : eval_t W z_t z_t.

eval_t_s : eval_t W (s_t ET) (s_t VT)
           <- eval_t W ET VT.

eval_t_pair : eval_t W (pair_t ET1 ET2) (pair_t VT1 VT2)
              <- eval_t W ET1 VT1
              <- eval_t W ET2 VT2.

eval_t_fst_0 : eval_t wumber_0 (fst_t ET) VT1
               <- eval_t wumber_0 ET (pair_t VT1 VT2).

eval_t_fst_>0 : eval_t (wumber_n W) (fst_t ET) (fst_t VT)
                <- eval_t (wumber_n W) ET VT.

eval_t_snd_0 : eval_t wumber_0 (snd_t ET) VT2
               <- eval_t wumber_0 ET (pair_t VT1 VT2).

eval_t_snd_>0 : eval_t (wumber_n W) (snd_t ET) (snd_t VT)
                <- eval_t (wumber_n W) ET VT.

eval_t_lam_0 : eval_t wumber_0 (lam_t ET) (lam_t ET).

eval_t_lam_>0 : eval_t (wumber_n W) (lam_t ET) (lam_t VT)
                <- ({x} eval_t (wumber_n W) x x ->
                        eval_t (wumber_n W) (ET x) (VT x)).

eval_t_app_0 : eval_t wumber_0 (app_t ET1 ET2) VT1
               <- eval_t wumber_0 ET1 (lam_t ET1')
               <- eval_t wumber_0 ET2 VT2
               <- eval_t wumber_0 (ET1' VT2) VT1.

eval_t_app_>0 : eval_t (wumber_n W) (app_t ET1 ET2) (app_t VT1 VT2)
                <- eval_t (wumber_n W) ET1 VT1
                <- eval_t (wumber_n W) ET2 VT2.

eval_t_case_z_0 : eval_t wumber_0 (case_t ET ET1 ET2) VT1
                  <- eval_t wumber_0 ET z_t
                  <- eval_t wumber_0 ET1 VT1.

eval_t_case_s_0 : eval_t wumber_0 (case_t ET ET1 ET2) VT2
                  <- eval_t wumber_0 ET (s_t VT)
                  <- eval_t wumber_0 (ET2 VT) VT2.

eval_t_case_>0 : eval_t (wumber_n W) (case_t ET ET1 ET2) (case_t VT VT1 VT2)
                 <- eval_t (wumber_n W) ET VT
                 <- eval_t (wumber_n W) ET1 VT1
                 <- ({x} eval_t (wumber_n W) x x ->
                         eval_t (wumber_n W) (ET2 x) (VT2 x)).

eval_t_fix_0 : eval_t wumber_0 (fix_t ET) VT
               <- eval_t wumber_0 (ET (fix_t ET)) VT.

eval_t_fix_>0 : eval_t (wumber_n W) (fix_t ET) (fix_t VT)
                <- ({x} eval_t (wumber_n W) x x ->
                        eval_t (wumber_n W) (ET x) (VT x)).
```

```
eval_t_next : eval_t W (next ET) (next VT)
               <- eval_t (wumber_n W) ET VT.

eval_t_prev_1 : eval_t (wumber_n wumber_0) (prev ET) VT
                 <- eval_t wumber_0 ET (next VT).

eval_t_prev_>1 : eval_t (wumber_n (wumber_n W)) (prev ET) (prev VT)
                  <- eval_t (wumber_n W) ET VT.

% -- Value soundness for evaluation of temporal expressions:
val_sound_t : eval_t W ET VT -> val_t W VT -> type. %name val_sound_t VSTP.
%mode val_sound_t +ETP -VTP.

val_sound_t_z : val_sound_t eval_t_z val_t_z.

val_sound_t_s : val_sound_t (eval_t_s ETP) (val_t_s VTP)
                 <- val_sound_t ETP VTP.

val_sound_t_pair : val_sound_t (eval_t_pair ETP2 ETP1) (val_t_pair VTP2 VTP1)
                    <- val_sound_t ETP1 VTP1
                    <- val_sound_t ETP2 VTP2.

val_sound_t_fst_0 : val_sound_t (eval_t_fst_0 ETP) VTP1
                     <- val_sound_t ETP (val_t_pair VTP2 VTP1).

val_sound_t_fst_>0 : val_sound_t (eval_t_fst_>0 ETP) (val_t_fst_>0 VTP)
                      <- val_sound_t ETP VTP.

val_sound_t_snd_0 : val_sound_t (eval_t_snd_0 ETP) VTP2
                     <- val_sound_t ETP (val_t_pair VTP2 VTP1).

val_sound_t_snd_>0 : val_sound_t (eval_t_snd_>0 ETP) (val_t_snd_>0 VTP)
                      <- val_sound_t ETP VTP.

val_sound_t_lam_0 : val_sound_t eval_t_lam_0 val_t_lam_0.

val_sound_t_lam_>0 : val_sound_t (eval_t_lam_>0 ETP) (val_t_lam_>0 VTP)
                      <- ({x} {etp} {vtp} val_sound_t etp vtp ->
                                          val_sound_t (ETP x etp) (VTP x vtp)).

val_sound_t_app_0 : val_sound_t (eval_t_app_0 ETP ETP' ETP'') VTP
                     <- val_sound_t ETP VTP.

val_sound_t_app_>0 : val_sound_t (eval_t_app_>0 ETP2 ETP1)
                                 (val_t_app_>0 VTP2 VTP1)
                      <- val_sound_t ETP1 VTP1
                      <- val_sound_t ETP2 VTP2.

val_sound_t_case_z_0 : val_sound_t (eval_t_case_z_0 ETP1 ETP) VTP1
                        <- val_sound_t ETP1 VTP1.

val_sound_t_case_s_0 : val_sound_t (eval_t_case_s_0 ETP2 ETP) VTP2
                        <- val_sound_t ETP2 VTP2.

val_sound_t_case_>0 :
  val_sound_t (eval_t_case_>0 ETP2 ETP1 ETP) (val_t_case_>0 VTP2 VTP1 VTP)
   <- val_sound_t ETP VTP
   <- val_sound_t ETP1 VTP1
   <- ({x} {etp} {vtp} val_sound_t etp vtp ->
                       val_sound_t (ETP2 x etp) (VTP2 x vtp)).

val_sound_t_fix_0 : val_sound_t (eval_t_fix_0 ETP) VTP
                     <- val_sound_t ETP VTP.

val_sound_t_fix_>0 :
  val_sound_t (eval_t_fix_>0 ETP) (val_t_fix_>0 VTP)
   <- ({x} {etp} {vtp} val_sound_t etp vtp ->
                       val_sound_t (ETP x etp) (VTP x vtp)).

val_sound_t_next : val_sound_t (eval_t_next ETP) (val_t_next VTP)
                    <- val_sound_t ETP VTP.
```

```
val_sound_t_prev_1 : val_sound_t (eval_t_prev_1 ETP) VTP
                      <- val_sound_t ETP (val_t_next VTP).

val_sound_t_prev_>1 : val_sound_t (eval_t_prev_>1 ETP) (val_t_prev_>0 VTP)
                       <- val_sound_t ETP VTP.

%block vst_b1 : some {W : wumber} {TT : tp_t}
                block {x : et (wumber_n W) TT}
                      {etp : eval_t (wumber_n W) x x}
                      {vtp : val_t (wumber_n W) x}
                      {vstp : val_sound_t etp vtp}.

%worlds (vst_b1) (val_sound_t _ _).
%total ETP (val_sound_t ETP _).

% -- Equality of evaluation of temporal expressions:
eq_t_eval : eq_t ET ET' -> eval_t W ET VT -> eval_t W ET' VT -> type.
%name eq_t_eval QP.
%mode eq_t_eval +QP +ETP -ETP'.
eq_t_eval_r : eq_t_eval eq_t_r ETP ETP.

% -- Determinacy for evaluation of temporal expressions:
eval_det_t : eval_t W ET VT -> eval_t W ET VT' -> eq_t VT VT' -> type.
%mode eval_det_t +ETP +ETP' -QP.

eval_det_t_z : eval_det_t eval_t_z eval_t_z eq_t_r.

eval_det_t_s : eval_det_t (eval_t_s ETP) (eval_t_s ETP') QP'
                <- eval_det_t ETP ETP' QP
                <- eq_t_s QP QP'.

eval_det_t_pair : eval_det_t (eval_t_pair ETP2 ETP1)
                             (eval_t_pair ETP2' ETP1')
                             QP
                   <- eval_det_t ETP1 ETP1' QP1
                   <- eval_det_t ETP2 ETP2' QP2
                   <- eq_t_pair QP1 QP2 QP.

eval_det_t_fst_0 : eval_det_t (eval_t_fst_0 ETP) (eval_t_fst_0 ETP') QP1
                    <- eval_det_t ETP ETP' QP
                    <- eq_t_pair_1 QP QP1.

eval_det_t_fst_>0 : eval_det_t (eval_t_fst_>0 ETP) (eval_t_fst_>0 ETP') QP'
                     <- eval_det_t ETP ETP' QP
                     <- eq_t_fst QP QP'.

eval_det_t_snd_0 : eval_det_t (eval_t_snd_0 ETP) (eval_t_snd_0 ETP') QP2
                    <- eval_det_t ETP ETP' QP
                    <- eq_t_pair_2 QP QP2.

eval_det_t_snd_>0 : eval_det_t (eval_t_snd_>0 ETP) (eval_t_snd_>0 ETP') QP'
                     <- eval_det_t ETP ETP' QP
                     <- eq_t_snd QP QP'.

eval_det_t_lam_0 : eval_det_t eval_t_lam_0 eval_t_lam_0 eq_t_r.

eval_det_t_lam_>0 :
  eval_det_t (eval_t_lam_>0 ETP) (eval_t_lam_>0 ETP') QP'
   <- ({x} {etp} eval_det_t etp etp eq_t_r ->
                 eval_det_t (ETP x etp) (ETP' x etp) (QP x))
   <- eq_t_lam QP QP'.

eval_det_t_app_0 : eval_det_t (eval_t_app_0 ETP3 ETP2 ETP1)
                              (eval_t_app_0 ETP3' ETP2' ETP1')
                              QP''
                    <- eval_det_t ETP1 ETP1' QP1
                    <- eval_det_t ETP2 ETP2' QP2
                    <- eq_t_lam_b QP1 QP2 QP
                    <- eq_t_sym QP QP'
                    <- eq_t_eval QP' ETP3' ETP3''
                    <- eval_det_t ETP3 ETP3'' QP''.
```

```
eval_det_app_>0 : eval_det_t (eval_t_app_>0 ETP2 ETP1)
                             (eval_t_app_>0 ETP2' ETP1')
                             QP
                  <- eval_det_t ETP1 ETP1' QP1
                  <- eval_det_t ETP2 ETP2' QP2
                  <- eq_t_app QP1 QP2 QP.

eval_det_t_case_zz_0 : eval_det_t (eval_t_case_z_0 ETP1 ETP)
                                  (eval_t_case_z_0 ETP1' ETP')
                                  QP1
                       <- eval_det_t ETP1 ETP1' QP1.

eval_det_t_case_zs_0 : eval_det_t (eval_t_case_z_0 ETP1 ETP)
                                  (eval_t_case_s_0 ETP1' ETP')
                                  QP'
                       <- eval_det_t ETP ETP' QP
                       <- neq_t_zs QP QP'.

eval_det_t_case_sz_0 : eval_det_t (eval_t_case_s_0 ETP1 ETP)
                                  (eval_t_case_z_0 ETP1' ETP')
                                  QP''
                       <- eval_det_t ETP ETP' QP
                       <- eq_t_sym QP QP'
                       <- neq_t_zs QP' QP''.

eval_det_t_case_>0 :
  eval_det_t (eval_t_case_>0 ETP2 ETP1 ETP)
             (eval_t_case_>0 ETP2' ETP1' ETP')
              QP'
   <- eval_det_t ETP ETP' QP
   <- eval_det_t ETP1 ETP1' QP1
   <- ({x} {etp} eval_det_t etp etp eq_t_r ->
                 eval_det_t (ETP2 x etp) (ETP2' x etp) (QP2 x))
   <- eq_t_case QP QP1 QP2 QP'.

eval_det_t_case_ss_0 :
  eval_det_t ((eval_t_case_s_0 ETP1 ETP) : eval_t wumber_0 (case_t _ _ ET2) _)
             (eval_t_case_s_0 ETP1' ETP')
             QP''''
   <- eval_det_t ETP ETP' QP
   <- eq_t_s_b QP QP'
   <- eq_t_sym QP' QP''
   <- eq_t_hole QP'' ET2 QP'''
   <- eq_t_eval QP''' ETP1' ETP1''
   <- eval_det_t ETP1 ETP1'' QP''''.

eval_det_t_fix_0 : eval_det_t (eval_t_fix_0 ETP)
                              (eval_t_fix_0 ETP')
                              QP
                   <- eval_det_t ETP ETP' QP.

eval_det_t_fix_>0 :
  eval_det_t (eval_t_fix_>0 ETP) (eval_t_fix_>0 ETP') QP'
   <- ({x} {etp} eval_det_t etp etp eq_t_r ->
                 eval_det_t (ETP x etp) (ETP' x etp) (QP x))
   <- eq_t_fix QP QP'.

eval_det_t_next : eval_det_t (eval_t_next ETP) (eval_t_next ETP') QP'
                  <- eval_det_t ETP ETP' QP
                  <- eq_t_next QP QP'.

eval_det_t_prev_1 : eval_det_t (eval_t_prev_1 ETP) (eval_t_prev_1 ETP') QP'
                    <- eval_det_t ETP ETP' QP
                    <- eq_t_next_b QP QP'.

eval_det_t_prev_>1 : eval_det_t (eval_t_prev_>1 ETP) (eval_t_prev_>1 ETP') QP'
                     <- eval_det_t ETP ETP' QP
                     <- eq_t_prev QP QP'.

