A Pedagogical Semantics-Directed Metalanguage

Georgian-Vlad Saioc

Aalborg University
Computer Science

AALBORG UNIVERSITY

STUDENT REPORT

Title:
A Pedagogical Semantics-Directed Metalanguage

Theme:
Scientific Theme

Project Period:
Spring Semester 2020

Participant:
Georgian-Vlad Saioc

Supervisor:
Hans Hüttel

Copies: 1

Pages: 105

Date of Completion:
June 12, 2020

Abstract:

This report presents a metalanguage based on natural semantics. It is designed specifically with pedagogical purposes in mind.
The focus of the report it lies in presenting the specifications of the metalanguage, as well as the underlying implementation.

Contents

Preface

This report is written for the fourth semester of the Master's program in Computer Science at Aalborg University, while following the specialization of *Programming Technology*.

It is the second part of a two-report series on tools for building compilers and interpreters. It makes use of the foundation established in the previous report in an attempt to document the construction of such a tool meant to be used in a pedagogical environment.

Aalborg University, June 12, 2020

Georgian-Vlad Saioc
gsaioc18@student.aau.dk

Acknowledgements

I would like to express my gratitude towards my supervisor, Hans Hüttel, for the valuable guidance and constructive criticism offered during the development of the associated project, and this last year of study.

I also wish to thank Bent Thomsen and Lone Leth Thomsen for participating in the survey mentioned in the introduction, and contributing with additional insight.

Last but not least, many thanks are due to my family and friends for ample support during this period.

Chapter 1

Introduction

The development of high-level programming languages is a central contributor to the significant evolution in interfacing between humans and computers over the past decades. A natural consequence is the prevalence of interpreters, compilers and transpilers (i. e. translation tools[22]) in the digital landscape. They allow programming language users to quickly express and reason about algorithms and behaviors, while abstracting over the concrete physical and logical implementation details.

In the author's opinion, up-and-coming programmers, developers, and computer scientists could potentially build a stronger intuition about many programming technologies they use on a near-daily basis, through a more fine-grained understanding of the anatomy of high-level languages. Academic circles agree, considering many universities have their curricula include courses on programming languages as a concept and compiler construction[1], with some also allowing students the opportunity to work on projects consisting of designing languages and implementing corresponding translation tools.

1.1 Problem statement

At the time of writing, translation tool front-end, i. e. lexical analysis and parsing, have been successfully automated, underpinned by the theory behind regular and context-free languages[23].

Progressing towards the back end, there is increasingly less homogeneity and accessibility. For descriptions of language semantics, several approaches[22] are used, the two most popular being:

- Implementation by way of a general-purpose language as a standalone pro-

gram, this being the most popular.

- Semantics formalisms, with operational semantics being particularly popular due to its intuitive layout, and closely followed by denotational semantics, for more mathematically inclined designers. Normally, these semantics are given in pand-and-paper, but there is a variation[20, 9, 27, 19] involving automated implementations of metalanguages based on semantics formalisms. However, this latter approach is less popular.

As mentioned before, curricula also include projects revolving around designing languages and building compilers. It is expected that, in such a context, the student body will naturally gravitate towards one of these two approaches for designing language, and, for those opting for semantics formalisms, pen-and-aper is still the first choice. In such a case, even those opting to use a semantics formalism for the design, ultimately do the implementation in a general-purpose language.

1.1.1 Common misgivings in student projects

As corroborated in a survey with part of the professor body at Aalborg University, there are several reoccurring grievances when it comes to projects about programming languages:

1. One of them is the distance between the pen-and-paper semantics specifications and the actual implementations. In fact, when pen-and-paper specifications are forgone completely in favor of direct implementation, the result can suffer from many unintended behaviors, and time used in wrestling with the features of the general-purpose language.
2. If a semantics formalism is used, the absence of debugging for pen-and-paper specifications can lead to many subtle mistakes. Typically, operational semantics is used, with one particularly popular style of errors being ill-formed transition systems.

> **Example 1.1 (Ill-formed transition systems)**
> Consider a natural semantics transition system for basic imperative statements[1]. It involves transitions from an initial configuration, formed from a statement and a state, to a final configuration, giving the updated state.
>
> An example of a blatant violation of this constraint would be a rule such as:
>
> $$\text{[MISSING-STATE]:} \quad \mathtt{S} \rightarrow s'$$
>
> In this single statement axiom, the statement is present in the initial configuration, but the state is missing.

Similar errors arise from mixing different transition systems. Consider an imperative **if** statement. Typical representations involve evaluating a boolean expression, e, and executing the statements corresponding to the appropriate branch. Taking the case where the guard is true, a student might erroneously give a rule as:

$$\text{[IF-TRUE]:}\quad \frac{s \vdash \mathtt{e} \to tt \qquad \langle \mathtt{S}_1,\ s\rangle \to s'}{\langle \textbf{if } \mathtt{e} \textbf{ then } \mathtt{S}_1 \textbf{ else } \mathtt{S}_2,\ s\rangle \to s'}$$

In the above, the transition system is supposed to consider both transitions from statement-state pairs to states, and the evaluation of e to a boolean value, given state s. However, there are marked differences between the definitions of transitions for expression evaluation and statement execution:

- The syntactic categories that e and S belong to are different, thus observing different formation rules.
- Expression evaluations are given with state s as an environment, whereas executions include it in the configuration, and have no environment attached.
- Final configurations belong to different sets, with expressions evaluating to a set that at least includes boolean values, whereas statements evaluate to updated states.

While there would be ways to reconcile these differences within the same transition system (e. g. defining the set of final configurations to include both boolean values and states, introducing a formation rule $\mathtt{S} ::= \mathtt{e}$, defining final configurations as a union between the set of values and the set of states), it would be much cleaner to give a separate transition system for handling expression evaluations. increasing both modularity and readability.

Another example, stated to be from an actual 4^{th} semester project, is as follows:

$$\frac{\langle \mathtt{D}_\mathtt{F}, env_\mathtt{F}[\mathtt{f} \mapsto \mathtt{S}]\rangle \to env'_\mathtt{F}}{env_\mathtt{V} \vdash \langle \textbf{fun } \mathtt{f} \textbf{ is } \mathtt{S}\ \mathtt{D}_\mathtt{F}, sto, env_\mathtt{F}\rangle \to \langle sto, env'_\mathtt{F}\rangle}$$

This rule is highly similar to PROC-BIP$_{\text{BSS}}$ for the language **Bip**[12], representing an update of the procedure environment by binding variables to statements in an imperative language. In this case we have several differences between the premise and conclusion transitions, namely:

- The binding environment is different, where variable environment $env_{\mathtt{V}}$ is absent from the premise entirely.
- The members of the initial configurations differ, where in the conclusion we have a 3-tuple, whereas the premise is a pair, lacking store *sto*. In the textbook version of the rule, the store is absent, but regardless of the intention behind including it in the conclusion, the discrepancy between the conclusion and premise means the student-given rule is ill-formed.
- The final configurations differ, with the premise giving evaluations to a function environment $env'_{\mathtt{F}}$, whereas in the conclusion, the initial configuration evaluates to a pair between *sto* and $env'_{\mathtt{F}}$.

3. Another problem arises from students leaning too heavily on parser generators during the initial stages of language design, overplaying the importance of concrete syntax, even though the abstract syntax ultimately informs the semantics of a language. It would be beneficial to encourage usage of the latter in the initial stages, and simply extrapolating the concrete syntax from the abstract syntax, as opposed to agonizing over syntactical details.

1.1.2 Contrary considerations

As a solution to the problems expressed in section 1.1.1, an emphasis on the option of implementation by metalanguage was presented. However, participants in the survey have presented reasonable points about the negative aspects of such an approach. Among these is the notion that any introduced layer of abstraction, while alleviating the requirement of dealing with implementation details, still comes at the cost of sacrificing some level of control and/or power over the underlying systems, perhaps even obfuscating the understanding of said systems, or reducing its efficiency.

Participants have also emphasised that this ties the language design potential to limitations introduced by the formalism chose for the metalanguage. This goes to show that the most popular formalisms, while powerful, are not universal solutions. Examples of limitations are the inability of traditional natural semantics in differentiating between failing and non-terminating transitions (later addressed with the introduction of pretty-big-step semantics[5]), or the lack of modular design for denotational semantics (partially addressed with higher-order semantics[17]) and difficulties with non-determinism. Concluding, several partici-

[1] Appendix D.1

pants have expressed their opinion that implementing semantics through general-purpose languages is ultimately an acceptable, if more laborious, solution.

1.2 Proposal

Several key points are outlined in sections 1.1.1 and 1.1.2:

- No universal solution for expressing program semantics has presented itself. Each introduced formalism features different benefits and drawbacks.
- There are doubts when it comes to replacing general-purpose language implementations of semantics in compilers and interpreters with a particularized tool, as was the case with parser generators. Despite the overhead involved in such implementations, it is ultimately a more complete solution.
- Operational semantics is frequently used to great effect by students, even if in its pen-and-paper form. However, the pen-and-paper format lends itself to easily overlooked errors, a particularly pernicious one being ill-formed transition systems.

Considering this, the proposed solution is a metalanguage interpreter based on natural semantics as a supplementary tool in language construction. Its purpose is rather checking for specification correctness, as opposed to replacing general-purpose implementations. Concretely, the stated goals are:

- In terms of functionality, allowing verification of properties of a semantics specification, such as well-formedness, and allowing evaluations in said language, comparing the results yielded from the implemented semantics with the intended behaviors. This narrower focus detracts from other concerns, such as execution efficiency.
- In terms of readability, creating a syntax tailored to resemble pen-and-paper specifications as much as possible, by including specific constructs for definitions of abstract syntax and transition systems. Making the syntax simple can make it more attractive for students when trying to assist themselves with the tool.

The solution is not meant to be universal, but to allow sufficient flexibility to assist in a student project. As additional features, the language is equipped with:

- λ-expressions, through an extension of the simply-typed lambda calculus. These allow declarative descriptions of behaviors similar to textbooks[12], where rules are not concerned.
- Optional pretty printing, by generating a reasonable quality `.tex` file from

the interpreted specifications, which can be grafted onto a student report, although some manual interference might be necessary when typesetting.

Similar tools in this area are RML[20], Typol[27] and LETOS[9]. These are much more mature when it comes to:

- Pretty printing in the case of Typol and LETOS, e. g. using invisible constructors for representing abstract syntax, and in general being focused on publication-level rendering
- Rule selection, especially in the case of Typol, which uses Prolog as an underlying system for applying rules[2].
- Tracing, e. g. LETOS generates web pages from derivation trees that can be analyzed with Netscape.
- Specification modularity, where RML can split a language definition across multiple files.
- Compilers or interpreter generation, absent in the current iteration of the metalanguage, e. g. RML[20] is focused especially on generating efficient compilers implemented in C[16], LETOS generates Miranda[18] code.

This goes to show that many improvements can still be made to the current iteration when it comes to more advanced features of natural semantics implementations. On the other hand, taken on a case by case basis, the metalanguage has the following advantages:

- Against RML, none of the investigated sources have mentioned pretty printing. Perhaps RML is the closest in terms of syntax, but opts to express transition systems as functions, and lacks simple features, such as labeling of rules, outside of using comments.
- Against Typol, its reliance on Prolog and logic programming in general could give perhaps a too steep barrier of entry to 4^{th} semester students. Logic programming stands out as being a less popular programming paradigm than the functional and imperative ones. A similar argument could have been made in previous years[20] about the functional paradigm. However, the advent of web technologies relying on asynchronous, event-driven programming, has led to a surge in popularity[26] of languages like Python[21] and JavaScript[14]. While these languages are not purely functional, being mixed paradigm instead, they do employ many features such as similar scoping rules, anonymous functions and closures. As such, a student is likely to

[2]More on this in section 3.2.4

be less resistant when introduced to the simple functional capacities of the metalanguage.