%block eut_b1 : some {W : wumber} {TT : tp_t}
                block {x : et (wumber_n W) TT}
```

```
                    {etp : eval_t (wumber_n W) x x}
                    {eutp : eval_det_t etp etp eq_t_r}.

%worlds (eut_b1)
        (eq_t_sym _ _)
        (eq_t_hole _ _ _)
        (eq_t_s _ _)
        (eq_t_s_b _ _)
        (neq_t_zs _ _)
        (eq_t_pair _ _ _)
        (eq_t_pair_1 _ _)
        (eq_t_pair_2 _ _)
        (eq_t_fst _ _)
        (eq_t_snd _ _)
        (eq_t_lam _ _)
        (eq_t_lam_b _ _ _)
        (eq_t_app _ _ _)
        (eq_t_case _ _ _ _)
        (eq_t_fix _ _)
        (eq_t_next _ _)
        (eq_t_next_b _ _)
        (eq_t_prev _ _)
        (eq_t_eval _ _ _)
        (eval_det_t _ _ _).

%total {} (eq_t_sym _ _).
%total {} (eq_t_hole _ _ _).
%total {} (eq_t_s _ _).
%total {} (eq_t_s_b _ _).
%total {} (neq_t_zs _ _).
%total {} (eq_t_pair _ _ _).
%total {} (eq_t_pair_1 _ _).
%total {} (eq_t_pair_2 _ _).
%total {} (eq_t_fst _ _).
%total {} (eq_t_snd _ _).
%total {} (eq_t_lam _ _).
%total {} (eq_t_lam_b _ _ _).
%total {} (eq_t_app _ _ _).
%total {} (eq_t_case _ _ _ _).
%total {} (eq_t_fix _ _).
%total {} (eq_t_next _ _).
%total {} (eq_t_next_b _ _).
%total {} (eq_t_prev _ _).
%total {} (eq_t_eval _ _ _).
%total ETP (eval_det_t ETP _ _).
```

A.6 contextual_box.elf

The file contextual_box.elf contains the Twelf codes concerning the language Mini-ML$_{\mathrm{co}}$.

```
%%% Contextual modal Mini-ML

% --- Contextual types

tp_c : type. %name tp_c TC.

ctx_c : type. %name ctx_c CC.
ctx_c_0 : ctx_c.
ctx_c_n : tp_c -> ctx_c -> ctx_c.

int_c : tp_c.
product_c : tp_c -> tp_c -> tp_c.
arrow_c : tp_c -> tp_c -> tp_c.
code_c : ctx_c -> tp_c -> tp_c.

% --- Contextual expressions

ec : world -> tp_c -> type. %name ec EC.
```

```
z_c : ec W int_c.
s_c : ec W int_c -> ec W int_c.

pair_c : ec W TC1 -> ec W TC2 -> ec W (product_c TC1 TC2).
fst_c : ec W (product_c TC1 TC2) -> ec W TC1.
snd_c : ec W (product_c TC1 TC2) -> ec W TC2.

lam_c : (ec W TC1 -> ec W TC2) -> ec W (arrow_c TC1 TC2).
app_c : ec W (arrow_c TC1 TC2) -> ec W TC1 -> ec W TC2.

case_c : ec W int_c -> ec W TC -> (ec W int_c -> ec W TC) -> ec W TC.

fix_c : (ec W TC -> ec W TC) -> ec W TC.

boxarg : world -> ctx_c -> tp_c -> type. %name boxarg B.
boxarg_0 : ec W TC -> boxarg W ctx_c_0 TC.
boxarg_n : (ec W TC' -> boxarg W CC TC) -> boxarg W (ctx_c_n TC' CC) TC.

box_c : {W} ({w} boxarg w CC TC) -> ec W (code_c CC TC).

uvar : ctx_c -> tp_c -> type. %name uvar U.

let_box_c : ec W (code_c CC TC1) -> (uvar CC TC1 -> ec W TC2) -> ec W TC2.

sequence_c : world -> ctx_c -> type. %name sequence_c SC.
sequence_c_0 : sequence_c W ctx_c_0.
sequence_c_n : ec W TC -> sequence_c W CC -> sequence_c W (ctx_c_n TC CC).

clo : uvar CC TC -> sequence_c W CC -> ec W TC.

% --- Equality of contextual expressions and types

eq_c : ec W TC -> ec W' TC' -> type. %name eq_c QP.
%mode eq_c +EC +EC'.
eq_c_r : eq_c EC EC.

eq_c_sym : eq_c EC EC' -> eq_c EC' EC -> type. %name eq_c_sym QP.
%mode eq_c_sym +QP -QP'.
eq_c_sym_r : eq_c_sym eq_c_r eq_c_r.

eq_c_hole : eq_c EC EC' ->
            {H : ec W TC -> ec W' TC'}
            eq_c (H EC) (H EC') ->
            type.
%name eq_c_hole QP.
%mode eq_c_hole +QP +H -QP'.
eq_c_hole_r : eq_c_hole eq_c_r H eq_c_r.

eq_c_s : eq_c EC EC' -> eq_c (s_c EC) (s_c EC') -> type. %name eq_c_s QP.
%mode eq_c_s +QP -QP'.
eq_c_s_r : eq_c_s eq_c_r eq_c_r.

eq_c_s_b : eq_c (s_c EC) (s_c EC') -> eq_c EC EC' -> type. %name eq_c_s_b QP.
%mode eq_c_s_b +QP -QP'.
eq_c_s_b_r : eq_c_s_b eq_c_r eq_c_r.

neq_c_zs : eq_c (z_c : ec W int_c) ((s_c EC) : ec W int_c) ->
           eq_c EC1 EC2 ->
           type.
%name neq_c_zs QP.
%mode +{W : world} +{EC : ec W int_c}
      +{QP : eq_c z_c (s_c EC)}
      +{W1 : world} +{TC1 : tp_c} +{EC1 : ec W1 TC1}
      +{W2 : world} +{TC2 : tp_c} +{EC2 : ec W2 TC2}
      -{QP' : eq_c EC1 EC2}
      neq_c_zs QP QP'.

eq_c_pair : eq_c EC1 EC1' ->
            eq_c EC2 EC2' ->
            eq_c (pair_c EC1 EC2) (pair_c EC1' EC2') ->
            type.
```

```
%name eq_c_pair QP.
%mode eq_c_pair +QP1 +QP2 -QP.
eq_c_pair_r : eq_c_pair eq_c_r eq_c_r eq_c_r.

eq_c_pair_b1 : eq_c (pair_c EC1 EC2) (pair_c EC1' EC2') ->
               eq_c EC1 EC1' ->
               type.
%name eq_c_pair_b1 QP.
%mode eq_c_pair_b1 +QP -QP'.
eq_c_pair_b1_r : eq_c_pair_b1 eq_c_r eq_c_r.

eq_c_pair_b2 : eq_c (pair_c EC1 EC2) (pair_c EC1' EC2') ->
               eq_c EC2 EC2' ->
               type.
%name eq_c_pair_b2 QP.
%mode eq_c_pair_b2 +QP -QP'.
eq_c_pair_b2_r : eq_c_pair_b2 eq_c_r eq_c_r.

eq_c_fst : eq_c EC EC' -> eq_c (fst_c EC) (fst_c EC') -> type.
%name eq_c_fst QP.
%mode eq_c_fst +QP -QP'.
eq_c_fst_r : eq_c_fst eq_c_r eq_c_r.

eq_c_snd : eq_c EC EC' -> eq_c (snd_c EC) (snd_c EC') -> type.
%name eq_c_snd QP.
%mode eq_c_snd +QP -QP'.
eq_c_snd_r : eq_c_snd eq_c_r eq_c_r.

eq_c_lam : ({x} eq_c (EC x) (EC' x)) -> eq_c (lam_c EC) (lam_c EC') -> type.
%name eq_c_lam QP.
%mode eq_c_lam +QP -QP'.
eq_c_lam_r : eq_c_lam ([x] eq_c_r) eq_c_r.

eq_c_lam_b : eq_c (lam_c EC1) (lam_c EC1') ->
             eq_c EC2 EC2' ->
             eq_c (EC1 EC2) (EC1' EC2') ->
             type.
%name eq_c_lam_b QP.
%mode eq_c_lam_b +QP1 +QP2 -QP3.
eq_c_lam_b_r : eq_c_lam_b eq_c_r eq_c_r eq_c_r.

eq_c_app : eq_c EC1 EC1' ->
           eq_c EC2 EC2' ->
           eq_c (app_c EC1 EC2) (app_c EC1' EC2') ->
           type.
%name eq_c_app QP.
%mode eq_c_app +QP1 +QP2 -QP.
eq_c_app_r : eq_c_app eq_c_r eq_c_r eq_c_r.

eq_c_case : eq_c EC EC' ->
            eq_c EC1 EC1' ->
            ({x} eq_c (EC2 x) (EC2' x)) ->
            eq_c (case_c EC EC1 EC2) (case_c EC' EC1' EC2') ->
            type.
%name eq_c_case QP.
%mode eq_c_case +QP +QP1 +QP2 -QP'.
eq_c_case_r : eq_c_case eq_c_r eq_c_r ([x] eq_c_r) eq_c_r.

eq_c_fix : ({x} eq_c (EC x) (EC' x)) -> eq_c (fix_c EC) (fix_c EC') -> type.
%name eq_c_fix QP.
%mode eq_c_fix +QP -QP'.
eq_c_fix_r : eq_c_fix ([x] eq_c_r) eq_c_r.

eq_c_let_box : eq_c EC1 EC1' ->
               ({u} eq_c (EC2 u) (EC2' u)) ->
               eq_c (let_box_c EC1 EC2) (let_box_c EC1' EC2') ->
               type.
%name eq_c_let_box QP.
%mode eq_c_let_box +QP1 +QP2 -QP.
eq_c_let_box_r : eq_c_let_box eq_c_r ([u] eq_c_r) eq_c_r.

eq_boxarg : boxarg W CC TC -> boxarg W CC' TC' -> type. %name eq_boxarg QP.
```

```
%mode eq_boxarg +B +B'.
eq_boxarg_r : eq_boxarg B B.

eq_sym_boxarg : (eq_boxarg B B') -> (eq_boxarg B' B) -> type.
%name eq_sym_boxarg QP.
%mode eq_sym_boxarg +QP -QP'.
eq_sym_boxarg_r : eq_sym_boxarg eq_boxarg_r eq_boxarg_r.

eq_boxarg_0 : eq_c EC EC' ->
              eq_boxarg (boxarg_0 EC) (boxarg_0 EC') ->
              type.
%name eq_boxarg_0 QP.
%mode eq_boxarg_0 +QP -QP'.
eq_boxarg_0_r : eq_boxarg_0 eq_c_r eq_boxarg_r.

eq_boxarg_n : ({x} eq_boxarg (B x) (B' x)) ->
              eq_boxarg (boxarg_n B) (boxarg_n B') ->
              type.
%name eq_boxarg_n QP.
%mode eq_boxarg_n +QP -QP'.
eq_boxarg_n_r : eq_boxarg_n ([x] eq_boxarg_r) eq_boxarg_r.

eq_c_box : {W}
           ({w} eq_boxarg (B w) (B' w)) ->
           eq_c (box_c W B) (box_c W B') ->
           type.
%name eq_c_box QP.
%mode eq_c_box +W +QP -QP'.
eq_c_box_r : eq_c_box W ([w : world] eq_boxarg_r) eq_c_r.

eq_box_boxarg : eq_c (box_c W B) (box_c W B') ->
                ({w} eq_boxarg (B w) (B' w)) ->
                type.
%name eq_box_boxarg QP.
%mode eq_box_boxarg +QP -QP'.
eq_box_boxarg_r : eq_box_boxarg eq_c_r ([w] eq_boxarg_r).

eq_c_sequence : sequence_c W CC ->
                sequence_c W' TC' ->
                type.
%name eq_c_sequence QP.
%mode eq_c_sequence +SC +SC'.
eq_c_sequence_r : eq_c_sequence SC SC.

eq_c_sequence_n : eq_c EC EC' ->
                  eq_c_sequence SC SC' ->
                  eq_c_sequence (sequence_c_n EC SC) (sequence_c_n EC' SC') ->
                  type.
%name eq_c_sequence_n QP.
%mode eq_c_sequence_n +QP +QP' -QP''.
eq_c_sequence_n_r : eq_c_sequence_n eq_c_r eq_c_sequence_r eq_c_sequence_r.

eq_clo : eq_c_sequence SC SC' ->
         eq_c (clo U SC) (clo U SC') ->
         type.
%name eq_clo QP.
%mode +{W : world} +{W' : world} +{CC : ctx_c} +{TC : tp_c}
      +{SC : sequence_c W CC} +{SC' : sequence_c W' CC}
      +{QP : eq_c_sequence SC SC'}
      +{U : uvar CC TC} -{QP' : eq_c (clo U SC) (clo U SC')}
      (eq_clo QP QP').
eq_clo_r : eq_clo eq_c_sequence_r eq_c_r.

eq_c_tp : tp_c -> tp_c -> type.
%mode eq_c_tp +TC +TC'.
eq_c_tp_r : eq_c_tp TC TC.

eq_c_tp_product : eq_c_tp TC1 TC1' ->
                  eq_c_tp TC2 TC2' ->
                  eq_c_tp (product_c TC1 TC2) (product_c TC1' TC2') ->
                  type.
%mode eq_c_tp_product +QP1 +QP2 -QP.
```

```
eq_c_tp_product_r : eq_c_tp_product eq_c_tp_r eq_c_tp_r eq_c_tp_r.
%worlds () (eq_c_tp_product _ _ _).
%total {} (eq_c_tp_product _ _ _).

eq_c_tp_arrow : eq_c_tp TC1 TC1' ->
                eq_c_tp TC2 TC2' ->
                eq_c_tp (arrow_c TC1 TC2) (arrow_c TC1' TC2') ->
                type.
%mode eq_c_tp_arrow +QP1 +QP2 -QP.
eq_c_tp_arrow_r : eq_c_tp_arrow eq_c_tp_r eq_c_tp_r eq_c_tp_r.
%worlds () (eq_c_tp_arrow _ _ _).
%total {} (eq_c_tp_arrow _ _ _).

eq_c_tp_code : eq_c_tp TC TC' ->
               eq_c_tp (code_c ctx_c_0 TC) (code_c ctx_c_0 TC') ->
               type.
%mode eq_c_tp_code +QP -QP'.
eq_c_tp_code_r : eq_c_tp_code eq_c_tp_r eq_c_tp_r.
%worlds () (eq_c_tp_code _ _).
%total {} (eq_c_tp_code _ _).

eq_c_tp_e : eq_c_tp TC TC' -> ec W TC -> ec W TC' -> type.
%mode eq_c_tp_e +QP +EC -EC'.
eq_c_tp_e_r : eq_c_tp_e QP EC EC.

% --- Contextual values

val_c : ec W TC -> type. %name val_c VCP.
%mode val_c +EC.
val_c_z : val_c z_c.
val_c_s : val_c (s_c VC)
           <- val_c VC.
val_c_pair : val_c (pair_c VC1 VC2)
              <- val_c VC1
              <- val_c VC2.
val_c_lam : val_c (lam_c EC).
val_c_box : val_c (box_c W B).
%worlds () (val_c _).

% --- Operational semantics for contextual expressions

% -- Evaluation of a closure:
eval_clo : boxarg W CC TC -> sequence_c W CC -> ec W TC -> type.
%name eval_clo ECP.
%mode eval_clo +B +SC -EC.
eval_clo_0 : eval_clo (boxarg_0 EC) sequence_c_0 EC.
eval_clo_n : eval_clo (boxarg_n EC) (sequence_c_n EC0 SC) EC'
              <- eval_clo (EC EC0) SC EC'.

eval_clo_total : {B} {SC} eval_clo B SC EC -> type.
%mode eval_clo_total +B +SC -ECP.
eval_clo_total_0 : eval_clo_total (boxarg_0 EC) sequence_c_0 eval_clo_0.
eval_clo_total_n : eval_clo_total (boxarg_n EC)
                                  (sequence_c_n EC0 SC)
                                  (eval_clo_n ECP)
                    <- eval_clo_total (EC EC0) SC ECP.

% -- Substitution of boxarg for a modal variable:
subst_boxarg : ({w} boxarg w CC TC1) ->
               (uvar CC TC1 -> ec W TC2) ->
               ec W TC2 ->
               type.
%name subst_boxarg SBP.
%mode subst_boxarg +B +EC -EC'.

subst_boxarg_boxarg : ({w} boxarg w CC TC) ->
                      (uvar CC TC -> boxarg W CC' TC') ->
                      boxarg W CC' TC' ->
                      type.
%name subst_boxarg_boxarg SBBP.
%mode subst_boxarg_boxarg +B +B' -B''.
```

```
subst_boxarg_boxarg_0 :
  subst_boxarg_boxarg B ([u] boxarg_0 (EC u)) (boxarg_0 EC')
   <- subst_boxarg B EC EC'.

subst_boxarg_boxarg_n :
  subst_boxarg_boxarg B
                      ([u] boxarg_n ([x] (B' x u)))
                      (boxarg_n B'')
   <- ({x}
       ({cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x) ->
       subst_boxarg_boxarg B (B' x) (B'' x)).

subst_boxarg_sequence : ({w} boxarg w CC TC) ->
                        (uvar CC TC -> sequence_c W CC') ->
                        (sequence_c W CC') ->
                        type.
%name subst_boxarg_sequence SBSP.
%mode subst_boxarg_sequence +B +SC -SC'.

subst_boxarg_sequence_0 :
  subst_boxarg_sequence B ([u] sequence_c_0) sequence_c_0.

subst_boxarg_sequence_n :
  subst_boxarg_sequence B
                        ([u] (sequence_c_n (EC u) (SC u)))
                        (sequence_c_n EC' SC')
   <- subst_boxarg B EC EC'
   <- subst_boxarg_sequence B SC SC'.

subst_boxarg_z : subst_boxarg B ([u] z_c) z_c.

subst_boxarg_s : subst_boxarg B ([u] s_c (EC u)) (s_c EC')
                  <- subst_boxarg B EC EC'.

subst_boxarg_pair : subst_boxarg B
                                 ([u] (pair_c (EC1 u) (EC2 u)))
                                 (pair_c EC1' EC2')
                    <- subst_boxarg B EC1 EC1'
                    <- subst_boxarg B EC2 EC2'.

subst_boxarg_fst : subst_boxarg B ([u] (fst_c (EC u))) (fst_c EC')
                    <- subst_boxarg B EC EC'.

subst_boxarg_snd : subst_boxarg B ([u] (snd_c (EC u))) (snd_c EC')
                    <- subst_boxarg B EC EC'.

subst_boxarg_lam :
  subst_boxarg B ([u] lam_c ([x] EC x u)) (lam_c EC')
   <- ({x} ({cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x) ->
           subst_boxarg B (EC x) (EC' x)).

subst_boxarg_app : subst_boxarg B ([u] app_c (EC1 u) (EC2 u)) (app_c EC1' EC2')
                    <- subst_boxarg B EC1 EC1'
                    <- subst_boxarg B EC2 EC2'.

subst_boxarg_case :
  subst_boxarg B
               ([u] (case_c (EC u) (EC1 u) ([x] EC2 x u)))
               (case_c EC' EC1' EC2')
   <- subst_boxarg B EC EC'
   <- subst_boxarg B EC1 EC1'
   <- ({x} ({cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x) ->
           subst_boxarg B (EC2 x) (EC2' x)).

subst_boxarg_fix :
  subst_boxarg B ([u] (fix_c ([x] EC x u))) (fix_c EC')
   <- ({x} ({cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x) ->
           subst_boxarg B (EC x) (EC' x)).

subst_boxarg_box : subst_boxarg B ([u] box_c W ([w] B' w u)) (box_c W B'')
                    <- ({w} subst_boxarg_boxarg B (B' w) (B'' w)).
```

```
subst_boxarg_let_box : subst_boxarg B
                                    ([u] (let_box_c (EC1 u) ([v] EC2 v u)))
                                    (let_box_c EC1' EC2')
                        <- subst_boxarg B EC1 EC1'
                        <- ({v} subst_boxarg B (EC2 v) (EC2' v)).

subst_boxarg_clo_u : subst_boxarg B ([u] (clo u (SC u))) EC
                      <- subst_boxarg_sequence B SC SC'
                      <- eval_clo (B W) SC' EC.

% The variable u is bound further to the right than U is quantified and
% thus u cannot occur freely in U...
subst_boxarg_clo_v : subst_boxarg B ([u] (clo U (SC u))) (clo U SC')
                      <- subst_boxarg_sequence B SC SC'.

%block sb_b1 :
  some {W : world} {TC : tp_c}
  block {x : ec W TC}
        {sbp : {cc : ctx_c} {tc : tp_c} {b : {w : world} boxarg w cc tc}
               subst_boxarg b ([u : uvar cc tc] x) x}.
%block sb_b2 : some {CC : ctx_c} {TC : tp_c} block {v : uvar CC TC}.
%block sb_b3 : block {w : world}.

%worlds (sb_b1 | sb_b2 | sb_b3)
        (eval_clo _ _ _)
        (subst_boxarg_sequence _ _ _)
        (subst_boxarg_boxarg _ _ _)
        (subst_boxarg _ _ _).

%total SC (eval_clo _ SC _).
%total (SC B EC)
       (subst_boxarg_sequence _ SC _)
       (subst_boxarg_boxarg _ B _)
       (subst_boxarg _ EC _).

% -- Totality of boxarg substitution:
subst_boxarg_total : {B} {EC} subst_boxarg B EC EC' -> type.
%mode subst_boxarg_total +B +EC -SBP.

subst_boxarg_boxarg_total : {B} {B'} subst_boxarg_boxarg B B' B'' -> type.
%mode subst_boxarg_boxarg_total +B +B' -SBBP.

subst_boxarg_boxarg_total_0 :
  subst_boxarg_boxarg_total B ([u] boxarg_0 (EC u)) (subst_boxarg_boxarg_0 SBP)
   <- subst_boxarg_total B EC SBP.

subst_boxarg_boxarg_total_n :
  subst_boxarg_boxarg_total B
                            ([u] boxarg_n ([x] (B' x u)))
                            (subst_boxarg_boxarg_n SBBP)
   <- ({x}
       {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b} subst_boxarg_total b ([u] x) (sbp cc tc b)) ->
      subst_boxarg_boxarg_total B (B' x) (SBBP x sbp)).

subst_boxarg_sequence_total : {B} {SC} subst_boxarg_sequence B SC SC' -> type.
%mode subst_boxarg_sequence_total +B +SC -SC'.

subst_boxarg_sequence_total_0 :
  subst_boxarg_sequence_total B ([u] sequence_c_0) subst_boxarg_sequence_0.

subst_boxarg_sequence_total_n :
  subst_boxarg_sequence_total B
                           ([u] (sequence_c_n (EC u) (SC u)))
                           (subst_boxarg_sequence_n SBSP SBP)
   <- subst_boxarg_sequence_total B SC SBSP
   <- subst_boxarg_total B EC SBP.

subst_boxarg_total_z : subst_boxarg_total B ([u] z_c) subst_boxarg_z.

subst_boxarg_total_s :
  subst_boxarg_total B ([u] (s_c (EC u))) (subst_boxarg_s SBP)
```

```
   <- subst_boxarg_total B EC SBP.

subst_boxarg_total_pair : subst_boxarg_total B
                                          ([u] (pair_c (EC1 u) (EC2 u)))
                                          (subst_boxarg_pair SBP2 SBP1)
                          <- subst_boxarg_total B EC1 SBP1
                          <- subst_boxarg_total B EC2 SBP2.

subst_boxarg_total_fst : subst_boxarg_total B
                                          ([u] (fst_c (EC u)))
                                          (subst_boxarg_fst SBP)
                          <- subst_boxarg_total B EC SBP.

subst_boxarg_total_snd : subst_boxarg_total B
                                          ([u] (snd_c (EC u)))
                                          (subst_boxarg_snd SBP)
                          <- subst_boxarg_total B EC SBP.

subst_boxarg_total_lam :
  subst_boxarg_total B ([u] (lam_c ([x] EC x u))) (subst_boxarg_lam SBP)
   <- ({x} {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b} subst_boxarg_total b ([u] x) (sbp cc tc b)) ->
       subst_boxarg_total B (EC x) (SBP x sbp)).

subst_boxarg_total_app : subst_boxarg_total B
                                          ([u] (app_c (EC1 u) (EC2 u)))
                                          (subst_boxarg_app SBP2 SBP1)
                         <- subst_boxarg_total B EC1 SBP1
                         <- subst_boxarg_total B EC2 SBP2.
subst_boxarg_total_case :
  subst_boxarg_total B
                     ([u] (case_c (EC u) (EC1 u) ([x] EC2 x u)))
                     (subst_boxarg_case SBP2 SBP1 SBP)
   <- subst_boxarg_total B EC SBP
   <- subst_boxarg_total B EC1 SBP1
   <- ({x} {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b} subst_boxarg_total b ([u] x) (sbp cc tc b)) ->
       subst_boxarg_total B (EC2 x) (SBP2 x sbp)).

subst_boxarg_total_fix :
  subst_boxarg_total B ([u] (fix_c ([x] EC x u))) (subst_boxarg_fix SBP)
   <- ({x} {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b} subst_boxarg_total b ([u] x) (sbp cc tc b)) ->
       subst_boxarg_total B (EC x) (SBP x sbp)).

subst_boxarg_total_box :
  subst_boxarg_total B ([u] (box_c W ([w] (B' w u)))) (subst_boxarg_box SBBP)
   <- ({w} subst_boxarg_boxarg_total B (B' w) (SBBP w)).

subst_boxarg_total_let_box :
  subst_boxarg_total B
                     ([u] (let_box_c (EC1 u) ([v] EC2 v u)))
                     (subst_boxarg_let_box SBP2 SBP1)
   <- subst_boxarg_total B EC1 SBP1
   <- ({v} subst_boxarg_total B (EC2 v) (SBP2 v)).

subst_boxarg_total_clo_u :
  subst_boxarg_total B ([u] (clo u (SC u))) (subst_boxarg_clo_u ECP SBSP)
   <- subst_boxarg_sequence_total B SC (SBSP : subst_boxarg_sequence B SC SC')
   <- eval_clo_total (B W) SC' ECP.

subst_boxarg_total_clo_v :
  subst_boxarg_total B ([u] (clo U (SC u))) (subst_boxarg_clo_v SBSP)
   <- subst_boxarg_sequence_total B SC SBSP.