- LETOS is the closest to completely overshadowing the proposal. Perhaps the biggest caveat with LETOS is taking as input Miranda code interpolated in a LaTeX document. The proposed metalanguage automatically generates all the LaTeX boilerplate, albeit not at publication quality, allowing for cleaner input and separation between specifications and pretty printing. Another caveat of LETOS is its more involved syntax when it comes to defining abstract syntax, again as a requirement for pretty printing. Considering these facts, it would be markedly easier and quicker to start defining a specification in the proposed metalanguage for a 4^{th} semester student, if at the cost of losing power.

1.3 Overview

The following chapters will explore in-depth the metalanguage and related concepts. Chapter 2 covers its specifications, detailing essential elements in the abstract syntax, semantics, and the type system. It begins by expressing the top-level abstract syntax, named the spine, followed by an elaboration of the types, represented by domains and syntax definitions as abstract data types. Proceeding to the syntax, semantics and type system of expressions, for the latter two, the unique constructs of the metalanguage are presented. Finally, the constructs relevant to expressing natural semantics are explored: configurations and transition systems. Having established all these notions, the top-level semantics is given, showing how all of these constructs tie together to express language specifications.

Chapter 3 gives a brief overview of the implementation and other topics, such as design decisions, and how they play into the pedagogical considerations. Other areas deliberated on include limitations of the metalanguage and potential improvements. These are used to draw the conclusions and inform the future work presented in chapter 4.

The appendix covers additional information, such as generic semantics and type rules used by the metalanguage λ-expressions, several proofs, and example specifications, showing how to use the metalanguage, and the generated LaTeX output.

Chapter 2

Metalanguage specifications

This chapter includes a presentation of the abstract syntax and semantics of the metalanguage. It is heaavily inspired by, and structured in a manner similar to Lawrence Paulson's compiler generator[19], but opts for implementing a natural semantics as a formalism, as opposed to semantic grammars.

2.1 Preliminaries

Much of the notation for describing the abstract syntax and transition systems uses notational conventions established in literature on natural semantics[15, 12] studied by the author. However, to increased conciseness, some additions are introduced.

Syntactic categories are occasionally introduced in the format $\mathsf{x} \in \mathbf{X}$, where $\mathbf{X}$ represents the syntactic category itself, while x is a metavariable used when expressing formation and transition rules.

Transitions systems are named by using bolded and italicized camel case, e. g. ***Exp***, ***Rule***. Typically, syntactic categories share the name with transition systems describing their semantics.

All transition systems have their signature denoted in the format:

$$\boldsymbol{X} : E \vdash C_1 \Downarrow C_2$$

which is to be read as: transitions in $\boldsymbol{X}$ use a binding model with domain E, evaluating configurations with domain C_1 to configurations with domain C_2. The binding model can be omitted for transition systems that do not involve one. Mirroring this, transitions found in the axiom and rule definitions of $\boldsymbol{X}$ are written as:

$$e \vdash c_1 \Downarrow c_2$$

where e, c_1, and c_2 have domains E, C_1 and C_2, respectively. Optionally, rules and axioms in a transition system can be labeled. For composite terms, if one of its components is not used in a rule or axiom, it will be denoted as an underscore.

Axioms and rules can optionally have side conditions and local variable declarations, or refer to other transition systems. The latter scenario is denoted as:

$$e' \vdash c_1' \Downarrow_{X'} c_2'$$

re-contextualizing the meaning of the $\Downarrow$ operator to represent transitions in $\boldsymbol{X'}$, thus following its rules.

Similar notations are used for type systems, except the $\Downarrow$ operator is exchanged for ::. For example, for type system $\boldsymbol{T}$, the signature would be written as:

$$\boldsymbol{T} : E \vdash C :: T$$

where T denotes a domain of types. Annotating the turnstile with some variable, means type judgements use it in some capacity. The typical use case is a type environment, binding type variables.

2.1.1 Notions on algebraic vectors

Some of the formation rules for abstract syntax use vector notation to denote zero or more children, essentially resulting in an n-ary tuple. To shorten the notation when there is a need to unfurl the vector to its individual components, the following notation is used:

$$\overrightarrow{x} \text{ can be written as } \overrightarrow{x_i}^{n}$$

specifying that index i is a natural number from 1 to n, and component x_i corresponds to the i-th member of the vector. Special mention goes to the fact that indexes can be used outside of the scope of the vectors with special rules for quantifiers and iterators[1]. A 0-dimensional vector is expressed as $\overrightarrow{0}$.

If x would be a composite term, $a\ b$, where individual components are used in different contexts, the index is applied recursively to each component, for example:

$$\overrightarrow{a\ b} \text{ would be written as } \overrightarrow{a_i\ b_i}^{n}$$

[1] Example **2.1** illustrates some uses

Example 2.1 (Unfurling vectors)
Consider some rule involving $\overrightarrow{x}$ written as:

$$\frac{\forall i : x_i \Downarrow v_i}{\overrightarrow{x_i^n} \Downarrow \sum_i v_i}$$

The rule is satisfied if, for every component x_i, it is that the case that it evaluates to some value v_i. When used with a quantifier such as $\forall i$, it is a shorthand for $\forall i,\ i \in \mathbb{N}, 1 \leq i \leq n$. When used in an iteration, such as $\sum_i$, it is equivalent to $\sum_{i=1}^{n}$, in this case indicating that the final result is the sum of all values v_i.

Components can be added to vectors with the following notations:

$$\overrightarrow{x_i, x'_1, .., x'_m}^{\,n} \quad \overrightarrow{x'_1, .., x'_m, x_i}^{\,n}$$

The operation on the left creates a vector of $n + m$ dimensions by adding each x'_j, $1 \leq j \leq m$, component at the end of the original vector of size n. Conversely, the operation on the right creates a vector of size $n + m$, adding the components at the beginning.

2.1.2 Common syntactic categories

Below are some generic syntactic categories and their corresponding metavariables, either representing constant values, or which are used repeatedly throughout the metalanguage specification:

- `n` ∈ **Int** - represents the syntactic category of integer literals.
- `s` ∈ **String** - denotes the syntactic category of string literals.
- `y` ∈ **Symbol** - denotes the syntactic category of symbol literals. Symbols are represented similarly to strings, but lack any operations and use a restricted alphabet. Their purpose is denoting variables in the source language that a specification represents, as opposed to variables used in the specification proper.
- `x` ∈ **Var** - denotes the syntactic category of variables and metavariables used in various definitions, e. g. domain definitions, expressions, transition system names, rule labels. Considering its prevalence in several different formation rules, and that some even use multiple instances of `x` for different purposes, it will occasionally be superscripted, generally with the parent metavariable,

to help in differentiation, e. g. for some formation rule `M` ::= `x`, if `x` is encountered in a rule with an `x` formed through a different syntactic category, the former will be represented as $\mathtt{x}^{\mathtt{M}}$.

2.2 The abstract syntax of specifications

A specification consists of a sequence of definitions, followed by a sequence of evaluations using the definitions. These sequences form the what we will call the spine of a specification.

Definitions are split between four major categories:

- *Data*, computed from expressions given as an extended variation on the traditional λ-calculus. These consist of variables local to the specification that can be used throughout. An example use case for a metalanguage user is defining custom, reusable functions and constants.
- *Domains*, expressing the types of data that are used when representing functions, configurations, transition system signatures, etc. Built-in basic domains are those representing integers, booleans, strings, and symbols. Advanced custom domains can be composed through disjoint union, mapping and cross-product.
- *Abstract syntax formation rules*, for defining algebraic data types through which one can codify the structure of the abstract syntax in the source language.
- *Transition systems*, describing the semantics of the source language defined by the specifications. They are given as inference rules over transitions in a big-step semantics.

The spine is described by several syntactic categories, defined as:

- `Sp` $\in$ **Spec** - is the syntactic category describing a complete specification.
- `Df` $\in$ **Def** - is the syntactic category of definitions in the specification.
- `Ev` $\in$ **Eval** - denotes an evaluation in the specification.

The formation rules for these syntactic categories are defined as follows:

$$
\begin{aligned}
\texttt{Sp} ::= \ & \vec{\texttt{Df}}\ \vec{\texttt{Ev}} \\
\texttt{Df} ::= \ & \textsf{let}\ \texttt{x} := \texttt{e} \\
& \mid \textsf{letrec}\ \texttt{x}_1 : \texttt{D} := \texttt{x}_2\ .\ \texttt{e} \\
& \mid \textsf{domain}\ \texttt{x}^{\texttt{D}} := \texttt{D} \\
& \mid \textsf{syntax}\ \texttt{x}^{\texttt{S}} := \texttt{S} \\
& \mid \textsf{system}\ \texttt{x}^{\texttt{T}} : \texttt{D}_1\ \texttt{|-}\ \texttt{D}_2 \texttt{=>}\ \texttt{D}_3 := \texttt{T} \\
\texttt{Ev} ::= \ & \textsf{evaluate}\ \texttt{e}_1\ \texttt{|-}\ \texttt{e}_2\ \textsf{in}\ \texttt{x}^{\texttt{T}} \\
& \mid \textsf{evaluate}\ \texttt{e}
\end{aligned}
$$

All as yet introduced metavariables, their syntactic categories, and the appropriate semantics, will be covered in the following sections. The semantics of specification spines are given in section 2.7, following these introductions.

2.3 Types

A specification can include definitions for domains and abstract syntax formation rules. Together, these form the types of a specification, used when defining data or transition systems. In section 1.1.1, the notion of ill-formed transitions was brought up. In the metalanguage, an ill-formed transition would essentially equate to an *ill-typed* one, and these types serve as the building blocks for the type systems later introduced.

2.3.1 Domains

Domains are one of the types introduced in the metalanguage. The terms domain and type will be used interchangeably throughout the rest of the report. The abstract syntax of domains is formed from several syntactic categories:

- $\texttt{D} \in$ **Dom** - represents the syntactic category of domains, with its formation rules reflecting aliases, mappings (functions), products, and disjoint unions.
- $\texttt{B} \in$ **Bdom** - forms the syntactic category of basic domains.
- $\texttt{t} \in$ **Tag** - is the syntactic domain of tags used in denominating different components in a disjoint union.[2]

[2]Syntactically, it is the case that **Tag** is identical to **Var**.

The formation rules of domains are expressed as follows:

$$\begin{aligned}
\mathtt{D} ::= {} & \mathtt{B} \mid \mathtt{x} \\
& \mid \mathtt{D}_1 \bar{\rightarrow} \mathtt{D}_2 \mid \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \\
& \mid \bar{\bigcup} \overrightarrow{\mathtt{t}\ \textsf{as}\ \mathtt{D}} \\
\mathtt{B} ::= {} & \textsf{int} \mid \textsf{bool} \mid \textsf{str} \mid \textsf{sym} \mid \epsilon
\end{aligned}$$

The basic domains **int**, **bool**, **str** and **sym** represent integers, booleans, strings and symbols respectively. The empty domain is denoted as ϵ. It cannot be explicitly expressed by a metalanguage user in the syntax of a specification, but becomes useful when describing some behaviors in the metalanguage, e. g. tag injection, discussed in section 2.4.3.3.

2.3.2 Abstract syntax definitions

Metalanguage users can define abstract syntax in their specifications, allowing the semantics to be syntax-directed. This involves specifying syntactic categories and their formation rules. Syntax definitions are evaluated to a union domain, where its formation rules are the components of the union, the constructors becoming tags, and the metavariables (if any) being represented by a domain value.