%block sbt_b1 :
  some {W : world} {TC : tp_c}
  block {x : ec W TC}
        {sbp : {cc : ctx_c} {tc : tp_c} {b : {w : world} boxarg w cc tc}
               subst_boxarg b ([u : uvar cc tc] x) x}
        {sbtp : {cc : ctx_c} {tc : tp_c} {b : {w : world} boxarg w cc tc}
                subst_boxarg_total b ([u : uvar cc tc] x) (sbp cc tc b)}.
```

```
%block sbt_b2 : block {w : world}.
%block sbt_b3 : some {CC : ctx_c} {TC : tp_c} block {v : uvar CC TC}.

% -- Big-step evaluation rules for contextual expressions:
eval_c : ec W TC -> ec W TC -> type. %name eval_c ECP.
%mode eval_c +EC -VC.

eval_c_z : eval_c z_c z_c.

eval_c_s : eval_c (s_c EC) (s_c VC)
            <- eval_c EC VC.

eval_c_pair : eval_c (pair_c EC1 EC2) (pair_c VC1 VC2)
               <- eval_c EC1 VC1
               <- eval_c EC2 VC2.

eval_c_fst : eval_c (fst_c EC) VC1
              <- eval_c EC (pair_c VC1 VC2).

eval_c_snd : eval_c (snd_c EC) VC2
              <- eval_c EC (pair_c VC1 VC2).

eval_c_lam : eval_c (lam_c EC) (lam_c EC).

eval_c_app : eval_c (app_c EC1 EC2) VC1
              <- eval_c EC1 (lam_c EC1')
              <- eval_c EC2 VC2
              <- eval_c (EC1' VC2) VC1.

eval_c_case_z : eval_c (case_c EC EC1 EC2) VC1
                 <- eval_c EC z_c
                 <- eval_c EC1 VC1.

eval_c_case_s : eval_c (case_c EC EC1 EC2) VC2
                 <- eval_c EC (s_c VC)
                 <- eval_c (EC2 VC) VC2.

eval_c_fix : eval_c (fix_c EC) VC
              <- eval_c (EC (fix_c EC)) VC.

eval_c_box : eval_c (box_c W B) (box_c W B).

eval_c_let_box : eval_c (let_box_c EC1 EC2) VC
                  <- eval_c EC1 (box_c W B)
                  <- subst_boxarg B EC2 EC2'
                  <- eval_c EC2' VC.

%worlds () (eval_c _ _).
%covers eval_c +EC -VC.

eval_copy_c : eval_c EC VC -> eval_c EC VC -> type.
%mode eval_copy_c +ECP -ECP'.
eval_copy_c_r : eval_copy_c ECP ECP.
%worlds () (eval_copy_c _ _).
%total {} (eval_copy_c _ _).

% --- Value soundness for evaluation of contextual expressions:
val_sound_c : eval_c EC VC -> val_c VC -> type. %name val_sound_c VSCP.
%mode val_sound_c +ECP -VCP.

val_sound_c_z : val_sound_c eval_c_z val_c_z.

val_sound_c_s : val_sound_c (eval_c_s ECP) (val_c_s VCP)
                 <- val_sound_c ECP VCP.

val_sound_c_pair : val_sound_c (eval_c_pair ECP2 ECP1) (val_c_pair VCP2 VCP1)
                    <- val_sound_c ECP1 VCP1
                    <- val_sound_c ECP2 VCP2.

val_sound_c_fst : val_sound_c (eval_c_fst ECP) VCP1
                   <- val_sound_c ECP (val_c_pair VCP2 VCP1).
```

```
val_sound_c_snd : val_sound_c (eval_c_snd ECP) VCP2
                   <- val_sound_c ECP (val_c_pair VCP2 VCP1).

val_sound_c_lam : val_sound_c eval_c_lam val_c_lam.

val_sound_c_app : val_sound_c (eval_c_app ECP3 ECP2 ECP1) VCP
                   <- val_sound_c ECP3 VCP.

val_sound_c_case_z : val_sound_c (eval_c_case_z ECP1 ECP) VCP1
                      <- val_sound_c ECP1 VCP1.

val_sound_c_case_s : val_sound_c (eval_c_case_s ECP2 ECP) VCP2
                      <- val_sound_c ECP2 VCP2.

val_sound_c_fix : val_sound_c (eval_c_fix ECP) VCP
                   <- val_sound_c ECP VCP.

val_sound_c_box : val_sound_c eval_c_box val_c_box.

val_sound_c_let_box : val_sound_c (eval_c_let_box ECP2 SBP ECP1) VCP
                       <- val_sound_c ECP2 VCP.

%worlds () (val_sound_c _ _).
%total ECP (val_sound_c ECP _).

% -- Equality of contextual evaluation:
eq_eval_clo : eq_boxarg B B' ->
              eq_c_sequence SC SC' ->
              eval_clo B SC EC ->
              eval_clo B' SC' EC' ->
              eq_c EC EC' ->
              type.
%name eq_eval_clo QP.
%mode eq_eval_clo +QP +QP' +ECP +ECP' -QP''.
eq_eval_clo_0 : eq_eval_clo eq_boxarg_r
                            eq_c_sequence_r
                            eval_clo_0
                            eval_clo_0
                            eq_c_r.
eq_eval_clo_n : eq_eval_clo eq_boxarg_r
                            eq_c_sequence_r
                            (eval_clo_n ECP)
                            (eval_clo_n ECP')
                            QP
                <- eq_eval_clo eq_boxarg_r eq_c_sequence_r ECP ECP' QP.

eq_c_eval : eq_c EC EC' -> eval_c EC VC -> eval_c EC' VC -> type.
%name eq_c_eval QP.
%mode eq_c_eval +QP +ECP -ECP'.
eq_c_eval_r : eq_c_eval eq_c_r ECP ECP.

% -- Equality of boxarg substitution:
eq_subst_boxarg : ({w} eq_boxarg (B w) (B' w)) ->
                  subst_boxarg B EC EC' ->
                  subst_boxarg B' EC EC'' ->
                  eq_c EC' EC'' ->
                  type.
%name eq_subst_boxarg QP.
%mode eq_subst_boxarg +QP +SBP +SBP' -QP'.

eq_subst_boxarg_boxarg : ({w} eq_boxarg (B1 w) (B1' w)) ->
                         (subst_boxarg_boxarg B1 B2 B3) ->
                         (subst_boxarg_boxarg B1' B2 B3') ->
                         eq_boxarg B3 B3' ->
                         type.
%name eq_subst_boxarg_boxarg QP.
%mode eq_subst_boxarg_boxarg +QP +SBBP +SBBP' -QP'.

eq_subst_boxarg_boxarg_0 :
  eq_subst_boxarg_boxarg QP
                         (subst_boxarg_boxarg_0 SBP)
                         (subst_boxarg_boxarg_0 SBP')
```

```
                        QP''
   <- eq_subst_boxarg QP SBP SBP' QP'
   <- eq_boxarg_0 QP' QP''.

eq_subst_boxarg_boxarg_n :
  eq_subst_boxarg_boxarg QP
                        (subst_boxarg_boxarg_n SBBP)
                        (subst_boxarg_boxarg_n SBBP')
                        QP''
   <- ({x}
       {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b : {w} boxarg w cc tc} {qp : {w} eq_boxarg (b w) (b w)}
        eq_subst_boxarg qp (sbp cc tc b) (sbp cc tc b) eq_c_r) ->
       eq_subst_boxarg_boxarg QP
                              (SBBP x sbp)
                              (SBBP' x sbp)
                              (QP' x))
   <- eq_boxarg_n QP' QP''.

eq_subst_boxarg_sequence : ({w} eq_boxarg (B w) (B' w)) ->
                           subst_boxarg_sequence B SC SC' ->
                           subst_boxarg_sequence B' SC SC'' ->
                           eq_c_sequence SC' SC'' ->
                           type.
%name eq_subst_boxarg_sequence QP.
%mode eq_subst_boxarg_sequence +QP +SBSP +SBSP' -QP'.

eq_subst_boxarg_sequence_0 : eq_subst_boxarg_sequence QP
                                                      subst_boxarg_sequence_0
                                                      subst_boxarg_sequence_0
                                                      eq_c_sequence_r.

eq_subst_boxarg_sequence_n :
  eq_subst_boxarg_sequence QP
                           (subst_boxarg_sequence_n SBSP SBP)
                           (subst_boxarg_sequence_n SBSP' SBP')
                           QP'''
   <- eq_subst_boxarg QP SBP SBP' QP'
   <- eq_subst_boxarg_sequence QP SBSP SBSP' QP''
   <- eq_c_sequence_n QP' QP'' QP'''.

eq_subst_boxarg_z : eq_subst_boxarg QP subst_boxarg_z subst_boxarg_z eq_c_r.

eq_subst_boxarg_s : eq_subst_boxarg QP
                                    (subst_boxarg_s SBP)
                                    (subst_boxarg_s SBP')
                                    QP''
                     <- eq_subst_boxarg QP SBP SBP' QP'
                     <- eq_c_s QP' QP''.

eq_subst_boxarg_pair : eq_subst_boxarg QP
                                       (subst_boxarg_pair SBP2 SBP1)
                                       (subst_boxarg_pair SBP2' SBP1')
                                       QP'
                        <- eq_subst_boxarg QP SBP1 SBP1' QP1
                        <- eq_subst_boxarg QP SBP2 SBP2' QP2
                        <- eq_c_pair QP1 QP2 QP'.

eq_subst_boxarg_fst : eq_subst_boxarg QP
                                      (subst_boxarg_fst SBP)
                                      (subst_boxarg_fst SBP')
                                      QP''
                        <- eq_subst_boxarg QP SBP SBP' QP'
                        <- eq_c_fst QP' QP''.

eq_subst_boxarg_snd : eq_subst_boxarg QP
                                      (subst_boxarg_snd SBP)
                                      (subst_boxarg_snd SBP')
                                      QP''
                        <- eq_subst_boxarg QP SBP SBP' QP'
                        <- eq_c_snd QP' QP''.
```

```
eq_subst_boxarg_lam :
  eq_subst_boxarg QP
                  (subst_boxarg_lam SBP)
                  (subst_boxarg_lam SBP')
                  QP''
   <- ({x}
       {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b : {w} boxarg w cc tc} {qp : {w} eq_boxarg (b w) (b w)}
        eq_subst_boxarg qp (sbp cc tc b) (sbp cc tc b) eq_c_r) ->
       eq_subst_boxarg QP (SBP x sbp) (SBP' x sbp) (QP' x))
   <- eq_c_lam QP' QP''.

eq_subst_boxarg_app : eq_subst_boxarg QP
                                      (subst_boxarg_app SBP2 SBP1)
                                      (subst_boxarg_app SBP2' SBP1')
                                      QP'
                      <- eq_subst_boxarg QP SBP1 SBP1' QP1
                      <- eq_subst_boxarg QP SBP2 SBP2' QP2
                      <- eq_c_app QP1 QP2 QP'.

eq_subst_boxarg_case :
  eq_subst_boxarg QP
                  (subst_boxarg_case SBP2 SBP1 SBP)
                  (subst_boxarg_case SBP2' SBP1' SBP')
                  QP''
   <- eq_subst_boxarg QP SBP SBP' QP'
   <- eq_subst_boxarg QP SBP1 SBP1' QP1
   <- ({x}
       {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b : {w} boxarg w cc tc} {qp : {w} eq_boxarg (b w) (b w)}
        eq_subst_boxarg qp (sbp cc tc b) (sbp cc tc b) eq_c_r) ->
       eq_subst_boxarg QP (SBP2 x sbp) (SBP2' x sbp) (QP2 x))
   <- eq_c_case QP' QP1 QP2 QP''.


eq_subst_boxarg_fix :
  eq_subst_boxarg QP (subst_boxarg_fix SBP) (subst_boxarg_fix SBP') QP''
   <- ({x}
       {sbp : {cc} {tc} {b : {w} boxarg w cc tc} subst_boxarg b ([u] x) x}
       ({cc} {tc} {b : {w} boxarg w cc tc} {qp : {w} eq_boxarg (b w) (b w)}
        eq_subst_boxarg qp (sbp cc tc b) (sbp cc tc b) eq_c_r) ->
       eq_subst_boxarg QP (SBP x sbp) (SBP' x sbp) (QP' x))
   <- eq_c_fix QP' QP''.

eq_subst_boxarg_box :
  eq_subst_boxarg QP
                  (subst_boxarg_box SBBP)
                  (subst_boxarg_box SBBP')
                  QP''
   <- ({w} eq_subst_boxarg_boxarg QP (SBBP w) (SBBP' w) (QP' w))
   <- eq_c_box W QP' QP''.

eq_subst_boxarg_let_box :
  eq_subst_boxarg QP
                  (subst_boxarg_let_box SBP2 SBP1)
                  (subst_boxarg_let_box SBP2' SBP1')
                  QP'
   <- eq_subst_boxarg QP SBP1 SBP1' QP1
   <- ({u} eq_subst_boxarg QP (SBP2 u) (SBP2' u) (QP2 u))
   <- eq_c_let_box QP1 QP2 QP'.

eq_subst_boxarg_clo_u : eq_subst_boxarg QP
                                        (subst_boxarg_clo_u ECP SBSP)
                                        (subst_boxarg_clo_u ECP' SBSP')
                                        QP''
                        <- eq_subst_boxarg_sequence QP SBSP SBSP' QP'
                        <- eq_eval_clo (QP W) QP' ECP ECP' QP''.

eq_subst_boxarg_clo_v : eq_subst_boxarg QP
                                        (subst_boxarg_clo_v SBSP)
                                        (subst_boxarg_clo_v SBSP')
                                        QP''
```

```
                        <- eq_subst_boxarg_sequence QP SBSP SBSP' QP'
                        <- eq_clo QP' QP''.