2.3.2.1 Abstract syntax

Its syntactic category is denoted as $\mathtt{S} \in \textsf{Syntax}$, and its members are term constructors. The formation rules are:

$$\mathtt{S} ::= \overrightarrow{\mathtt{t}\ \textsf{of}\ \mathtt{D}}$$

2.3.2.2 Semantics

The operational semantics of abstract syntax definitions relies on the following transition system:

$$\textit{Syntax} : \textsf{Syntax} \Downarrow \textsf{Dom}$$

The semantics of abstract syntax definitions involves creating a member of the **Dom** syntactic category from equivalent members of **Syntax** through the following axiom:

$$\text{[SYNTAX]:} \quad \overrightarrow{(\mathtt{t}\ \textsf{of}\ \mathtt{D})} \Downarrow \bar{\bigcup} \overrightarrow{(\mathtt{t}\ \textsf{as}\ \mathtt{D})}$$

Example 2.2 (Abstract syntax transformation)
Consider a subset of the formation rules for boolean expressions in **Bims** (appendix D.1). Formally, through $\mathtt{b} \in \mathbf{Bexp}$ and $\mathtt{a} \in \mathbf{AExp}$, the syntactic category of arithmetic expressions, they would be expressed as:

$$\begin{aligned} \mathtt{b} ::= \ & \mathbf{true} \mid \mathbf{false} \\ & \mid \mathit{and}\ \mathtt{b}_1\ \mathtt{b}_2 \\ & \mid \mathit{eq}\ \mathtt{a}_1\ \mathtt{a}_2 \end{aligned}$$

In the metalanguage, the abstract syntax tree of this syntax definition is:

$$(\mathit{bconst}\ \mathbf{of\ bool},\ \mathit{and}\ \mathbf{of\ Bexp} \bar{\times} \mathbf{Bexp},\ \mathit{eq}\ \mathbf{of\ AExp} \bar{\times} \mathbf{AExp})$$

which, given axiom SYNTAX, would evaluate to:

$$\bar{\bigcup}\ (\mathit{bconst}\ \mathbf{as\ bool},\ \mathit{and}\ \mathbf{as\ Bexp} \bar{\times} \mathbf{Bexp},\ \mathit{eq}\ \mathbf{as\ AExp} \bar{\times} \mathbf{AExp})$$

2.3.3 General notions and operations

Domain environments are expressed through domain $\bar{\Delta}$, defined as:

$$\bar{\Delta} = \mathbf{Var} \rightharpoonup \mathbf{Dom}$$

where members of $\bar{\Delta}$, denoted as Δ, are bindings from domain variables to domains definitions.

The following set represents all union domains:

$$\mathbf{Dom}_{\bar{\cup}} = \{\ \mathtt{D} \mid \mathtt{D} \in \mathbf{Dom},\ \mathtt{D}\ \text{is a union domain term}\ \}$$

One useful operation is determining whether a tag-domain pair belongs to a specific union domain. This operation, denoted as $\exists_{\mathtt{t}}$, has the following signature:

$$\exists_{\mathtt{t}} : (\mathbf{Tag} \times \mathbf{Dom} \times \mathbf{Dom}) \to \{tt, ff\}$$

and is defined as:

$$\exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}', \mathtt{D}) = \begin{cases} tt, & \text{if } \mathtt{D} = \bar{\bigcup}\ \overrightarrow{\mathtt{t}_i\ \mathbf{as}\ \mathtt{D}_i}^{\,n} \ \wedge\ \mathtt{t}\ \mathbf{as}\ \mathtt{D}' \in \bigcup_i \{\mathtt{t}_i\ \mathbf{as}\ \mathtt{D}_i\} \\ ff, & \text{otherwise} \end{cases}$$

Another operation, denoted as $\Delta_{\perp}$, is finding the "bedrock" domain of a domain variable. Given a domain environment, the function will substitute variables with

their definitions until a non-variable is encountered. This is useful for determining the first non-variable definition in a chain of domain variable synonyms. It has the following signature:

$$\Delta_\perp : (\bar{\Delta} \times \mathbf{Dom}) \rightharpoonup \mathbf{Dom}$$

The definition of $\Delta_\perp$ is:

$$\Delta_\perp(\Delta, \mathtt{D}) = \begin{cases} \mathtt{D} \text{ if } \mathtt{D} \neq \mathtt{x} \\ \Delta_\perp(\Delta, \Delta(\mathtt{D})) \text{ if } \mathtt{D} = \mathtt{x} \end{cases}$$

2.3.4 Well-formed domains

Part of the static analysis on domain definitions and expressions is ensuring that domains are well-formed. The foremost criterion is that, given Δ, there are no free domain variables in a specification, i. e. $\Delta(\mathtt{x})$ is defined for all used $\mathtt{x}$. Naturally, this applies to any domain expression, including anonymous annotations (i. e. used inline), for example when declaring the types used by a transition system or type annotations λ-expression.

2.3.4.1 Non-unions

An advantage of basic, function and product domains is that they can be aliased directly or used anonymously. However, to ensure that they are well-formed, circular definitions are disallowed.

Example 2.3 (Non-union domain definitions)
The following domains are well-formed:

$$\begin{aligned} &\mathbf{domain}\ d_1 := \mathbf{sym} \vec{\to} (d_2 \bar{\times} d_3) \\ &\mathbf{domain}\ d_2 := \mathbf{int} \\ &\mathbf{domain}\ d_3 := d_2 \vec{\to} d_2 \end{aligned}$$

If all domain variables are substituted, the complete definition of d_1 is:

$$d_1 = \mathbf{sym} \vec{\to} (\mathbf{int} \bar{\times} (\mathbf{int} \vec{\to} \mathbf{int}))$$

However, consider the following definitions, covering different cases of ill-formed domains:

$$\begin{aligned} &\mathbf{domain}\ d_1 := \mathbf{sym} \vec{\to} d_1 \\ &\mathbf{domain}\ d_2 := \mathbf{int} \bar{\times} d_3 \\ &\mathbf{domain}\ d_3 := d_4 \bar{\times} d_4 \\ &\mathbf{domain}\ d_4 := d_2 \\ &\mathbf{domain}\ d_5 := d_5 \end{aligned}$$

These cases represent various forms of circular referencing, e. g. d_5 is a case of self-aliasing, d_1 is a circular self-reference, while $d_{2\text{ to }4}$ represent a more general case of circularity.

To prevent this, a directed graph[3] is built from all the named non-union definitions. The left-hand side of a domain definition is the parent, and whatever domain variables appear on the right-hand side are its children. If the graph is acyclic, then no circular references have occurred.

2.3.4.2 Unions

Well-formedness of union domains requires different considerations. They can have circular references, allowing both mutual and self-recursion. On the other hand, the following restrictions are imposed:

- They cannot be used anonymously, i. e. they can only be declared at the root of a domain definition.
- All tags across all union domains in a specification must be pair-wise distinct, so as to allow inferring the domain of an expression by its tag (section 2.4.3.3).
- They cannot be aliased directly.
- They must be well-ordered.

Example 2.4 (Union definitions)
Consider the following union definitions:

$$\textsf{domain } d_1 := \bar{\bigcup} (t_1 \textsf{ as int}, \ t_2 \textsf{ as } d_2)$$
$$\textsf{domain } d_2 := \bar{\bigcup} (t_3 \textsf{ as } d_1 \vec{\to} d_1)$$

These union domains are well-formed since all tags are pair-wise distinct, d_1 is well-ordered through t_1, and d_2 is well-ordered through its references to d_1. The circular reference between d_1 and d_2 poses no problem. Hence, all requirements for well-formedness are satisfied.

For examples of ill-formed unions, consider the following definitions:

$$\textsf{domain } d_1 := \bar{\bigcup} (t_1 \textsf{ as } d_1)$$
$$\textsf{domain } d_2 := \bar{\bigcup} (t_2 \textsf{ as int}, \ t_2 \textsf{ as } d_2, t_1 \textsf{ as } \epsilon)$$
$$\textsf{domain } d_3 := \textsf{int} \vec{\to} \bar{\bigcup} (t_3 \textsf{ as bool})$$
$$\textsf{domain } d_4 := d_2$$

[3] For a formal representation, refer to A.1

> d_1 is not well-ordered, lacking a tag pointing to a well-ordered domain. d_2 is well-ordered, but it uses tag t_2 twice, as well as t_1, already used by d_1. The union component of d_3 is well-ordered and uses distinct tags, but is anonymous, as opposed to being the root of the definition. d_4 violates the constraint regarding direct aliasing of unions, giving a second name to d_2.

Similarly to well-formedness of non-unions, a dependency graph is built[4] from all the unions in a specification, constructing a node for every union constructor and tag. Each tag is the child of the union it belongs to, and unions are children of a tag node, if they are found in the domain the latter encapsulates. Tags encapsulating a well-formed domain with no unions are leaves. If there is at least one path from every union node to a leaf, then all unions are well-ordered.

2.3.5 Domain equivalence

Considering the notions on well-formedness, domain equivalence can be defined, considering it is a crucial component of the type systems defined later. Denoted as $\equiv$, it is defined for well-formed elements in $\mathbf{Dom} \setminus \mathbf{Dom}_{\bar{\cup}}$, given Δ. It checks for structural equivalence on non-unions, and name equivalence for unions.

Domain equivalence is defined firstly by the following axioms, describing equivalence for basic domains and unions:

$$\text{[CONST]:}\quad \Delta \vdash \mathtt{B}_1 \equiv \mathtt{B}_2 \quad \text{if } \mathtt{B}_1 = \mathtt{B}_2$$

$$\text{[UNION]:}\quad \Delta \vdash \mathtt{x}_1 \equiv \mathtt{x}_2 \quad \begin{array}{l}\text{if } \mathtt{x}_1 = \mathtt{x}_2, \text{ and} \\ \Delta(\mathtt{x}_1), \Delta(\mathtt{x}_2) \in \mathbf{Dom}_{\bar{\cup}}\end{array}$$

Basic domains are equivalent if they are the same construct, while unions rely on the aforementioned name equivalence.

Product and function types are equivalent if both of their constituents are respectively equivalent.

$$\text{[PRODUCT]:}\quad \frac{\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}'_1 \qquad \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}'_2}{\Delta \vdash \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \equiv \mathtt{D}'_1 \bar{\times} \mathtt{D}'_2}$$

$$\text{[FUNCTION]:}\quad \frac{\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}'_1 \qquad \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}'_2}{\Delta \vdash \mathtt{D}_1 \bar{\to} \mathtt{D}_2 \equiv \mathtt{D}'_1 \bar{\to} \mathtt{D}'_2}$$

[4] For a formal representation, refer to appendix A.2

Variables pointing to non-union domains are replaced with the values they are bound to in the environment. The equivalence is then examined between the unchanged domain and the resulting values.

$$\text{[VAR-1]:}\quad \frac{\Delta \vdash \mathtt{D} \equiv \Delta(\mathtt{x})}{\Delta \vdash \mathtt{D} \equiv \mathtt{x}} \quad \text{if } \Delta(\mathtt{x}) \notin \mathbf{Dom}_{\bar{\cup}}$$

$$\text{[VAR-2]:}\quad \frac{\Delta \vdash \Delta(\mathtt{x}) \equiv \mathtt{D}}{\Delta \vdash \mathtt{x} \equiv \mathtt{D}} \quad \text{if } \Delta(\mathtt{x}) \notin \mathbf{Dom}_{\bar{\cup}}$$

Theorem 2.3.1 (Domain equivalence is an equivalence relation) *Assuming that a domain environment is well-formed with respect to all of its defined domain variables, the domain equivalence relation is a proper equivalence, i. e. it is reflexive, transitive and symmetric.*[5]

Whenever the same metavariable for domains is used in a type rule, the implication is that both occurrences are expected to be equivalent, for the rule to apply, when performing type checking on a concrete specification.