%block esb_b1 : block {w : world}.
%block esb_b2 :
  some {W : world} {TC : tp_c}
  block {x : ec W TC}
        {sbp : {cc : ctx_c} {tc : tp_c} {b : {w : world} boxarg w cc tc}
               subst_boxarg b ([u : uvar cc tc] x) x}
        {qp : {cc} {tc}
              {b : {w} boxarg w cc tc}
              {qp' : {w} eq_boxarg (b w) (b w)}
              eq_subst_boxarg qp' (sbp cc tc b) (sbp cc tc b) eq_c_r}.
%block esb_b3 : some {CC : ctx_c} {TC : tp_c} block {u : uvar CC TC}.

%worlds (esb_b1 | esb_b2 | esb_b3)
        (eq_c_s _ _)
        (eq_c_pair _ _ _)
        (eq_c_pair_b2 _ _)
        (eq_c_pair_b1 _ _)
        (eq_c_fst _ _)
        (eq_c_snd _ _)
        (eq_c_lam _ _)
        (eq_c_app _ _ _)
        (eq_c_case _ _ _ _)
        (eq_c_fix _ _)
        (eq_c_let_box _ _ _)
        (eq_c_box _ _ _)
        (eq_boxarg_0 _ _)
        (eq_boxarg_n _ _)
        (eq_c_sequence_n _ _ _)
        (eq_clo _ _)
        (eq_eval_clo _ _ _ _ _)
        (eq_subst_boxarg_sequence _ _ _ _)
        (eq_subst_boxarg_boxarg _ _ _ _)
        (eq_subst_boxarg _ _ _ _).

%total {} (eq_c_s _ _).
%total {} (eq_c_pair _ _ _).
%total {} (eq_c_pair_b1 _ _).
%total {} (eq_c_pair_b2 _ _).
%total {} (eq_c_fst _ _).
%total {} (eq_c_snd _ _).
%total {} (eq_c_lam _ _).
%total {} (eq_c_app _ _ _).
%total {} (eq_c_case _ _ _ _).
%total {} (eq_c_fix _ _).
%total {} (eq_c_let_box _ _ _).
%total {} (eq_c_box _ _ _).
%total {} (eq_boxarg_0 _ _).
%total {} (eq_boxarg_n _ _).
%total {} (eq_c_sequence_n _ _ _).
%total {} (eq_clo _ _).
%total ECP (eq_eval_clo _ _ ECP _ _).
%total (SBSP SBBP SBP)
       (eq_subst_boxarg_sequence _ SBSP _ _)
       (eq_subst_boxarg_boxarg _ SBBP _ _)
       (eq_subst_boxarg _ SBP _ _).

% --- Determinacy of evaluation of contextual expressions
eval_det_c : eval_c EC VC -> eval_c EC VC' -> eq_c VC VC' -> type.
%mode eval_det_c +ECP +ECP' -QP.

eval_det_c_z : eval_det_c eval_c_z eval_c_z eq_c_r.

eval_det_c_s : eval_det_c (eval_c_s ECP) (eval_c_s ECP') QP'
                <- eval_det_c ECP ECP' QP
                <- eq_c_s QP QP'.

eval_det_c_pair : eval_det_c (eval_c_pair ECP2 ECP1)
                             (eval_c_pair ECP2' ECP1')
                             QP
```

```
                  <- eval_det_c ECP1 ECP1' QP1
                  <- eval_det_c ECP2 ECP2' QP2
                  <- eq_c_pair QP1 QP2 QP.

eval_det_c_fst : eval_det_c (eval_c_fst ECP)
                            (eval_c_fst ECP')
                            QP'
                  <- eval_det_c ECP ECP' QP
                  <- eq_c_pair_b1 QP QP'.

eval_det_c_snd : eval_det_c (eval_c_snd ECP)
                            (eval_c_snd ECP')
                            QP'
                  <- eval_det_c ECP ECP' QP
                  <- eq_c_pair_b2 QP QP'.

eval_det_c_lam : eval_det_c eval_c_lam eval_c_lam eq_c_r.

eval_det_c_app : eval_det_c (eval_c_app ECP3 ECP2 ECP1)
                            (eval_c_app ECP3' ECP2' ECP1')
                            QP''
                  <- eval_det_c ECP1 ECP1' QP1
                  <- eval_det_c ECP2 ECP2' QP2
                  <- eq_c_lam_b QP1 QP2 QP
                  <- eq_c_sym QP QP'
                  <- eq_c_eval QP' ECP3' ECP3''
                  <- eval_det_c ECP3 ECP3'' QP''.

eval_det_c_case_zz : eval_det_c (eval_c_case_z ECP1 ECP)
                                (eval_c_case_z ECP1' ECP')
                                QP1
                      <- eval_det_c ECP1 ECP1' QP1.

eval_det_c_case_zs : eval_det_c (eval_c_case_z ECP1 ECP)
                                (eval_c_case_s ECP1' ECP')
                                QP'
                      <- eval_det_c ECP ECP' QP
                      <- neq_c_zs QP QP'.

eval_det_c_case_sz : eval_det_c (eval_c_case_s ECP1 ECP)
                                (eval_c_case_z ECP1' ECP')
                                QP''
                      <- eval_det_c ECP ECP' QP
                      <- eq_c_sym QP QP'
                      <- neq_c_zs QP' QP''.

eval_det_c_case_ss :
  eval_det_c ((eval_c_case_s ECP1 ECP) : eval_c (case_c _ _ EC2) _)
             (eval_c_case_s ECP1' ECP')
             QP''''
   <- eval_det_c ECP ECP' QP
   <- eq_c_s_b QP QP'
   <- eq_c_sym QP' QP''
   <- eq_c_hole QP'' EC2 QP'''
   <- eq_c_eval QP''' ECP1' ECP1''
   <- eval_det_c ECP1 ECP1'' QP''''.

eval_det_c_fix : eval_det_c (eval_c_fix ECP)
                            (eval_c_fix ECP')
                            QP
                  <- eval_det_c ECP ECP' QP.

eval_det_c_box : eval_det_c eval_c_box eval_c_box eq_c_r.

eval_det_c_let_box :
  eval_det_c ((eval_c_let_box ECP2 SBP ECP1) : eval_c (let_box_c _ EC2) _)
             (eval_c_let_box ECP2' SBP' ECP1')
             QP'''
   <- eval_det_c ECP1 ECP1' QP1
   <- eq_box_boxarg QP1 QP
   <- ({w} eq_sym_boxarg (QP w) (QP' w))
   <- eq_subst_boxarg QP' SBP' SBP QP''
```

```
   <- eq_c_eval QP'' ECP2' ECP2''
   <- eval_det_c ECP2 ECP2'' QP'''.

%block eu_b1 : block {w : world}.

%worlds (eu_b1)
        (eq_c_eval _ _ _)
        (eq_c_sym _ _)
        (eq_c_hole _ _ _)
        (eq_c_s_b _ _)
        (neq_c_zs _ _)
        (eq_c_lam_b _ _ _)
        (eq_sym_boxarg _ _)
        (eq_box_boxarg _ _)
        (eval_det_c _ _ _).

%total {} (eq_c_eval _ _ _).
%total {} (eq_c_sym _ _).
%total {} (eq_c_hole _ _ _).
%total {} (eq_c_s_b _ _).
%total {} (neq_c_zs _ _).
%total {} (eq_c_lam_b _ _ _).
%total {} (eq_sym_boxarg _ _).
%total {} (eq_box_boxarg _ _).
%total ECP (eval_det_c ECP _ _).

% -- Small-step evaluation rules for contextual expressions:
step_c : ec W TC -> ec W TC -> type. %name step_c SCP.
%mode step_c +EC -EC'.

step_c_s : step_c (s_c EC) (s_c EC')
           <- step_c EC EC'.

step_c_pair_1 : step_c (pair_c VC1 EC2) (pair_c VC1 EC2')
                 <- val_c VC1
                 <- step_c EC2 EC2'.

step_c_pair_2 : step_c (pair_c EC1 EC2) (pair_c EC1' EC2)
                 <- step_c EC1 EC1'.

step_c_fst_1 : step_c (fst_c (pair_c VC1 VC2)) VC1
                <- val_c VC1
                <- val_c VC2.

step_c_fst_2 : step_c (fst_c EC) (fst_c EC')
                <- step_c EC EC'.

step_c_snd_1 : step_c (snd_c (pair_c VC1 VC2)) VC2
                <- val_c VC1
                <- val_c VC2.

step_c_snd_2 : step_c (snd_c EC) (snd_c EC')
                <- step_c EC EC'.

step_c_app_1 : step_c (app_c (lam_c EC1) VC2) (EC1 VC2)
                <- val_c VC2.

step_c_app_2 : step_c (app_c VC1 EC2) (app_c VC1 EC2')
                <- val_c VC1
                <- step_c EC2 EC2'.

step_c_app_3 : step_c (app_c EC1 EC2) (app_c EC1' EC2)
                <- step_c EC1 EC1'.

step_c_case_1 : step_c (case_c z_c EC1 EC2) EC1.

step_c_case_2 : step_c (case_c (s_c VC) EC1 EC2) (EC2 VC)
                 <- val_c VC.

step_c_case_3 : step_c (case_c EC EC1 EC2) (case_c EC' EC1 EC2)
                 <- step_c EC EC'.
```

```
step_c_fix : step_c (fix_c EC) (EC (fix_c EC)).

step_c_let_box_1 : step_c (let_box_c (box_c W B) EC) EC'
                    <- subst_boxarg B EC EC'.

step_c_let_box_2 : step_c (let_box_c EC1 EC2) (let_box_c EC1' EC2)
                    <- step_c EC1 EC1'.

% -- Multiple computation steps for contextual expressions:
steps_c : ec W TC -> ec W TC -> type. %name steps_c SSCP.
%mode steps_c +EC -EC'.
steps_c_0 : steps_c EC EC.
steps_c_n : steps_c EC EC''
             <- step_c EC EC'
             <- steps_c EC' EC''.

% -- A contextual value evaluates to itself:
eval_val_c : val_c VC -> eval_c VC VC -> type.
%mode eval_val_c +VCP -ECP.
eval_val_c_z : eval_val_c val_c_z eval_c_z.
eval_val_c_s : eval_val_c (val_c_s VCP) (eval_c_s ECP)
                <- eval_val_c VCP ECP.
eval_val_c_pair : eval_val_c (val_c_pair VCP2 VCP1) (eval_c_pair ECP2 ECP1)
                   <- eval_val_c VCP1 ECP1
                   <- eval_val_c VCP2 ECP2.
eval_val_c_lam : eval_val_c val_c_lam eval_c_lam.
eval_val_c_box : eval_val_c val_c_box eval_c_box.
%worlds () (eval_val_c _ _).
%total VCP (eval_val_c VCP _).