2.4 Expressions

Expressions are given as an extended variation of λ-calculus. Their purpose is allowing the description of complex, reusable computational behaviors when defining language specifications. The following sections cover extensions to the typical λ-calculus formula introduced in the metalanguage[6][7].

2.4.1 Abstract syntax

The main syntactic category is **Exp** describing the structures of expressions, while **Const** describes various constants. Given $\mathtt{e} \in \mathbf{Exp}$ and $\mathtt{c} \in \mathbf{Const}$, the formation

[5] For formal proof, refer to appendix C.1.
[6] For the extensive semantics, refer to appendix B.1
[7] For the extensive type system, refer to appendix B.2

rules, excluding those describing infix binary operations, are as follows:

$$
\begin{aligned}
\mathtt{e} \ ::= \ & \mathtt{x} \mid \mathtt{c} \mid \epsilon \mid \mathbf{!}\ \mathtt{e} \mid \mathtt{-e} \\
& \mid \mathbf{lam}\ \mathtt{x} : \mathtt{D}\ .\ \mathtt{e} \mid \mathbf{app}\ \mathtt{e}_1\ \mathtt{e}_2 \\
& \mid \mathbf{let}\ \mathtt{x} := \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2 \\
& \mid \mathbf{letrec}\ \mathtt{x}_1 : \mathtt{D} := \mathtt{x}_2\ .\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2 \\
& \mid \mathbf{if}\ \mathtt{e}_1\ \mathbf{then}\ \mathtt{e}_2\ \mathbf{else}\ \mathtt{e}_3 \\
& \mid (\mathtt{e}_1,\ \mathtt{e}_2) \mid \mathbf{head}\ \mathtt{e} \mid \mathbf{tail}\ \mathtt{e} \\
& \mid [\ \mathtt{e}_1\ \mathbf{to}\ \mathtt{e}_2\ ]\ \mathtt{e}_3 \\
& \mid \mathtt{e} \gg \mathtt{t} \mid \mathtt{t}\ [\ \mathtt{e}\ ] \mid \mathtt{e}\ \mathbf{is}\ \mathtt{t} \\
\mathtt{c} \ ::= \ & \mathbf{true} \mid \mathbf{false} \mid \bot_{\mathtt{D}} \mid \mathtt{n} \mid \mathtt{s} \mid \mathtt{y}
\end{aligned}
$$

The term constructors denoting binary operators in **Exp** are given by the following sets:

$$
\begin{aligned}
\sigma_A &= \{+,\ \mathbf{-},\ \mathbf{*},\ /,\ \mathbf{\%}\} \\
\sigma_R &= \{=,\ \mathbf{!=},\ <,\ >,\ <=,\ >=\} \\
\sigma_B &= \{\mathbf{\&},\ |\} \\
\sigma_S &= \{++\} \\
\sigma &= \bigcup_{X \in \{A,\ R,\ B,\ S\}} \sigma_X
\end{aligned}
$$

where σ_A, σ_R, σ_B and σ_S represent arithmetic, relational, boolean and string operators, respectively. The formation rules of **Exp** are extended as follows:

$$\mathtt{e} ::= \ldots \mid \mathtt{e}_1\ \mathbf{op}\ \mathtt{e}_2,\ \forall\ \mathbf{op} \in \sigma$$

2.4.2 Final configurations

Expressions evaluate to members of the set Ω, which encompasses all possible values resulting from evaluating expressions. Given $\omega \in \Omega$, its formation rules are:

$$
\begin{aligned}
\omega ::= \ & \bar{n} \mid \bar{s} \mid \bar{x} \\
& \mid \mathit{tt} \mid \mathit{ff} \\
& \mid (\omega_1, \omega_2) \\
& \mid \mathtt{t}[\omega] \mid \mathtt{t} \\
& \mid \langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle \\
& \mid \langle\!\langle e, \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e} \rangle\!\rangle
\end{aligned}
$$

Each formation rule corresponds to a value in one of the subsets of Ω:

- $\bar{n} \in \mathbb{Z}$ represents the values of integers.
- $\bar{s} \in \Sigma^*$ represents string values, obtained by applying the Kleene star to alphabet Σ.
- $\bar{x} \in \chi^*$ denotes symbol values similarly to strings, but using alphabet χ instead.
- *tt* and *ff* are the boolean constants "true" and "false", respectively.
- (ω_1, ω_2) represents a pair of values. Pairs are obtained from evaluating expressions belonging to a domain formed by $\bar{\times}$.
- `t` belongs to **Tag**. The construct $\mathtt{t}[\omega]$ creates a pair between `t` and ω, where `t` encapsulates ω. Alternatively, tags can be given as "bare", hence `t` with no ω attached.
- The construct $\langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle$ represents a closure between environment e and the result of evaluating a λ-expression in the syntax of specifications. e is a partial function, mapping variables to final configurations, with signature:

$$e : \textbf{Var} \rightharpoonup \Omega$$

- Conversely, to accommodate recursive definitions, $\langle\!\langle e, \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e} \rangle\!\rangle$ represents a recursive closure, used wherever `x` is meant to be accessed within its own body, represented by $\lambda \mathtt{x}'.\mathtt{e}$.

All operations normally defined over the various basic subsets of Ω, such as $\mathbb{Z}$, have their domains implicitly extended to Ω, and are partially defined over their original range, so as to allow using the original operators within the metalanguage specification unimpeded. If the operands involved are not part of the original domain of the operation, the result is considered undefined.[8]

Example 2.5 (Extending the sum operation of integers)
The sum of integers, normally defined as $(\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$, has signature $(\Omega \times \Omega) \rightharpoonup \mathbb{Z}$ instead.
For some ω_1 and ω_2, $\omega_1 + \omega_2$ is defined only if (ω_1, ω_2) is of the form $(\bar{n}_1, \bar{n}_2)$, hence in $\mathbb{Z} \times \mathbb{Z}$.
Take arbitrary expression $4 + 5$, defined over pairs of integers, evaluating to 9, since $(4, 5) \in \mathbb{Z} \times \mathbb{Z}$.
By contrast, $tt + 5$ is undefined, since $(tt, 5) \notin \mathbb{Z} \times \mathbb{Z}$.

[8] These undefined values would consequently result in a run-time error. In practice, the type system will disallow improper applications (appendix B.2), so the metalanguage specification is not affected due to these extensions.

2.4.3 Semantics

In this section, we cover the semantics of expressions relevant to the critical features of the metalanguage. The signature of transitions over expressions is defined as:

$$\textbf{\textit{Exp}} : Env \vdash \textbf{Exp} \Downarrow \Omega$$

The domain Env is defined as:

$$Env = \textbf{Var} \rightharpoonup \Omega$$

2.4.3.1 Variables and constants

Elements in **Const** are mapped to final configurations through function $\mathscr{V}$ with the signature:

$$\mathscr{V} : \textbf{Const} \rightharpoonup \Omega$$

To achieve this, several semantic functions interpreting constant literals are used. These are:

$$[\![\_]\!]_{\mathscr{N}} : \textbf{Int} \to \mathbb{Z}$$
$$[\![\_]\!]_{\mathscr{S}} : \textbf{String} \to \Sigma^*$$
$$[\![\_]\!]_{\mathscr{X}} : \textbf{Symbol} \to \chi^*$$

Given a literal in the syntactic category in their domain, they will return the associated value in the range.
Using these, $\mathscr{V}$ can be defined as:

$$\mathscr{V}(\mathsf{c}) = \begin{cases} [\![\mathsf{c}]\!]_{\mathscr{N}}, \text{ if } \mathsf{c} \in \textbf{Int} \\ [\![\mathsf{c}]\!]_{\mathscr{S}}, \text{ if } \mathsf{c} \in \textbf{String} \\ [\![\mathsf{c}]\!]_{\mathscr{X}}, \text{ if } \mathsf{c} \in \textbf{Symbol} \\ \textit{tt}, \text{ if } \mathsf{c} = \textbf{true} \\ \textit{ff}, \text{ if } \mathsf{c} = \textbf{false} \end{cases}$$

The axioms of *Exp* involving constants and variables are:

$$\text{[VAR]:} \quad e \vdash \mathsf{x} \Downarrow e(\mathsf{x}) \quad \text{if } e(\mathsf{x}) \text{ is defined}$$

$$\text{[CONST]:} \quad e \vdash \mathsf{c} \Downarrow \mathscr{V}(\mathsf{c}) \quad \text{if } \mathscr{V}(\mathsf{c}) \text{ is defined}$$

meaning that, when evaluating a variable, x, its resulting value should correspond to the value mapped in e. For constant literals, $\mathscr{V}$ is applied to the constant and returns the corresponding value.

The bottom element, $\bot_{\mathsf{D}}$, is used to denote an undefined value and is meant to result in a run-time error if encountered by the axiom for constants during evaluation. Its purpose is in declaring partial functions.

Example 2.6 (An empty partial function declaration)
Consider a state s representing a mapping of variables to integers, where its signature would be:

$$s : \chi^* \rightharpoonup \mathbb{Z}$$

Initially, the state should be "empty" (all variable mappings are undefined). To represent this scenario, s can be expressed as:

$$\textsf{lam x : sym . } \bot_{\textsf{int}}$$

Any subsequent binding updates (section 2.4.3.2) keep the initial body of the expression for unbound values, resulting in a run-time error when applying s to them. The type annotation on $\bot$ is used in the type inference phase (section 2.4.4).

2.4.3.2 Functional elements

Many of the constructs representing functional aspects are shared with the language **Flan**[12], with some additions. This section covers an additional operation consisting of artificial enforcement of mappings for a λ-expression.

The metalanguage allows users to enforce the mapping from one expression to another in a function. A common use case for this feature would be denoting updates in states or environments when expressing transition rules. Mathematically, a binding update $[y \mapsto z]f$ is defined as:

$$([y \mapsto z]f)(x) = \begin{cases} f(x) & \text{if } x \neq y \\ z & \text{if } x = y \end{cases}$$

This is reflected in the semantics of binding updates. The value of the updated expression must be a new closure, with a function body changed accordingly.

This language construct requires an internal extension of the abstract syntax of expressions, i. e. this abstract syntax construct does not result from building an AST from a parse tree, but is rather used for run-time transformations. The extension consists of a closure, pairing an environment with an expression, and is expressed by the formation rule:

$$\mathsf{e} ::= \{e, \mathsf{e}\}$$

The rule for evaluating closures is given as:

$$\text{[CLOSURE]:} \quad \frac{e' \vdash \mathtt{e} \Downarrow \omega}{e \vdash \{e', \mathtt{e}\} \Downarrow \omega}$$

Considering these, the rules covering updates for non-recursive and recursive closures, respectively, are:

$$\text{[UPDATE-1]:} \quad \frac{e \vdash \mathtt{e}_3 \Downarrow \langle\!\langle e', \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle}{e \vdash [\ \mathtt{e}_1\ \textsf{to}\ \mathtt{e}_2\ ]\ \mathtt{e}_3 \Downarrow \langle\!\langle e', \lambda \mathtt{x}.\textsf{if}\ \mathtt{x} = \{e, \mathtt{e}_1\}\ \textsf{then}\ \{e, \mathtt{e}_2\}\ \textsf{else}\ \mathtt{e} \rangle\!\rangle}$$

$$\text{[UPDATE-2]:} \quad \frac{e \vdash \mathtt{e}_3 \Downarrow \langle\!\langle e', \mathtt{x}', \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle}{e \vdash [\ \mathtt{e}_1\ \textsf{to}\ \mathtt{e}_2\ ]\ \mathtt{e}_3 \Downarrow \langle\!\langle e', \mathtt{x}', \lambda \mathtt{x}.\textsf{if}\ \mathtt{x} = \{e, \mathtt{e}_1\}\ \textsf{then}\ \{e, \mathtt{e}_2\}\ \textsf{else}\ \mathtt{e} \rangle\!\rangle}$$

2.4.3.3 Operations involving tags

Tag injection involves creating a union value from an expression, by encapsulating it in the given tag. Alternatively, when a tag is "bare" it merely results in evaluating to the value of the tag itself. The operational semantics is:

$$\text{[INJECT-1]:} \quad e \vdash \mathtt{t}\ [\ \epsilon\ ] \Downarrow \mathtt{t}$$

$$\text{[INJECT-2]:} \quad \frac{e \vdash \mathtt{e} \Downarrow \omega}{e \vdash \mathtt{t}\ [\ \mathtt{e}\ ] \Downarrow \mathtt{t}[\omega]}$$

Tag projection is meant to extract the value of an expression encapsulated under a given tag. It is noteworthy that "bare" tags cannot be projected on. The rule for projection on tags is:

$$\text{[PROJECT]:} \quad \frac{e \vdash \mathtt{e} \Downarrow \mathtt{t}[\omega]}{e \vdash \mathtt{e} \gg \mathtt{t} \Downarrow \omega}$$

Another useful feature is checking whether a given expression is an encapsulation through a specific tag. The rules handling encapsulated expressions and "bare" tags, respectively, are:

$$\text{[IS-TAG-1]:} \quad \frac{e \vdash \mathtt{e} \Downarrow \mathtt{t}'}{e \vdash \mathtt{e}\ \textsf{is}\ \mathtt{t} \Downarrow \mathtt{t} = \mathtt{t}'}$$

$$\text{[IS-TAG-2]:} \quad \frac{e \vdash \mathtt{e} \Downarrow \mathtt{t}'[\omega]}{e \vdash \mathtt{e}\ \textsf{is}\ \mathtt{t} \Downarrow \mathtt{t} = \mathtt{t}'}$$

2.4.4 Type system

Considering its rudimentary type declarations, the metalanguage employs a type system to statically prevent errors involving ill-typed expressions[9]. The system ***TExp*** involves notions of bidirectional typing, inferring the types of expressions for the most part, but still requiring type annotations for some constructs, such as recursive declarations.

It has the following signature:

$$\textbf{\textit{TExp}} : Env_T \overset{\Delta}{\vdash} \textsf{Exp} :: \textsf{Dom}$$

evaluating the types of members of **Exp**. Env_T is the signature of type environments, binding variables in specification expressions to their types, and is defined as:

$$Env_T = \textsf{Var} \rightharpoonup \textsf{Dom}$$

2.4.4.1 Variables and constants

To determine the type rules of the axiom for constants, function $\mathscr{T}$ is used, described by the following signature:

$$\mathscr{T} : \textsf{Const} \to \textsf{Dom}$$

and defined as:

$$\mathscr{T}(\mathsf{c}) = \begin{cases} \textsf{int}, \text{ if } \mathsf{c} \in \textsf{Int} \\ \textsf{str}, \text{ if } \mathsf{c} \in \textsf{String} \\ \textsf{sym}, \text{ if } \mathsf{c} \in \textsf{Symbol} \\ \textsf{bool}, \text{ if } \mathsf{c} \in \{\textsf{true}, \textsf{false}\} \\ \mathtt{D}, \text{ if } \mathsf{c} = \bot_{\mathtt{D}} \end{cases}$$

Whenever a variable is encountered, its type is the value that it is bound to in the type environment. For constants, the domain describing their type is retrieved by using $\mathscr{T}$. The respective axioms are defined as:

$$\text{[VAR]:} \quad E \overset{\Delta}{\vdash} \mathsf{x} :: E(\mathsf{x}) \quad \text{if } E(\mathsf{x}) \text{ is defined}$$

$$\text{[CONST]:} \quad E \overset{\Delta}{\vdash} \mathsf{c} :: \mathscr{T}(\mathsf{c})$$

[9] For an extensive proof for type system correctness, refer to appendix C.2.

2.4.4.2 Binding updates

The first prerequisite of a binding update is that the expression being updated must be a function domain. In addition, the domains of the actual and formal parameter must be matching, non-empty, basic domains, and the expression used as the new value in the codomain must have a matching type with the return type. The type rule is given as:

$$\text{[UPDATE]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{B} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_3 :: \mathtt{B} \to \mathtt{D}}{E \overset{\Delta}{\vdash} \mathtt{[}\ \mathtt{e}_1\ \mathbf{to}\ \mathtt{e}_2\ \mathtt{]}\ \mathtt{e}_3 :: \mathtt{B} \to \mathtt{D}} \quad \text{if } \mathtt{B} \neq \epsilon$$

2.4.4.3 Operations on tags

For tag injection, there are two cases:

- The injected expression is empty, resulting in a "bare" tag, and requiring no antecedent type judgements.
- Conversely, if the injected expression is not empty, the type of that expression must be determined, and match the encapsulated domain of that specific tag.

Regardless of case, the result is a variable that is bound to a union domain in domain environment Δ, if said union domain also contains the tag in question, and the domain attached to that tag is equivalent to the domain of the injected expression.

$$\text{[INJECT-1]:}\quad E \overset{\Delta}{\vdash} \mathtt{t}\ \mathtt{[}\ \epsilon\ \mathtt{]} :: \mathtt{x} \quad \text{if } \exists_{\mathtt{t}}(\mathtt{t}, \epsilon, \Delta(\mathtt{x}))$$

$$\text{[INJECT-2]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}}{E \overset{\Delta}{\vdash} \mathtt{t}\ \mathtt{[}\ \mathtt{e}\ \mathtt{]} :: \mathtt{x}} \quad \text{if } \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}))$$

The type rule of tag projection has several requirements:

- The type of the expression yields a variable bound to a union domain in the domain environment.
- The tag projected on is a member of this union.
- The domain encapsulated by this tag is not empty.

If all are satisfied, the result is the encapsulated domain. It is important to note, though, that the type system will not prevent run-time errors where there is a projection on a value belonging to the same union, but under a different tag. Formally, the type rule is:

$$\text{[PROJECT]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x}}{E \overset{\Delta}{\vdash} \mathtt{e} \gg \mathtt{t} :: \mathtt{D}} \quad \text{if } \exists\, \mathtt{D}, \text{ s.t. } \mathtt{D} \neq \epsilon \wedge \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}))$$

For checking whether an expression is a specific tag encapsulation, the requirements are that its type is a variable bound to a union domain, and that the tag is a member of that union. The resulting type is boolean.

$$\text{[IS-TAG]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x}}{E \overset{\Delta}{\vdash} \mathtt{e}\ \textbf{is}\ \mathtt{t} :: \textbf{bool}} \quad \begin{array}{l}\text{if } \Delta(\mathtt{x}) \in \mathbf{Dom}_{\bigcup} \\ \text{and } \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x})) \text{ for some } \mathtt{D}\end{array}$$

2.5 Expressing semantics

One necessary characteristic for the metalanguage is its capacity to express semantics. This entire section essentially boils down to giving a structural operational semantics of structural operational semantics. Therefore, it focuses on the semantics of language constructs that denote transition systems and their rules and axioms.

2.5.1 Configurations

Considering that operational semantics involves transitions between program configurations, it is necessary to be able to express configurations in the metalanguage. In this case, configurations are given in two ways.

- The first is by evaluating expressions and constructing configurations from the resulting values, meaning these configurations share the syntax and semantics with expressions, and are represented by **Exp**. This is encountered in the final configurations of conclusions, and the initial configurations of premises or side-conditions.
- The second involves specifying a configuration pattern. In conclusions, it is used to pattern match with initial configurations and the binding environment (if any), deciding whether to pursue a rule further, and in premises, for destructuring (i. e. "unpacking") final configurations, and determining whether some constants are equal to the evaluated results.

2.5.1.1 Abstract Syntax

The abstract syntax of configurations is given by $\mathtt{C} \in \mathbf{Config}$, and its formation rules are:

$$\mathtt{C} ::= \mathtt{x} \mid \mathtt{c} \mid (\mathtt{C}_1, \mathtt{C}_2) \mid \mathtt{t}\ [\ \mathtt{C}\ ] \mid \epsilon$$

These denote various patterns that can be encountered in configurations. Metavariables, constants and "bare" tags are denoted as simple patterns, whereas pairs and tags are composite, and involve recursion in determining their constituents.

2.5.1.2 Semantics

The semantics of configurations is given by a transition system with the following signature:

$$\textit{Config} : \mathit{Env} \vdash \mathbf{Config} \times \Omega \Downarrow \mathit{Env}$$

Evaluating a configuration is essentially pattern matching, and ensures that the structure of the supplied value correctly corresponds to the expected configuration.

The axioms of ***Config*** describe the simple patterns. Variables produce an environment where they are bound to their values, whereas constants and "bare" tags leave the environment unchanged, if they are equivalent to the supplied value.

$$\text{[VAR]:} \quad e \vdash \langle \mathtt{x},\ \omega \rangle \Downarrow [\mathtt{x} \mapsto \omega]e$$

$$\text{[CONST]:} \quad e \vdash \langle \mathtt{c},\ \omega \rangle \Downarrow e \quad \text{if } \mathscr{V}(\mathtt{c}) = \omega$$

$$\text{[TAG-1]:} \quad e \vdash \langle \mathtt{t}_1\ [\ \epsilon\ ], \mathtt{t}_2 \rangle \Downarrow e \quad \text{if } \mathtt{t}_1 = \mathtt{t}_2$$

The rules of ***Config*** specify the semantics upon encountering composite patterns. If the pattern consists of a pair, the provided value of the configuration should also be a pair, and respective pairings between pattern and value members are recursively evaluated. Conversely, tag patterns are matched if the tag values are equal, and their respective projections are recursively evaluated successfully. The rules are:

$$\text{[PAIR]:} \quad \frac{e \vdash \langle \mathtt{C}_1,\ \omega_1 \rangle \Downarrow e'' \qquad e'' \vdash \langle \mathtt{C}_2,\ \omega_2 \rangle \Downarrow e'}{e \vdash \langle (\mathtt{C}_1, \mathtt{C}_2),\ (\omega_1,\ \omega_2) \rangle \Downarrow e'}$$

$$\text{[TAG-2]:} \quad \frac{e \vdash \langle \mathtt{C},\ \omega \rangle \Downarrow e'}{e \vdash \langle \mathtt{t}_1\ [\ \mathtt{C}\ ],\ \mathtt{t}_2[\omega] \rangle \Downarrow e'} \quad \text{if } \mathtt{t}_1 = \mathtt{t}_2$$

Example 2.7 (Pattern matching a configuration)
Consider in the specification of **Bims** (appendix D.1) the rule `MUL`. If there is some initial environment e, and the value that is being pattern matched is $mul[(3,4)]$, the derivation tree when evaluating the initial configuration in the conclusion of `MUL` would be as follows:

$$\dfrac{\dfrac{e \vdash \langle a_1,\ 3\rangle \Downarrow [a_1 \mapsto 3]e \qquad [a_1 \mapsto 3]e \vdash \langle a_2,\ 4\rangle \Downarrow [a_1 \mapsto 3,\ a_2 \mapsto 4]e}{e \vdash \langle \mathbf{(}a_1,\ a_2\mathbf{)},\ (3,4)\rangle \Downarrow [a_1 \mapsto 3,\ a_2 \mapsto 4]e}}{e \vdash \langle mul\mathbf{[(}a_1,\ a_2\mathbf{)]},\ mul[(3,4)]\rangle \Downarrow [a_1 \mapsto 3,\ a_2 \mapsto 4]e}$$

The pattern matching is successful, since at every step, the structure of the value matches the configuration pattern: the tags match through rule TAG-2, and the encapsulated value is a pair, according to rule PAIR. Following the axiom VAR, the result is an environment updated with respect to metavariables a_1 and a_2, bound to values 3 and 4, respectively.