% -- Concatenation of step and evaluation:
% E -> E' and E' ~> V => E ~> V
cat_step_eval_c : step_c EC EC' -> eval_c EC' VC -> eval_c EC VC -> type.
%mode cat_step_eval_c +SCP +ECP -ECP'.

inject_step_eval_c_case : step_c E E' ->
                          eval_c (case_c E' E1 E2) V ->
                          eval_c (case_c E E1 E2) V ->
                          type.
%mode inject_step_eval_c_case +SCP +ECP -ECP'.
inject_step_eval_c_case_z : inject_step_eval_c_case SCP
                                                    (eval_c_case_z ECP2 ECP1)
                                                    (eval_c_case_z ECP2 ECP1')
                             <- cat_step_eval_c SCP ECP1 ECP1'.
inject_step_eval_c_case_s : inject_step_eval_c_case SCP
                                                    (eval_c_case_s ECP2 ECP1)
                                                    (eval_c_case_s ECP2 ECP1')
                             <- cat_step_eval_c SCP ECP1 ECP1'.

cat_step_eval_c_s : cat_step_eval_c (step_c_s SCP)
                                    ECP
                                    (eval_c_s ECP'')
                     <- eval_copy_c ECP (eval_c_s ECP')
                     <- cat_step_eval_c SCP ECP' ECP''.

cat_step_eval_c_pair_1 : cat_step_eval_c (step_c_pair_1 SCP VCP)
                                         ECP
                                         (eval_c_pair ECP2' ECP1)
                          <- eval_copy_c ECP (eval_c_pair ECP2 ECP1)
                          <- cat_step_eval_c SCP ECP2 ECP2'.

cat_step_eval_c_pair_2 : cat_step_eval_c (step_c_pair_2 SCP)
                                         ECP
                                         (eval_c_pair ECP2 ECP1')
                          <- eval_copy_c ECP (eval_c_pair ECP2 ECP1)
                          <- cat_step_eval_c SCP ECP1 ECP1'.

cat_step_eval_c_fst_1 : cat_step_eval_c (step_c_fst_1 VCP2 VCP1)
                                        ECP
                                        (eval_c_fst (eval_c_pair ECP2 ECP))
                         <- eval_val_c VCP1 ECP1
                         <- eval_det_c ECP ECP1 QP
```

```
                        <- eval_val_c VCP2 ECP2.

cat_step_eval_c_fst_2 : cat_step_eval_c (step_c_fst_2 SCP)
                                        ECP
                                        (eval_c_fst ECP'')
                        <- eval_copy_c ECP (eval_c_fst ECP')
                        <- cat_step_eval_c SCP ECP' ECP''.

cat_step_eval_c_snd_1 : cat_step_eval_c (step_c_snd_1 VCP2 VCP1)
                                        ECP
                                        (eval_c_snd (eval_c_pair ECP ECP1))
                        <- eval_val_c VCP1 ECP1
                        <- eval_val_c VCP2 ECP2
                        <- eval_det_c ECP ECP2 QP.

cat_step_eval_c_snd_2 : cat_step_eval_c (step_c_snd_2 SCP)
                                        ECP
                                        (eval_c_snd ECP'')
                        <- eval_copy_c ECP (eval_c_snd ECP')
                        <- cat_step_eval_c SCP ECP' ECP''.

cat_step_eval_c_app_1 : cat_step_eval_c (step_c_app_1 VCP)
                                        ECP
                                        (eval_c_app ECP ECP' eval_c_lam)
                        <- eval_val_c VCP ECP'.

cat_step_eval_c_app_2 : cat_step_eval_c (step_c_app_2 SCP VCP)
                                        ECP
                                        (eval_c_app ECP3 ECP2' ECP1)
                        <- eval_copy_c ECP (eval_c_app ECP3 ECP2 ECP1)
                        <- cat_step_eval_c SCP ECP2 ECP2'.

cat_step_eval_c_app_3 : cat_step_eval_c (step_c_app_3 SCP)
                                        ECP
                                        (eval_c_app ECP3 ECP2 ECP1')
                        <- eval_copy_c ECP (eval_c_app ECP3 ECP2 ECP1)
                        <- cat_step_eval_c SCP ECP1 ECP1'.

cat_step_eval_c_case_1 : cat_step_eval_c step_c_case_1
                                         ECP
                                         (eval_c_case_z ECP eval_c_z).

cat_step_eval_c_case_2 : cat_step_eval_c (step_c_case_2 VCP)
                                         ECP
                                         (eval_c_case_s ECP (eval_c_s ECP'))
                         <- eval_val_c VCP ECP'.

cat_step_eval_c_case_3 : cat_step_eval_c (step_c_case_3 SCP)
                                         ECP
                                         ECP'
                         <- inject_step_eval_c_case SCP ECP ECP'.

cat_step_eval_c_fix : cat_step_eval_c step_c_fix ECP (eval_c_fix ECP).

cat_step_eval_c_let_box_1 : cat_step_eval_c (step_c_let_box_1 SBP)
                                            ECP
                                            (eval_c_let_box ECP SBP eval_c_box).

cat_step_eval_c_let_box_2 : cat_step_eval_c (step_c_let_box_2 SCP)
                                            ECP
                                            (eval_c_let_box ECP2 SBP ECP1')
                            <- eval_copy_c ECP (eval_c_let_box ECP2 SBP ECP1)
                            <- cat_step_eval_c SCP ECP1 ECP1'.

%worlds () (cat_step_eval_c _ _ _)
           (inject_step_eval_c_case _ _ _).

%total (SCP SCP') (cat_step_eval_c SCP _ _)
                  (inject_step_eval_c_case SCP' _ _).

% -- Concatenation of steps and evaluation:
% E ->* E' and E' ~> V => E ~> V
```

```
cat_steps_eval_c : steps_c EC EC' -> eval_c EC' VC -> eval_c EC VC -> type.
%mode cat_steps_eval_c +SSCP +ECP -ECP'.

cat_steps_eval_c_0 : cat_steps_eval_c steps_c_0 ECP ECP.

cat_steps_eval_c_n : cat_steps_eval_c (steps_c_n SSCP SCP) ECP ECP''
                      <- cat_steps_eval_c SSCP ECP ECP'
                      <- cat_step_eval_c SCP ECP' ECP''.

%worlds () (cat_steps_eval_c _ _ _).
%total SSCP (cat_steps_eval_c SSCP _ _).

% -- Concatenation of steps:
% E ->* E' and E' ->* E'' => E ->* E''
cat_steps_c : steps_c EC EC' -> steps_c EC' EC'' -> steps_c EC EC'' -> type.
%mode cat_steps_c +SSCP +SSCP' -SSCP''.

cat_steps_c_0 : cat_steps_c steps_c_0 SSCP SSCP.

cat_steps_c_n : cat_steps_c (steps_c_n SSCP SCP) SSCP' (steps_c_n SSCP'' SCP)
                <- cat_steps_c SSCP SSCP' SSCP''.

%worlds () (cat_steps_c _ _ _).
%total SSCP (cat_steps_c SSCP _ _).

% -- Injection of steps into steps:
inject_steps_c_s : steps_c EC EC' -> steps_c (s_c EC) (s_c EC') -> type.
%mode inject_steps_c_s +SSCP -SSCP'.

inject_steps_c_s_0 : inject_steps_c_s steps_c_0 steps_c_0.

inject_steps_c_s_n : inject_steps_c_s (steps_c_n SSCP SCP)
                                      (steps_c_n SSCP' (step_c_s SCP))
                      <- inject_steps_c_s SSCP SSCP'.

%worlds () (inject_steps_c_s _ _).
%total SSCP (inject_steps_c_s SSCP _).

inject_steps_c_pair_1 : steps_c EC2 EC2' ->
                        val_c VC1 ->
                        steps_c (pair_c VC1 EC2) (pair_c VC1 EC2') ->
                        type.
%mode inject_steps_c_pair_1 +SSCP +VCP -SSCP'.

inject_steps_c_pair_1_0 : inject_steps_c_pair_1 steps_c_0 VCP steps_c_0.

inject_steps_c_pair_1_n :
  inject_steps_c_pair_1 (steps_c_n SSCP SCP)
                        VCP
                        (steps_c_n SSCP' (step_c_pair_1 SCP VCP))
   <- inject_steps_c_pair_1 SSCP VCP SSCP'.

%worlds () (inject_steps_c_pair_1 _ _ _).
%total SSCP (inject_steps_c_pair_1 SSCP _ _).

inject_steps_c_pair_2 : steps_c EC1 EC1' ->
                        steps_c (pair_c EC1 EC2) (pair_c EC1' EC2) ->
                        type.
%mode +{W : world} +{TC1 : tp_c} +{TC2 : tp_c}
      +{EC1 : ec W TC1} +{EC1' : ec W TC1} +{EC2 : ec W TC2}
      +{SSCP : steps_c EC1 EC1'}
      -{SSCP':steps_c (pair_c EC1 EC2) (pair_c EC1' EC2)}
      (inject_steps_c_pair_2 SSCP SSCP').

inject_steps_c_pair_2_0 : inject_steps_c_pair_2 steps_c_0 steps_c_0.

inject_steps_c_pair_2_n :
  inject_steps_c_pair_2 (steps_c_n SSCP SCP)
                        (steps_c_n SSCP' (step_c_pair_2 SCP))
   <- inject_steps_c_pair_2 SSCP SSCP'.

%worlds () (inject_steps_c_pair_2 _ _).
```

```
%total SSCP (inject_steps_c_pair_2 SSCP _).

inject_steps_c_fst : val_c VC1 -> val_c VC2 ->
                     steps_c EC (pair_c VC1 VC2) ->
                     steps_c (fst_c EC) VC1 ->
                     type.
%mode inject_steps_c_fst +VCP1 +VCP2 +SSCP -SSCP'.

inject_steps_c_fst_0 :
  inject_steps_c_fst VCP1 VCP2
                     steps_c_0
                     (steps_c_n steps_c_0 (step_c_fst_1 VCP2 VCP1)).

inject_steps_c_fst_n : inject_steps_c_fst VCP1 VCP2
                                          (steps_c_n SSCP SCP)
                                          (steps_c_n SSCP' (step_c_fst_2 SCP))
                       <- inject_steps_c_fst VCP1 VCP2 SSCP SSCP'.

%worlds () (inject_steps_c_fst _ _ _ _).
%total SSCP (inject_steps_c_fst _ _ SSCP _).

inject_steps_c_snd : val_c VC1 -> val_c VC2 ->
                     steps_c EC (pair_c VC1 VC2) ->
                     steps_c (snd_c EC) VC2 ->
                     type.
%mode inject_steps_c_snd +VCP1 +VCP2 +SSCP -SSCP'.

inject_steps_c_snd_0 :
  inject_steps_c_snd VCP1 VCP2
                     steps_c_0
                     (steps_c_n steps_c_0 (step_c_snd_1 VCP2 VCP1)).

inject_steps_c_snd_n : inject_steps_c_snd VCP1 VCP2
                                          (steps_c_n SSCP SCP)
                                          (steps_c_n SSCP' (step_c_snd_2 SCP))
                       <- inject_steps_c_snd VCP1 VCP2 SSCP SSCP'.

%worlds () (inject_steps_c_snd _ _ _ _).
%total SSCP (inject_steps_c_snd _ _ SSCP _).

inject_steps_c_app_1 : steps_c EC1 EC1' ->
                       steps_c (app_c EC1 EC2) (app_c EC1' EC2) ->
                       type.
%mode +{W : world} +{TC1 : tp_c} +{TC2 : tp_c}
      +{EC1 : ec W (arrow_c TC1 TC2)}
      +{EC1' : ec W (arrow_c TC1 TC2)}
      +{EC2 : ec W TC1}
      +{SSCP : steps_c EC1 EC1'}
      -{SSCP' : steps_c (app_c EC1 EC2) (app_c EC1' EC2)}
      (inject_steps_c_app_1 SSCP SSCP').

inject_steps_c_app_1_0 : inject_steps_c_app_1 steps_c_0 steps_c_0.

inject_steps_c_app_1_n :
  inject_steps_c_app_1 (steps_c_n SSCP SCP)
                       (steps_c_n SSCP' (step_c_app_3 SCP))
   <- inject_steps_c_app_1 SSCP SSCP'.

%worlds () (inject_steps_c_app_1 _ _).
%total SSCP (inject_steps_c_app_1 SSCP _).

inject_steps_c_app_2 :
  steps_c EC2 EC2' ->
  steps_c (app_c (lam_c EC1) EC2) (app_c (lam_c EC1) EC2') ->
  type.
%mode +{W : world} +{TC1 : tp_c} +{TC2 : tp_c}
      +{EC1 : ec W TC1 -> ec W TC2}
      +{EC2 : ec W TC1} +{EC2' : ec W TC1}
      +{SSCP : steps_c EC2 EC2'}
      -{SSCP' : steps_c (app_c (lam_c EC1) EC2) (app_c (lam_c EC1) EC2')}
      (inject_steps_c_app_2 SSCP SSCP').
```

```
inject_steps_c_app_2_0 : inject_steps_c_app_2 steps_c_0 steps_c_0.

inject_steps_c_app_2_n :
  inject_steps_c_app_2 (steps_c_n SSCP SCP)
                       (steps_c_n SSCP' (step_c_app_2 SCP val_c_lam))
   <- inject_steps_c_app_2 SSCP SSCP'.

%worlds () (inject_steps_c_app_2 _ _).
%total SSCP (inject_steps_c_app_2 SSCP _).

inject_steps_c_case : steps_c EC EC' ->
                      steps_c (case_c EC EC1 EC2) (case_c EC' EC1 EC2) ->
                      type.
%mode +{W : world} +{TC1 : tp_c}
      +{EC : ec W int_c} +{EC' : ec W int_c}
      +{EC1 : ec W TC1} +{EC2 : ec W int_c -> ec W TC1}
      +{SSCP : steps_c EC EC'}
      -{SSCP' : steps_c (case_c EC EC1 EC2) (case_c EC' EC1 EC2)}
      (inject_steps_c_case SSCP SSCP').

inject_steps_c_case_0 : inject_steps_c_case steps_c_0 steps_c_0.

inject_steps_c_case_n :
  inject_steps_c_case (steps_c_n SSCP SCP)
                      (steps_c_n SSCP' (step_c_case_3 SCP))
   <- inject_steps_c_case SSCP SSCP'.

%worlds () (inject_steps_c_case _ _).
%total SSCP (inject_steps_c_case SSCP _).

inject_steps_c_let_box : steps_c EC1 EC1' ->
                         steps_c (let_box_c EC1 EC2) (let_box_c EC1' EC2) ->
                         type.
%mode +{W : world} +{CC : ctx_c} +{TC1 : tp_c} +{TC2 : tp_c}
      +{EC1 : ec W (code_c CC TC1)} +{EC1' : ec W (code_c CC TC1)}
      +{EC2 : uvar CC TC1 -> ec W TC2}
      +{SSCP : steps_c EC1 EC1'}
      -{SSCP' : steps_c (let_box_c EC1 EC2) (let_box_c EC1' EC2)}
      (inject_steps_c_let_box SSCP SSCP').

inject_steps_c_let_box_0 : inject_steps_c_let_box steps_c_0 steps_c_0.

inject_steps_c_let_box_n :
  inject_steps_c_let_box (steps_c_n SSCP SCP)
                         (steps_c_n SSCP' (step_c_let_box_2 SCP))
   <- inject_steps_c_let_box SSCP SSCP'.