2.5.2 Transition systems

2.5.2.1 Abstract syntax

Transition systems are formed from the composition of three syntactic categories:

- $\mathtt{T} \in$ **TSystem** - represents an entire transition system. A transition system is represented by a collection of rules, used when evaluating the transitions. Its formation rules are:

$$\mathtt{T} ::= \overrightarrow{\mathtt{R}}$$

- $\mathtt{R} \in$ **Rule** - forms individual rules. Rules are composed of a conclusion and premises. If premises are absent, the rule is instead an axiom. The formation rules of r are:

$$\mathtt{R} ::= \mathbf{[[}\ \mathtt{x}\ \mathbf{]]}\mathtt{:}\ \mathtt{C}_1\ \mathtt{|-}\ \mathtt{C}_2\ \mathtt{=>}\ \mathtt{e} \setminus \vec{\mathtt{P}}$$

For brevity, formation rules of rules in transition systems without a binding model are excluded. In addition, to allow easier expression of composite models (e. g. the environment-store model), binding environments are also expressed through the configuration syntax and semantics.

- $\mathtt{P} \in$ **Premise** - forms the premises of rules. Premises can be of several types: transitions (internal or referring to another transition system), side conditions, or local declarations. The formation rules are:

$$\begin{aligned}\mathtt{P} ::= & \ \textbf{if}\ \mathtt{e} \\ | & \ \textbf{let}\ \mathtt{x} := \mathtt{e} \\ | & \ \textbf{letrec}\ \mathtt{x}_1 : \mathtt{D} := \mathtt{x}_2\ .\ \mathtt{e} \\ | & \ \mathtt{e}_1\ \mathtt{|\text{-}}\ \mathtt{e}_2 \Rightarrow \mathtt{C} \\ | & \ \mathtt{e}_1\ \mathtt{|\text{-}}\ \mathtt{e}_2 =\mathtt{x}^{\mathtt{T}}\!> \mathtt{C}\end{aligned}$$

2.5.2.2 Semantics

Expressing the complete semantics of transition systems requires three transition systems:

- ***TSystem*** performs syntax-directed evaluations on a transition system definition. It takes two environments: one holding the values of variables, the other binding variables to other transition systems. Its final configurations are in Ω, and its initial configurations are a 3-tuple between a member of **TSystem**, a value meant to be evaluated, representing the initial configuration in the source language, and a value corresponding to the binding model (if any is used by the transition system). Its signature is:

$$\textit{TSystem} : \mathit{Env} \times \mathit{Env}^{\Downarrow} \vdash \textbf{TSystem} \times \Omega \times \Omega \Downarrow \Omega$$

  The domain $\mathit{Env}^{\Downarrow}$ is the transition system environment, and is defined as:

$$\mathit{Env}^{\Downarrow} = \textbf{Var} \rightharpoonup \textbf{TSystem}$$

  Since variables denoting transition systems can be used only in specific contexts, introducing this additional environment means that variable names can be shared between transition systems and other entities. One notable example where this segregation is advantageous is a transition system sharing the name with the abstract syntax formation rules for which it describes the semantics.

- ***Rule*** checks whether an individual rule can be applied successfully. It has the following signature:

$$\textit{Rule} : \mathit{Env} \times \mathit{Env}^{\Downarrow} \vdash \textbf{Rule} \times \Omega \times \Omega \Downarrow \Omega$$

- ***Premise*** is the transition system according to which the evaluation of premises in rules is carried out. Its binding model also consists of an environment

and transition system environment, its initial configurations are members of **Premise**, and it evaluates to an environment. The signature is:

$$\textbf{\textit{Premise}} : Env \times Env^{\Downarrow} \vdash \textbf{Premise} \Downarrow Env$$

TSystem has one axiom. Whenever an abstract syntax tree is supposed to be evaluated by a transition system, the rules comprising the system are evaluated until one manages to produce a closed derivation tree. Formally, it is expressed as:

$$\text{[TRANSITION-SYSTEM]:}\quad \frac{\exists i : e, e^{\Downarrow} \vdash \langle \mathtt{R}_i,\ \omega,\ \omega_e \rangle \Downarrow_{\textbf{\textit{Rule}}} \omega'}{e, e^{\Downarrow} \vdash \langle \overrightarrow{\mathtt{R}_i}^{n},\ \omega,\ \omega_e \rangle \Downarrow \omega'}$$

The axioms of ***Rule*** are evaluated according to these steps:

1. Destructure the environment pattern, binding all variables to their values in the variable environment of the specification (this step is skipped for transition systems without a binding model).
2. Using the new environment, determine whether the initial configuration pattern matches the supplied value. Store the resulting environment, that will include the bindings to values of all metavariables found in the configuration pattern.
3. Refer to transition system ***Premise*** to determine whether all the premises are satisfied. Each premise evaluation uses the environment resulting from evaluating the previous premise. The first premise uses the environment resulting from matching the initial configuration. The last environment will be used in the next step. If there are no premises, the environment resulting from 2. will be used instead.
4. Using the environment from 3., evaluate the expression representing the final configuration in the conclusion. The result of this expression corresponds to the result of the evaluation.

Formally, the axiom is defined as:

$$\text{[RULE]:}\quad \frac{\begin{array}{c} e \vdash \langle \mathtt{C}_1,\ \omega_e \rangle \Downarrow_{\textbf{\textit{Config}}} e' \qquad e' \vdash \langle \mathtt{C}_2,\ \omega' \rangle \Downarrow_{\textbf{\textit{Config}}} e_0 \\ \forall i : e_{i-1},\ e^{\Downarrow} \vdash \mathtt{P}_i \Downarrow_{\textbf{\textit{Premise}}} e_i \\ e_n \vdash \mathtt{e} \Downarrow_{\textbf{\textit{Exp}}} \omega \end{array}}{e, e^{\Downarrow} \vdash \langle \texttt{[[x]]: } \mathtt{C}_1 \texttt{ |- } \mathtt{C}_2 \texttt{ => e \ } \overrightarrow{\mathtt{P}_i}^{n},\ \omega',\ \omega_e \rangle \Downarrow \omega}$$

Premise involves several axioms, corresponding to each formation rule of `P`. If a premise is a side condition, the expression is evaluated according to ***Exp***, and, if

the result is true, it evaluates to the current variable environment. The axiom is given as:

$$\text{[IF]:}\quad e, e^{\Downarrow} \vdash \textbf{if } \mathtt{e} \Downarrow e \quad \text{if } e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \mathit{tt}$$

A premise can also be a local declaration, in which case the expression is evaluated, and the resulting configuration is the current environment with the variable bound to that value. The axiom for local declaration is given as:

$$\text{[LET]:}\quad e, e^{\Downarrow} \vdash \textbf{let } \mathtt{x} := \mathtt{e} \Downarrow [\mathtt{x} \mapsto \omega]e \quad \text{where } e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \omega$$

Much like expressions, local declarations can also be recursive, in which case the axiom is expressed as:

$$\text{[LETREC]:}\quad e, e^{\Downarrow} \vdash \textbf{letrec } \mathtt{x}_1 : \_ := \mathtt{x}_2 \ . \ \mathtt{e} \Downarrow [\mathtt{x} \mapsto \langle\!\langle e, \mathtt{x}_1, \lambda \mathtt{x}_2.\mathtt{e} \rangle\!\rangle]e$$

The final axioms involve other transitions in the premises. When evaluating a transition premise, the following steps are performed:

1. Evaluate the expression representing the variable environment in the source language (as before, this step is skipped if the binding model is absent in the transition system).
2. Evaluate the expression representing the initial configuration in the premise.
3. Using the values extracted from 1. and 2., construct a configuration, given the current environment, transition system environment and the abstract syntax tree of a transition system. Evaluate this configuration according to the rules of ***TSystem***. Here, there is a difference between internal transitions and those referring to other systems:
   - If the transition refers to rules within the same system, the value bound to ϕ is retrieved from the transition system environment. ϕ is a symbol that the syntax tree of the currently evaluated transition system is bound to. Its purpose is circumventing having to propagate the name of the currently evaluated transition system throughout the evaluations.
   - Alternatively, if the transition refers to another system, its abstract syntax is used, and the transition system environment must be updated at ϕ with this new value.
4. Afterwards, destructure the evaluation result by matching it against the final configuration in the premise. Similarly to the initial configuration in the conclusion, it will evaluate to an environment where all the metavariables are bound to the values in the corresponding positions in the pattern.

The rules are formally defined as:

$$\text{[TRANSITION-1]:}\quad \frac{\begin{array}{c} e \vdash \mathtt{e}_1 \Downarrow_{\textit{Exp}} \omega_e \qquad e \vdash \mathtt{e}_2 \Downarrow_{\textit{Exp}} \omega' \\ e, e^{\Downarrow} \vdash \langle e^{\Downarrow}(\phi),\ \omega',\ \omega_e \rangle \Downarrow_{\textit{TSystem}} \omega \\ e \vdash \langle \mathtt{C},\ \omega \rangle \Downarrow_{\textit{Config}} env' \end{array}}{e, e^{\Downarrow} \vdash \mathtt{e}_1 \ \mathtt{|-}\ \mathtt{e}_2 \ \mathtt{=>}\ \mathtt{C} \Downarrow env'}$$

$$\text{[TRANSITION-2]:}\quad \frac{\begin{array}{c} e \vdash \mathtt{e}_1 \Downarrow_{\textit{Exp}} \omega_{env} \qquad e \vdash \mathtt{e}_2 \Downarrow_{\textit{Exp}} \omega' \\ e, [\phi \mapsto e^{\Downarrow}(\mathtt{x}^{\mathtt{T}})] e^{\Downarrow} \vdash \langle e^{\Downarrow}(\mathtt{x}^{\mathtt{T}}),\ \omega',\ \omega_{env} \rangle \Downarrow_{\textit{TSystem}} \omega \\ e \vdash \langle \mathtt{C},\ \omega \rangle \Downarrow_{\textit{Config}} env' \end{array}}{e, e^{\Downarrow} \vdash \mathtt{e}_1 \ \mathtt{|-}\ \mathtt{e}_2 \ \mathtt{=x^T>}\ \mathtt{C} \Downarrow env'}$$

2.5.3 Type system

2.5.3.1 General notions

Similarly to how the semantics of transition systems uses a separate environment, $Env^{\Downarrow}$, to bind variables to transition systems (section 2.5.2), the corresponding type system also introduces a transition system type environment with domain $Env_T^{\Downarrow}$, defined as:

$$Env_T^{\Downarrow} = \mathbf{Var} \rightharpoonup \mathbf{Dom}^{\Downarrow}$$

where $\mathtt{D}^{\Downarrow} \in \mathbf{Dom}^{\Downarrow}$ is the syntactic category of transition system domains. The formation rules of $\mathbf{Dom}^{\Downarrow}$ are as follows:

$$\mathtt{D}^{\Downarrow} ::= \mathtt{D}_1 \ \mathtt{|-}\ \mathtt{D}_2 \ \mathtt{=>}\ \mathtt{D}_3$$

2.5.3.2 Configurations

Type checking configurations involves a semantics defined over $\mathbf{Config} \times \mathbf{Dom}$ which serves a dual purpose of pattern matching the domain, as well as updating the type environment with respect to all metavariables found within the pattern. Its definition is:

$$\textit{TConfig} : Env_T \overset{\Delta}{\vdash} \mathbf{Config} \times \mathbf{Dom} \Downarrow Env_T$$

mirroring that of *Config* (section 2.5.1), but where configurations are pattern matched against domains, as opposed to values.