%worlds () (inject_steps_c_let_box _ _).
%total SSCP (inject_steps_c_let_box SSCP _).

% -- Equivalens between big and small step evaluation for contextual
% -- expressions:
% E ->* V and V value => E ~> V
small_to_big_c : steps_c EC VC -> val_c VC -> eval_c EC VC -> type.
%mode small_to_big_c +SSCP +VCP -ECP.
small_to_big_c_proof : small_to_big_c SSCP VCP ECP'
                       <- eval_val_c VCP ECP
                       <- cat_steps_eval_c SSCP ECP ECP'.
%worlds () (small_to_big_c _ _ _).
%total {} (small_to_big_c _ _ _).

% E ~> V => E ->* V
big_to_small_c : eval_c EC VC -> steps_c EC VC -> type.
%mode big_to_small_c +ECP -SCP.

big_to_small_c_z : big_to_small_c eval_c_z steps_c_0.

big_to_small_c_s : big_to_small_c (eval_c_s ECP) SSCP'
                   <- big_to_small_c ECP SSCP
                   <- inject_steps_c_s SSCP SSCP'.

big_to_small_c_pair : big_to_small_c (eval_c_pair ECP2 ECP1) SSCP''
```

```
                      <- big_to_small_c ECP1 SSCP1
                      <- inject_steps_c_pair_2 SSCP1 SSCP
                      <- val_sound_c ECP1 VCP1
                      <- big_to_small_c ECP2 SSCP2
                      <- inject_steps_c_pair_1 SSCP2 VCP1 SSCP'
                      <- cat_steps_c SSCP SSCP' SSCP''.

big_to_small_c_fst : big_to_small_c (eval_c_fst ECP) SSCP'
                      <- big_to_small_c ECP SSCP
                      <- val_sound_c ECP (val_c_pair VCP2 VCP1)
                      <- inject_steps_c_fst VCP1 VCP2 SSCP SSCP'.

big_to_small_c_snd : big_to_small_c (eval_c_snd ECP) SSCP'
                      <- big_to_small_c ECP SSCP
                      <- val_sound_c ECP (val_c_pair VCP2 VCP1)
                      <- inject_steps_c_snd VCP1 VCP2 SSCP SSCP'.

big_to_small_c_lam : big_to_small_c eval_c_lam steps_c_0.

big_to_small_c_app :
  big_to_small_c (eval_c_app ECP3 ECP2 ECP1) SSCP'
   <- big_to_small_c ECP1 SSCP1
   <- inject_steps_c_app_1 SSCP1 SSCP1'
   <- big_to_small_c ECP2 SSCP2
   <- inject_steps_c_app_2 SSCP2 SSCP2'
   <- cat_steps_c SSCP1' SSCP2' SSCP
   <- big_to_small_c ECP3 SSCP3
   <- val_sound_c ECP2 VCP
   <- cat_steps_c SSCP (steps_c_n SSCP3 (step_c_app_1 VCP)) SSCP'.

big_to_small_c_case_z :
  big_to_small_c (eval_c_case_z ECP1 ECP) SSCP''
   <- big_to_small_c ECP SSCP
   <- inject_steps_c_case SSCP SSCP'
   <- big_to_small_c ECP1 SSCP1
   <- cat_steps_c SSCP' (steps_c_n SSCP1 step_c_case_1) SSCP''.

big_to_small_c_case_s :
  big_to_small_c (eval_c_case_s ECP2 ECP) SSCP''
   <- big_to_small_c ECP SSCP
   <- inject_steps_c_case SSCP SSCP'
   <- big_to_small_c ECP2 SSCP2
   <- val_sound_c ECP (val_c_s VCP)
   <- cat_steps_c SSCP' (steps_c_n SSCP2 (step_c_case_2 VCP)) SSCP''.

big_to_small_c_fix :
  big_to_small_c (eval_c_fix ECP) (steps_c_n SSCP step_c_fix)
   <- big_to_small_c ECP SSCP.

big_to_small_c_box : big_to_small_c eval_c_box steps_c_0.

big_to_small_c_let_box :
  big_to_small_c (eval_c_let_box ECP2 SBP ECP1) SSCP
   <- big_to_small_c ECP1 SSCP1
   <- inject_steps_c_let_box SSCP1 SSCP1'
   <- big_to_small_c ECP2 SSCP2
   <- cat_steps_c SSCP1' (steps_c_n SSCP2 (step_c_let_box_1 SBP)) SSCP.

% -- Not-stuckness for contextual expressions to be used in proof of progress:
not_stuck_c : ec W TC -> type. %name not_stuck_c NSCP.
%mode not_stuck_c +EC.
not_stuck_c_val : not_stuck_c VC
                   <- val_c VC.
not_stuck_c_step : not_stuck_c EC
                    <- step_c EC EC'.

not_stuck_c_s : not_stuck_c EC -> not_stuck_c (s_c EC) -> type.
%mode not_stuck_c_s +NSCP -NSCP'.
not_stuck_c_s_val : not_stuck_c_s (not_stuck_c_val VCP)
                                  (not_stuck_c_val (val_c_s VCP)).
not_stuck_c_s_step : not_stuck_c_s (not_stuck_c_step SCP)
                                   (not_stuck_c_step (step_c_s SCP)).
```

```
not_stuck_c_pair : not_stuck_c EC1 ->
                   not_stuck_c EC2 ->
                   not_stuck_c (pair_c EC1 EC2) ->
                   type.
%mode not_stuck_c_pair +NSCP1 +NSCP2 -NSCP.

not_stuck_c_pair_vv : not_stuck_c_pair (not_stuck_c_val VCP1)
                                       (not_stuck_c_val VCP2)
                                       (not_stuck_c_val (val_c_pair VCP2 VCP1)).
not_stuck_c_pair_vs :
  not_stuck_c_pair (not_stuck_c_val VCP)
                   (not_stuck_c_step SCP)
                   (not_stuck_c_step (step_c_pair_1 SCP VCP)).
not_stuck_c_pair_sv :
  not_stuck_c_pair (not_stuck_c_step SCP)
                   (not_stuck_c_val VCP)
                   (not_stuck_c_step (step_c_pair_2 SCP)).
not_stuck_c_pair_ss :
  not_stuck_c_pair (not_stuck_c_step SCP1)
                   (not_stuck_c_step SCP2)
                   (not_stuck_c_step (step_c_pair_2 SCP1)).

not_stuck_c_fst : not_stuck_c EC -> not_stuck_c (fst_c EC) -> type.
%mode not_stuck_c_fst +NSEP -NSEP'.
not_stuck_c_fst_val :
  not_stuck_c_fst (not_stuck_c_val (val_c_pair VCP2 VCP1))
                  (not_stuck_c_step (step_c_fst_1 VCP2 VCP1)).
not_stuck_c_fst_step :
  not_stuck_c_fst (not_stuck_c_step SCP)
                  (not_stuck_c_step (step_c_fst_2 SCP)).

not_stuck_c_snd : not_stuck_c EC -> not_stuck_c (snd_c EC) -> type.
%mode not_stuck_c_snd +NSEP -NSEP'.
not_stuck_c_snd_val :
  not_stuck_c_snd (not_stuck_c_val (val_c_pair VCP2 VCP1))
                  (not_stuck_c_step (step_c_snd_1 VCP2 VCP1)).
not_stuck_c_snd_step :
  not_stuck_c_snd (not_stuck_c_step SCP)
                  (not_stuck_c_step (step_c_snd_2 SCP)).

not_stuck_c_app : not_stuck_c EC1 ->
                  not_stuck_c EC2 ->
                  not_stuck_c (app_c EC1 EC2) ->
                  type.
%mode not_stuck_c_app +NSCP1 +NSCP2 -NSCP.
not_stuck_c_app_vv : not_stuck_c_app (not_stuck_c_val VCP1)
                                     (not_stuck_c_val VCP2)
                                     (not_stuck_c_step (step_c_app_1 VCP2)).
not_stuck_c_app_vs :
  not_stuck_c_app (not_stuck_c_val VCP)
                  (not_stuck_c_step SCP)
                  (not_stuck_c_step (step_c_app_2 SCP VCP)).
not_stuck_c_app_sv :
  not_stuck_c_app (not_stuck_c_step SCP)
                  (not_stuck_c_val VCP)
                  (not_stuck_c_step (step_c_app_3 SCP)).
not_stuck_c_app_ss :
  not_stuck_c_app (not_stuck_c_step SCP1)
                  (not_stuck_c_step SCP2)
                  (not_stuck_c_step (step_c_app_3 SCP1)).

not_stuck_c_case : not_stuck_c EC -> not_stuck_c (case_c EC EC1 EC2) -> type.
%mode +{W1 : world} +{TC1 : tp_c}
      +{EC : ec W1 int_c}
      +{EC1 : ec W1 TC1}
      +{EC2 : ec W1 int_c -> ec W1 TC1}
      +{NSEP : not_stuck_c EC}
      -{NSEP' : not_stuck_c (case_c EC EC1 EC2)}
     (not_stuck_c_case NSEP NSEP').
not_stuck_c_case_z : not_stuck_c_case (not_stuck_c_val val_c_z)
                                      (not_stuck_c_step step_c_case_1).
```

```
not_stuck_c_case_s : not_stuck_c_case (not_stuck_c_val (val_c_s VCP))
                                      (not_stuck_c_step (step_c_case_2 VCP)).
not_stuck_c_case_step :
  not_stuck_c_case (not_stuck_c_step SCP)
                   (not_stuck_c_step (step_c_case_3 SCP)).

not_stuck_c_let_box : not_stuck_c EC1 ->
                      not_stuck_c (let_box_c EC1 EC2) ->
                      type.
%mode +{W1 : world} +{TC1 : tp_c} +{TC2 : tp_c} +{CC : ctx_c}
      +{EC1 : ec W1 (code_c CC TC1)}
      +{EC2 : uvar CC TC1 -> ec W1 TC2}
      +{NSCP : not_stuck_c EC1}
      -{NSCP' : not_stuck_c (let_box_c EC1 EC2)}
      (not_stuck_c_let_box NSCP NSCP').
not_stuck_c_let_box_val :
  not_stuck_c_let_box (not_stuck_c_val (val_c_box : val_c (box_c W B)))
                      (not_stuck_c_step (step_c_let_box_1 SBP))
   <- subst_boxarg_total B EC2 SBP.
not_stuck_c_let_box_step :
  not_stuck_c_let_box (not_stuck_c_step SCP)
                      (not_stuck_c_step (step_c_let_box_2 SCP)).

% -- Progress property for contextual expressions:
progress_c : {EC : ec W TC} not_stuck_c EC -> type.
%mode progress_c +EC -NSCP.

progress_c_z : progress_c z_c (not_stuck_c_val val_c_z).

progress_c_s : progress_c (s_c EC) NSCP'
                <- progress_c EC NSCP
                <- not_stuck_c_s NSCP NSCP'.

progress_c_pair : progress_c (pair_c EC1 EC2) NSCP
                   <- progress_c EC1 NSCP1
                   <- progress_c EC2 NSCP2
                   <- not_stuck_c_pair NSCP1 NSCP2 NSCP.

progress_c_fst : progress_c (fst_c EC) NSEP'
                  <- progress_c EC NSEP
                  <- not_stuck_c_fst NSEP NSEP'.

progress_c_snd : progress_c (snd_c EC) NSEP'
                  <- progress_c EC NSEP
                  <- not_stuck_c_snd NSEP NSEP'.

progress_c_lam : progress_c (lam_c EC) (not_stuck_c_val val_c_lam).

progress_c_app : progress_c (app_c EC1 EC2) NSCP
                  <- progress_c EC1 NSCP1
                  <- progress_c EC2 NSCP2
                  <- not_stuck_c_app NSCP1 NSCP2 NSCP.

progress_c_case : progress_c (case_c EC EC1 EC2) NSEP'
                   <- progress_c EC NSEP
                   <- not_stuck_c_case NSEP NSEP'.

progress_c_fix : progress_c (fix_c EC) (not_stuck_c_step step_c_fix).

progress_c_box : progress_c (box_c W B) (not_stuck_c_val val_c_box).

progress_c_let_box : progress_c (let_box_c EC1 EC2) NSCP'
                      <- progress_c EC1 NSCP
                      <- not_stuck_c_let_box NSCP NSCP'.

%worlds (sbt_b1 | sbt_b2 | sbt_b3)
        (not_stuck_c_s _ _)
        (not_stuck_c_pair _ _ _)
        (not_stuck_c_fst _ _)
        (not_stuck_c_snd _ _)
        (not_stuck_c_app _ _ _)
        (not_stuck_c_case _ _)
```

```
        (subst_boxarg_total _ _ _)
        (subst_boxarg_boxarg_total _ _ _)
        (subst_boxarg_sequence_total _ _ _)
        (eval_clo_total _ _ _)
        (not_stuck_c_let_box _ _).
%worlds () (progress_c _ _).
%worlds () (big_to_small_c _ _).

%total NSCP (not_stuck_c_s NSCP _).
%total NSCP (not_stuck_c_pair NSCP _ _).
%total NSCP (not_stuck_c_fst NSCP _).
%total NSCP (not_stuck_c_snd NSCP _).
%total NSCP (not_stuck_c_app NSCP _ _).
%total NSCP (not_stuck_c_case NSCP _).
%total SC (eval_clo_total _ SC _).
%total (EC B SC)
       (subst_boxarg_total _ EC _)
       (subst_boxarg_boxarg_total _ B _)
       (subst_boxarg_sequence_total _ SC _).
%total NSCP (not_stuck_c_let_box NSCP _).
%total EC (progress_c EC _).
%total ECP (big_to_small_c ECP _).

% --- Translation of explicit expressions into contextual expressions

% -- Translation of types:
tr_tp_e~>c : tp -> tp_c -> type. %name tr_tp_e~>c TRTP.
%mode tr_tp_e~>c +T -TC.

tr_tp_e~>c_int : tr_tp_e~>c int int_c.

tr_tp_e~>c_product : tr_tp_e~>c (product T1 T2) (product_c TC1 TC2)
                     <- tr_tp_e~>c T1 TC1
                     <- tr_tp_e~>c T2 TC2.

tr_tp_e~>c_arrow : tr_tp_e~>c (arrow T1 T2) (arrow_c TC1 TC2)
                   <- tr_tp_e~>c T1 TC1
                   <- tr_tp_e~>c T2 TC2.

tr_tp_e~>c_code : tr_tp_e~>c (code T) (code_c ctx_c_0 TC)
                  <- tr_tp_e~>c T TC.

%worlds () (tr_tp_e~>c _ _).
%total T (tr_tp_e~>c T _).

% -- Translation of types - totality:
tr_tp_e~>c_total : {T} tr_tp_e~>c T TC -> type. %name tr_tp_e~>c_total TRTTP.
%mode tr_tp_e~>c_total +T -TRTP.

tr_tp_e~>c_total_int : tr_tp_e~>c_total int tr_tp_e~>c_int.

tr_tp_e~>c_total_product : tr_tp_e~>c_total (product T1 T2)
                                            (tr_tp_e~>c_product TRTP2 TRTP1)
                           <- tr_tp_e~>c_total T1 TRTP1
                           <- tr_tp_e~>c_total T2 TRTP2.

tr_tp_e~>c_total_arrow : tr_tp_e~>c_total (arrow T1 T2)
                                          (tr_tp_e~>c_arrow TRTP2 TRTP1)
                         <- tr_tp_e~>c_total T1 TRTP1
                         <- tr_tp_e~>c_total T2 TRTP2.

tr_tp_e~>c_total_code : tr_tp_e~>c_total (code T) (tr_tp_e~>c_code TRTP)
                        <- tr_tp_e~>c_total T TRTP.

%worlds () (tr_tp_e~>c_total _ _).
%total T (tr_tp_e~>c_total T _).

% -- Translation of types - uniqueness:
tr_tp_e~>c_unq : tr_tp_e~>c T TC -> tr_tp_e~>c T TC' -> eq_c_tp TC TC' -> type.
%mode tr_tp_e~>c_unq +TRTP +TRTP' -QP.

tr_tp_e~>c_unq_int : tr_tp_e~>c_unq tr_tp_e~>c_int tr_tp_e~>c_int eq_c_tp_r.
```

```
tr_tp_e~>c_unq_product : tr_tp_e~>c_unq (tr_tp_e~>c_product TRTP2 TRTP1)
                                        (tr_tp_e~>c_product TRTP2' TRTP1')
                                        QP
                         <- tr_tp_e~>c_unq TRTP1 TRTP1' QP1
                         <- tr_tp_e~>c_unq TRTP2 TRTP2' QP2
                         <- eq_c_tp_product QP1 QP2 QP.

tr_tp_e~>c_unq_arrow : tr_tp_e~>c_unq (tr_tp_e~>c_arrow TRTP2 TRTP1)
                                      (tr_tp_e~>c_arrow TRTP2' TRTP1')
                                      QP
                       <- tr_tp_e~>c_unq TRTP1 TRTP1' QP1
                       <- tr_tp_e~>c_unq TRTP2 TRTP2' QP2
                     <- eq_c_tp_arrow QP1 QP2 QP.

tr_tp_e~>c_unq_code: tr_tp_e~>c_unq (tr_tp_e~>c_code TRTP)
                                    (tr_tp_e~>c_code TRTP')
                                    QP'
                     <- tr_tp_e~>c_unq TRTP TRTP' QP
                     <- eq_c_tp_code QP QP'.

%worlds () (tr_tp_e~>c_unq _ _ _).
%total TRTP (tr_tp_e~>c_unq TRTP _ _).

% -- Translation of expressions:
tr_e~>c : ee W T -> tr_tp_e~>c T TC -> ec W TC -> type. %name tr_e~>c TRECP.
%mode tr_e~>c +EE -TRTP -EC.

tr_e~>c_z : tr_e~>c z_e tr_tp_e~>c_int z_c.

tr_e~>c_s : tr_e~>c (s_e EE) tr_tp_e~>c_int (s_c EC)
            <- tr_e~>c EE tr_tp_e~>c_int EC.

tr_e~>c_pair : tr_e~>c (pair_e EE1 EE2)
                       (tr_tp_e~>c_product TRTP2 TRTP1)
                       (pair_c EC1 EC2)
               <- tr_e~>c EE1 TRTP1 EC1
               <- tr_e~>c EE2 TRTP2 EC2.

tr_e~>c_fst : tr_e~>c (fst_e EE) TRTP1 (fst_c EC)
              <- tr_e~>c EE (tr_tp_e~>c_product TRTP2 TRTP1) EC.

tr_e~>c_snd : tr_e~>c (snd_e EE) TRTP2 (snd_c EC)
              <- tr_e~>c EE (tr_tp_e~>c_product TRTP2 TRTP1) EC.

tr_e~>c_lam : tr_e~>c (lam_e EE) (tr_tp_e~>c_arrow TRTP2 TRTP1) (lam_c EC)
              <- tr_tp_e~>c_total T1 TRTP1
              <- ({x} {y} tr_e~>c x TRTP1 y -> tr_e~>c (EE x) TRTP2 (EC y)).

tr_e~>c_app : tr_e~>c (app_e EE1 EE2) TRTP2 (app_c EC1 EC2')
              <- tr_e~>c EE1 (tr_tp_e~>c_arrow TRTP2 TRTP1) EC1
              <- tr_e~>c EE2 TRTP1' EC2
              <- tr_tp_e~>c_unq TRTP1' TRTP1 QP
              <- eq_c_tp_e QP EC2 EC2'.

tr_e~>c_case : tr_e~>c (case_e EE EE1 EE2) TRTP (case_c EC EC1 EC2')
               <- tr_e~>c EE tr_tp_e~>c_int EC
               <- tr_e~>c EE1 TRTP EC1
               <- ({x} {y} tr_e~>c x tr_tp_e~>c_int y ->
                           tr_e~>c (EE2 x) TRTP' (EC2 y))
               <- tr_tp_e~>c_unq TRTP' TRTP QP
               <- ({y} eq_c_tp_e QP (EC2 y)(EC2' y)).

tr_e~>c_fix : tr_e~>c (fix_e EE) TRTP (fix_c EC')
              <- tr_tp_e~>c_total T TRTP
              <- ({x} {y} tr_e~>c x TRTP y -> tr_e~>c (EE x) TRTP' (EC y))
              <- tr_tp_e~>c_unq TRTP' TRTP QP
              <- ({y} eq_c_tp_e QP (EC y)(EC' y)).

tr_e~>c_box : tr_e~>c (box_e EE)
                      (tr_tp_e~>c_code TRTP)
                      (box_c W ([w] (boxarg_0 (EC w))))
```

```
              <- ({w} tr_e~>c (EE w) TRTP (EC w)).

tr_e~>c_let_box : tr_e~>c (let_box_e EE1 EE2) TRTP2 (let_box_c EC1 EC2)
                   <- tr_e~>c EE1 (tr_tp_e~>c_code TRTP1) EC1
                   <- ({u} {v}
                       ({w} tr_e~>c (u w) TRTP1 (clo v sequence_c_0)) ->
                       tr_e~>c (EE2 u) TRTP2 (EC2 v)).

%block trec_b1 :
  some {W : world} {T : tp} {TC : tp_c} {TRTP : tr_tp_e~>c T TC}
  block {x : ee W T} {y : ec W TC} {trecp : tr_e~>c x TRTP y}.
%block trec_b2 : block {w : world}.
%block trec_b3 :
  some {T : tp} {TC : tp_c} {TRTP : tr_tp_e~>c T TC}
  block {u : {w : world} ee w T} {v : uvar ctx_c_0 TC}
        {trecp : {w : world} tr_e~>c (u w) TRTP (clo v sequence_c_0)}.
%block trec_b4 : some {W : world} {TC : tp_c}
                 block {y : ec W TC}.

%worlds (trec_b1 | trec_b2 | trec_b3 | trec_b4)
        (eq_c_tp_e _ _ _)
        (tr_e~>c _ _ _).
%total {} (eq_c_tp_e _ _ _).
%total EE (tr_e~>c EE _ _).
```

A.7 examples.elf

The file examples.elf contains a few examples.

```
plus_e =
  lam_e [m : ee W int]
        (lam_e [n : ee W int]
               (app_e (fix_e
                         [p : ee W (arrow int int)]
                         (lam_e
                            [m' : ee W int]
                            (case_e m'
                                    n
                                    ([m'-1 : ee W int] (s_e (app_e p m'-1))))))
                      m)).

%query 1 * eval_e (app_e (app_e plus_e (s_e (s_e z_e))) (s_e (s_e (s_e z_e))))
                  VC.

times_e =
  lam_e [m : ee W int]
        (lam_e [n : ee W int]
               (app_e (fix_e
                         [p : ee W (arrow int int)]
                         (lam_e
                            [m' : ee W int]
                            (case_e m'
                                    z_e
                                    ([m'-1 : ee W int]
                                     (app_e (app_e plus_e n) (app_e p m'-1))))))
                      m)).

%query 1 * eval_e (app_e (app_e times_e (s_e (s_e z_e))) (s_e (s_e (s_e z_e))))
                  VC.

power_e =
  fix_e
    [p]
    (lam_e [n]
           (case_e n
                   (box_e ([w] lam_e [x] s_e z_e))
                   ([m]
                    let_box_e
```

```
                    (app_e p m)
                    ([u] box_e ([w] (lam_e [x] (app_e (app_e times_e x)
                                                      (app_e (u w) x))))))).

%query 1 * eval_e (app_e power_e (s_e (s_e z_e))) VC.

plus_c =
  lam_c [m : ec W int_c]
        (lam_c [n : ec W int_c]
               (app_c (fix_c
                         [p : ec W (arrow_c int_c int_c)]
                         (lam_c
                            [m' : ec W int_c]
                            (case_c m'
                                    n
                                    ([m'-1 : ec W int_c] (s_c (app_c p
                                                                     m'-1))))))
                      m)).

times_c =
  lam_c [m : ec W int_c]
        (lam_c [n : ec W int_c]
               (app_c (fix_c
                         [p : ec W (arrow_c int_c int_c)]
                         (lam_c
                            [m' : ec W int_c]
                            (case_c m'
                                    z_c
                                    ([m'-1 : ec W int_c]
                                     (app_c (app_c plus_c n) (app_c p m'-1))))))
                      m)).

power_c =
  fix_c
    [p]
    (lam_c
       [n]
       (case_c
          n
          (box_c world_0 ([w] boxarg_n ([x] (boxarg_0 (s_c z_c)))))
          ([m]
           let_box_c
             (app_c p m)
             ([u] (box_c
                     world_0
                     ([w] (boxarg_n
                             ([x] (boxarg_0
                                     (app_c
                                        (app_c times_c x)
                                        (clo u (sequence_c_n x
                                                             sequence_c_0)))))))
                  ))))).

%query 1 * eval_c (app_c power_c (s_c (s_c z_c))) VC.

plus_t =
  lam_t [m : et W int_t]
        (lam_t [n : et W int_t]
               (app_t (fix_t
                         [p : et W (arrow_t int_t int_t)]
                         (lam_t
                            [m' : et W int_t]
                            (case_t m'
                                    n
                                    ([m'-1 : et W int_t] (s_t (app_t p m'-1))))
                         ))
                      m)).

times_t =
  lam_t [m : et W int_t]
        (lam_t [n : et W int_t]
               (app_t (fix_t
```

```
                        [p : et W (arrow_t int_t int_t)]
                        (lam_t
                           [m' : et W int_t]
                           (case_t m'
                                   z_t
                                   ([m'-1 : et W int_t]
                                    (app_t (app_t plus_t n) (app_t p m'-1))))))
                     m)).

power_t = [times]
  lam_t
    [n : et wumber_0 int_t]
    (next
       (lam_t
          [x : et (wumber_n wumber_0) int_t]
          (prev
             (app_t (fix_t
                       [p : et wumber_0 (arrow_t int_t (circle int_t))]
                       (lam_t
                          [n' : et wumber_0 int_t]
                          (case_t n'
                                  (next (s_t z_t))
                                  ([n'-1 : et wumber_0 int_t]
                                   (next (app_t (app_t times x)
                                                (prev (app_t p n'-1))))))))
                    n)))).

%query 1 * ETP : eval_t W (app_t (power_t times_t) z_t) VC.

%query 1 * eval_t wumber_0
                  (next (lam_t
                           [times]
                           (prev (app_t (power_t times) (s_t (s_t z_t))))))
                  VC.
```