For simple patterns, it involves checking that constants in the pattern belong to the given domain and tags belong to the given union, or, alternatively, updating the type environment with respect to any encountered metavariables. The given

axioms are:

$$\text{[VAR]:} \quad E \overset{\Delta}{\vdash} \langle \mathtt{x},\ \mathtt{D} \rangle \Downarrow [\mathtt{x} \mapsto \mathtt{D}]E \quad \text{if } \mathtt{D} \neq \epsilon$$

$$\text{[CONST]:} \quad E \overset{\Delta}{\vdash} \langle \mathtt{c},\ \mathtt{D} \rangle \Downarrow E \quad \text{if } \Delta \vdash \mathtt{D} \equiv \mathscr{T}(\mathtt{c}) \wedge \mathtt{c} \neq \bot_{\mathtt{D}}$$

$$\text{[TAG-1]:} \quad E \overset{\Delta}{\vdash} \langle \mathtt{t}\ \mathtt{[}\ \epsilon\ \mathtt{]},\ \mathtt{x}^{\mathtt{D}} \rangle \Downarrow E \quad \text{if } \exists_{\mathtt{t}}(\mathtt{t}, \epsilon, \Delta(\mathtt{x}^{\mathtt{D}}))$$

Composite patterns pass the type checking phase if they match the top-level structure of the domain, and each of its members are also type correct. If all these conditions are satisfied, then the configuration is properly typed. Formally, the rules are:

$$\text{[PAIR]:} \quad \frac{E \overset{\Delta}{\vdash} \langle \mathtt{C}_1,\ \mathtt{D}_1 \rangle \Downarrow E'' \qquad E'' \overset{\Delta}{\vdash} \langle \mathtt{C}_2,\ \mathtt{D}_2 \rangle \Downarrow E'}{E \overset{\Delta}{\vdash} \langle (\mathtt{C}_1,\ \mathtt{C}_2),\ \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \rangle \Downarrow E'}$$

$$\text{[TAG-2]:} \quad \frac{E \overset{\Delta}{\vdash} \langle \mathtt{C},\ \mathtt{D} \rangle \Downarrow E'}{E \overset{\Delta}{\vdash} \langle \mathtt{t}\ \mathtt{[}\ \mathtt{C}\ \mathtt{]},\ \mathtt{x}^{\mathtt{D}} \rangle \Downarrow E'} \quad \text{if } \exists\ \mathtt{D} \text{ s.t. } \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}^{\mathtt{D}}))$$

2.5.3.3 Transition systems

The type systems are supposed to check whether all constructs involved in a transition system definition are well-typed.

Much like the operational semantics, they are layered from the top- to the bottom-most constructs:

- ***TTSystem*** is the top-most type system, which determines whether a transition system is well-typed. Its signature is:

$$\textbf{\textit{TTSystem}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \textsf{TSystem} ::^{\checkmark}$$

- ***TRule*** checks that rules within a transition system are well-typed. Its signature is defined as:

$$\textbf{\textit{TRule}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \textsf{Rule} ::^{\checkmark}$$

- ***TPremise*** is the system which determines whether premises are well-typed. Since premises may also involve local declarations or transitions that require

destructuring of final configurations, resulting in environment updates, the type checker is expressed similarly to the one for configurations. The signature is:

$$\textbf{\textit{TPremise}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \textsf{Premise} \Downarrow Env_T$$

Similarly to the semantics, the rules expressed here will revolve around transition systems that have a binding model, and the rules for those without one simply omit all operations involving it.

A member of **TSystem** is well-typed if all the rules within the transition system that it represents are well-typed. Formally, the judgement is given as:

$$\text{[TRANSITION-SYSTEM]:} \quad \frac{\forall i : E, E^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{R}_i ::^{\checkmark}_{\textbf{\textit{TRule}}}}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \overrightarrow{\mathtt{R}_i}^{n} ::^{\checkmark}}$$

For individual rules, the steps mirror those of the rule in transition system ***Rule***, in section 2.5.2.2. They are as follows:

1. If a binding environment is present, destructure it as if it were a configuration. This is achieved by binding the types of all metavariables involved, while type-checking against the binding model domain. The latter is extracted from the transition system type environment, in which, again, ϕ denotes the currently type checked transition system.
2. Type check the initial configuration in the conclusion to ensure a matching pattern with the domain of initial configurations.
3. Sequentially ensure type correctness of every premise. The first premise uses the type environment from 2., and each subsequent premise uses the type environment resulting from the previous premise.
4. The final type environment resulting from 3. is used when checking whether final configuration in the conclusion is well-typed. Additionally, the resulting domain must be equivalent to the one for final configurations as given by the transition system type environment.

Formally, the rule is:

$$\text{[RULE]:} \quad \frac{\begin{array}{c} E \vdash \langle \mathtt{C}_1,\ \mathtt{D}_1 \rangle \Downarrow_{\textbf{\textit{TConfig}}} E' \qquad E' \overset{\Delta}{\vdash} \langle \mathtt{C}_2,\ \mathtt{D}_2 \rangle \Downarrow_{\textbf{\textit{TConfig}}} E_0 \\ \forall i : E_{i-1}, E^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{P}_i \Downarrow_{\textbf{\textit{TPremise}}} E_i \qquad E_n \overset{\Delta}{\vdash} \mathtt{e} ::_{\textbf{\textit{TExp}}} \mathtt{D}_3 \end{array}}{\begin{array}{c} E, E^{\Downarrow} \overset{\Delta}{\vdash} \textbf{[[x]]}\mathtt{: C_1 \ |\!-\ C_2 => e \setminus} \overrightarrow{\mathtt{P}_i}^{n} ::^{\checkmark} \\ \text{where } E^{\Downarrow}(\phi) = \mathtt{D_1 \ |\!-\ D_2 => D_3} \end{array}}$$

Type checking of premises involves several rules, one for each formation rule of **Premise**. For side conditions, the premise is well typed if the expression in the condition is boolean. If so, the type environment is returned unchanged.

$$\text{[IF]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} ::_{\mathit{TExp}} \textbf{bool}}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \textbf{if } \mathtt{e} \Downarrow E}$$

If a premise is a non-recursive local declaration, the type of the expression is evaluated through the rules of $TExp$. Then, the type environment is updated with the resulting type.

$$\text{[LET]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} ::_{\mathit{TExp}} \mathtt{D}}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \textbf{let } \mathtt{x} := \mathtt{e} \Downarrow [\mathtt{x} \mapsto \mathtt{D}]E}$$

If the declaration is recursive instead, the declared domain is verified to ensure it is a function domain. With the variable bound to the declared domain in the type environment, the function body is checked whether it yields a domain equivalent to the output of the function domain. The result is a type environment updated at the declared variable with the domain annotation.

$$\text{[LETREC]:}\quad \frac{[\mathtt{x}_1 \mapsto \mathtt{D}_1 \rightarrow \mathtt{D}_2,\ \mathtt{x}_2 \mapsto \mathtt{D}_2]E \overset{\Delta}{\vdash} \mathtt{e} ::_{\mathit{TExp}} \mathtt{D}_2}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \textbf{letrec } \mathtt{x}_1 : \mathtt{D}_1 \rightarrow \mathtt{D}_2 := \mathtt{x}_2 \ . \ \mathtt{e} \Downarrow [\mathtt{x} \mapsto \mathtt{D}_1 \rightarrow \mathtt{D}_2]E}$$

For premises involving transitions, much like those in section 2.5.2.2, there are different rules for internal transitions and for those referring to other systems, with the difference being which domain is used. The rules have three steps:

1. If a binding environment is required ensure domain equivalence between it and the expression used in the premise transition.
2. Likewise, ensure domain equivalence between that of the initial configuration and the one for its expression.
3. Destructure the final configuration and yield an updated type environment, also ensuring that the given pattern matches the final configuration domain. The updated environment serves as the final result of type checking the premise.

$$\text{[TRANSITION-1]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 ::_{\mathit{TExp}} \mathtt{D}_1 \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 ::_{\mathit{TExp}} \mathtt{D}_2 \qquad E \overset{\Delta}{\vdash} \langle \mathtt{C},\ \mathtt{D}_3 \rangle \Downarrow_{\mathit{TConfig}} E'}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{e}_1 \ \mathtt{|-}\ \mathtt{e}_2 \mathtt{=>}\ \mathtt{C} \Downarrow E'}$$
$$\text{if } E^{\Downarrow}(\phi) = \mathtt{D}_1 \ \mathtt{|-}\ \mathtt{D}_2 \mathtt{=>}\ \mathtt{D}_3$$

$$\text{[TRANSITION-2]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 ::_{\mathit{TExp}} \mathtt{D}_1 \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 ::_{\mathit{TExp}} \mathtt{D}_2 \qquad E \overset{\Delta}{\vdash} \langle \mathtt{C},\ \mathtt{D}_3 \rangle \Downarrow_{\mathit{TConfig}} E'}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{e}_1 \ \mathtt{|-}\ \mathtt{e}_2 \mathtt{=x^T>}\ \mathtt{C} \Downarrow E'}$$
$$\text{if } E^{\Downarrow}(\mathtt{x^T}) = \mathtt{D}_1 \ \mathtt{|-}\ \mathtt{D}_2 \mathtt{=>}\ \mathtt{D}_3$$

2.6 Evaluations

Users of the metalanguage can express evaluations by giving expressions evaluating to configurations and binding models as inputs to transition systems. One potential use of this feature is checking ad-hoc whether the configuration evaluates to the expected result. The semantics of evaluations is given by transition system ***Eval***, with the following signature:

$$\textbf{\textit{Eval}} : \mathit{Env} \times \mathit{Env}_{\Downarrow} \vdash \textbf{Eval} \Downarrow \Omega$$

As a supplementary feature, evaluations can also be performed on simple expressions, without requiring a transition system. The transition system is defined by the following rules:

$$\text{[EVAL]:}\quad \frac{e \vdash \mathtt{e}_1 \Downarrow_{\mathit{Exp}} \omega_e \qquad e \vdash \mathtt{e}_2 \Downarrow_{\mathit{Exp}} \omega' \qquad e, [\phi \mapsto ts] e^{\Downarrow} \vdash \langle ts,\ \omega',\ \omega_e \rangle \Downarrow_{\mathit{TSystem}} \omega}{e, e^{\Downarrow} \vdash \textbf{evaluate}\ \mathtt{e}_1 \ \mathtt{|-}\ \mathtt{e}_2 \mathtt{=x^T>} \Downarrow \omega} \quad \text{where } ts = e^{\Downarrow}(\mathtt{x^T})$$

$$\text{[EXP-EVAL]:}\quad e, \_ \vdash \textbf{evaluate}\ \mathtt{e} \Downarrow \omega \quad \text{if } e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \omega$$

The rule EVAL follows these steps:

1. If the transition system specifies a domain for the binding environment, evaluate the first expression, corresponding to the binding environment (if any).
2. Evaluate the second expression, the value of which will represent the initial configuration.
3. Execute the transition system over the initial configuration, pattern matching against rules and recursively checking premises, until the evaluation terminates (or diverges, depending on the situation). If any rule was applied successfully, then it gives the result of the evaluation.

Much like with the transition systems used to express transition systems (section 2.5.2.2), the axiom based on transition systems without a binding model is omitted, but can be easily constructed from the one given by disregarding the evaluation of ω_e.

2.6.1 Type system

The type system of evaluations ensures that there is a match between the domains of expressions supplied to the transition system, and those in its definition. Expression evaluations require that the given expression is well-typed. The type system, denoted as ***TEval***, has the following signature:

$$\textbf{\textit{TEval}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \textsf{Eval} ::^{\checkmark}$$

An evaluation is well-typed so long as the expressions used for the binding environment and initial configuration are typed with domains equivalent to those that the called transition system is defined with. An expression evaluation is well-typed if the expression itself is well-typed. The type rules of evaluations are:

$$\text{[EVAL]:} \quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 ::_{\textit{TExp}} \mathtt{D}_1 \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 ::_{\textit{TExp}} \mathtt{D}_2}{E, E^{\Downarrow} \vdash \textsf{evaluate } \mathtt{e}_1 \textsf{ |- } \mathtt{e}_2 \textsf{ in } \mathtt{x}^{\mathtt{T}} ::^{\checkmark}} \quad \text{where } E^{\Downarrow}(\mathtt{x}^{\mathtt{T}}) = \mathtt{D}_1 \textsf{ |- } \mathtt{D}_2 \Rightarrow \mathtt{D}_3$$

$$\text{[EVAL-EXP]:} \quad E, \_ \vdash \textsf{evaluate } \mathtt{e} ::^{\checkmark} \qquad \text{if } E \vdash \mathtt{e} ::_{\textit{TExp}} \mathtt{D}$$

2.7 Semantics of a specification

All the language constructs introduced so far are bound through the spine of a specification. As previously mentioned, the spine is a list of definitions, optionally followed by evaluations. The semantics of the spine involves the static analysis of a specification.

2.7.1 Domain and syntax definitions

The semantics of domain and syntax definitions involves a pruned abstract syntax tree of the original specification that strictly includes terms using the **domain** and **syntax** constructors. Considering that some domain aliases can be recursive or used before they are defined, an initial pass updates Δ with respect to all domain variables, binding them to their definitions. Two vectors, containing the non-union and union domain variables, are also produced. Abstract syntax definitions are

converted to the equivalent union domains in this stage. The semantics is given by transition system ***BindDom***, with signature:

$$\textbf{\textit{BindDom}} : \overrightarrow{\textsf{Def}} \times \bar{\Delta} \times \overrightarrow{\textsf{Var}} \times \overrightarrow{\textsf{Var}} \Downarrow \bar{\Delta} \times \overrightarrow{\textsf{Var}} \times \overrightarrow{\textsf{Var}}$$

where $\mathtt{v}^{\mathtt{Df}} \in \overrightarrow{\textsf{Def}}$ and $\mathtt{v} \in \overrightarrow{\textsf{Var}}$ are the syntactic categories representing vectors of definitions and variables, respectively. For each, the following formation rules are defined:

$$\mathtt{v}^{\mathtt{Df}} ::= \overrightarrow{\mathtt{Df}} \mid \mathtt{Df}$$
$$\mathtt{v} ::= \overrightarrow{\mathtt{x}} \mid \mathtt{x}$$

The axioms and rules of ***BindDom*** are:

[DOMAIN]: $\langle \textsf{domain } \mathtt{x} := \mathtt{D},\ \Delta,\ \overrightarrow{\mathtt{x}_i}^{n},\ \overrightarrow{\mathtt{x}_j}^{m} \rangle \Downarrow \langle [\mathtt{x} \mapsto \mathtt{D}]\Delta,\ \overrightarrow{\mathtt{x}_i, \mathtt{x}}^{n},\ \overrightarrow{\mathtt{x}_j}^{m} \rangle$
if $\mathtt{D} \notin \textsf{Dom}_{\bar{\cup}}$ and $\Delta(\mathtt{x})$ is undefined

[UNION]: $\langle \textsf{domain } \mathtt{x} := \mathtt{D},\ \Delta,\ \overrightarrow{\mathtt{x}}^{n},\ \overrightarrow{\mathtt{x}_j}^{m} \rangle \Downarrow \langle [\mathtt{x} \mapsto \mathtt{D}]\Delta,\ \overrightarrow{\mathtt{x}_i}^{n},\ \overrightarrow{\mathtt{x}_j, \mathtt{x}}^{m} \rangle$
if $\mathtt{D} \in \textsf{Dom}_{\bar{\cup}}$ and $\Delta(\mathtt{x})$ is undefined

[SYNTAX]: $\langle \textsf{syntax } \mathtt{x} := \mathtt{S},\ \Delta,\ \overrightarrow{\mathtt{x}_i}^{n},\ \overrightarrow{\mathtt{x}_j}^{m} \rangle \Downarrow \langle [\mathtt{x} \mapsto \mathtt{D}]\Delta,\ \overrightarrow{\mathtt{x}_i}^{n},\ \overrightarrow{\mathtt{x}_j, \mathtt{x}}^{m} \rangle$
where $\mathtt{S} \Downarrow_{\textit{Syntax}} \mathtt{D}$ and if $\Delta(\mathtt{x})$ is undefined

[ALL-DEFINITIONS]: $$\frac{\forall i : \langle \mathtt{Df}_i,\ \Delta_{i-1},\ \mathtt{v}_{i-1},\ \mathtt{v}'_{i-1} \rangle \Downarrow \langle \Delta_i,\ \mathtt{v}_i,\ \mathtt{v}'_i \rangle}{\langle \overrightarrow{\mathtt{Df}_i}^{n},\ \Delta_0,\ \mathtt{v}_0,\ \mathtt{v}'_0 \rangle \Downarrow \langle \Delta_n,\ \mathtt{v}_n,\ \mathtt{v}'_n \rangle}$$

The initial domain environment is undefined with respect to any domain variable, and $\mathtt{v}_0$ and $\mathtt{v}'_0$ are both 0-dimensional vectors, $\overrightarrow{0}$. Until the end of the section, n and m will refer to the number of non-union, respectively union variables in the specification.

Following binding, all conditions regarding well-formedness of domains need to be checked. One condition already met at this point is disallowing anonymous union domains, since this can be enforced syntactically.

2.7.1.1 Checking for free domain variables

Another pass is carried out across the complete syntax tree, to ensure that there are no unbound domain variables in the specification. This includes domain and

syntax definitions, as well as domain expressions used for annotating transition system, local declarations, or λ-expression formal parameters. The rules of this error semantics range over all the syntactic categories in a specification, with the only notable one being:

$$\Delta \vdash \mathtt{x}^{\mathtt{D}} \Downarrow_{\mathbf{err}} \quad \text{if } \Delta(\mathtt{x}^{\mathtt{D}}) \text{ is undefined}$$

If this succeeds, the error is propagated upstream to the root of the specification AST, and the specification evaluation stops.

2.7.1.2 Unique tag verification

The next constraint to be verified is pair-wise distinction between tags across all union definitions, through transition system ***UniqueTags*** with the following signature:

$$\textbf{\textit{UniqueTags}} : \bar{\Delta} \vdash \overrightarrow{\mathbf{Var}} \times \mathcal{P}(\mathbf{Tag}) \Downarrow \mathcal{P}(\mathbf{Tag})$$

By iterating over all union domain variables, tags can be aggregated in a set and checked for duplicates. The rule and axioms are as follows:

$$\text{[UNION]:} \quad \Delta \vdash \langle \mathtt{x},\ S \rangle \Downarrow \bigcup_i \{\mathtt{t}_i\} \cup S \qquad \begin{array}{l} \Delta(\mathtt{x}) = \bigcup\ \overrightarrow{\mathtt{t}_i \ \mathbf{as}\ \mathtt{D}_i}^{\,l} \\ \text{if } \forall i : \mathtt{t}_i \notin S \end{array}$$

$$\text{[ALL-UNIONS]:} \quad \frac{\forall j : \Delta \vdash \langle \mathtt{x}_j,\ S_{j-1} \rangle \Downarrow S_j}{\Delta \vdash \langle \overrightarrow{\mathtt{x}_j}^{\,m},\ S_0 \rangle \Downarrow S_n} \quad \text{where } S_0 = \emptyset$$

2.7.1.3 Verifying well-formedness of non-unions

The execution of ***WF*** involves iterating over every variable defining a domain or abstract syntax, passing along the newly constructed parent environment between adjacent evaluations. This ensures well-formedness over the entire specification.

Let p_0 be a member of P (defined in appendix A.1), which is defined as:

$$p_0(\mathtt{x}) = \emptyset, \quad \forall\ \mathtt{x}$$

It represents the initial state of the parents graph, i. e. it assumes there are no references between the declared domains. Considering p_0, the specification is well-formed if the following judgement holds for all:

$$\frac{\forall i : \Delta \vdash \langle \Delta(\mathtt{x}_i),\ p_{i-1},\ \mathtt{x}_i \rangle \Downarrow_{\mathit{WF}} p_i}{\Delta \vdash \overrightarrow{\mathtt{x}_i}^{\,n} \text{ is well-formed}}$$

2.7.1.4 Disallowing union aliases

What follows is inspecting for union domain aliases. This is achieved through the relation @, and function $\Delta_{\perp}$.

Given a domain environment and the vector of non-union variables, relation @ determines if any union aliasing has occurred. Any variable pointing to another variable would have been added to the non-union domain variable vector during the execution of ***BindDom***, even if the "bedrock" would have eventually been a union. If the "bedrock" definition of a supposedly non-union variable was indeed a union, then union aliasing has occurred. This check can occur only after the well-formedness analysis, since any circular aliasing would cause $\Delta_{\perp}$ to loop infinitely. The rule and axiom of @ are as follows:

$$\text{[VAR]:} \quad \Delta \vdash @\mathtt{x} \quad \text{if } \Delta_{\perp}(\Delta, \mathtt{x}) \notin \mathbf{Dom}_{\bigcup}$$

$$\text{[ALL-VARS]:} \quad \frac{\forall i : \Delta \vdash @\mathtt{x}_i}{\Delta \vdash @\overrightarrow{\mathtt{x}_i^{n}}}$$

2.7.1.5 Verifying well-ordering of unions

As the final step, unions are checked for well-ordering by following the rules of $\downarrow$ for every union domain variable in the specification. It is described formally by the following judgement:

$$\frac{\forall j : \Delta \vdash \downarrow \mathtt{x}_j}{\Delta \vdash \overrightarrow{\mathtt{x}_j^{m}} \text{ is well-ordered}}$$

2.7.2 Data definitions and expression correctness

The semantics of data definitions involve iterating over a vector of data definitions to produce an environments of values and types. Data definitions are type checked and evaluated in the order in which they are declared. As such, data definitions can only use variables which were previously declared, or including themselves in the case of recursive definitions.

For aggregating types and type checking, the transition system is ***BindDataT***, with the following signature:

$$\boldsymbol{BindDataT} : \bar{\Delta} \vdash \overrightarrow{\mathbf{Def}} \times Env_T \Downarrow Env_T$$

The type safety property for expressions assures that no type errors would occur

during evaluation. The rules of ***BindDataT*** are:

$$\text{[DATA]:}\quad \frac{E \overset{\Delta}{\vdash} \texttt{e} ::_{\textit{TExp}} \texttt{D}}{\Delta \vdash \langle \textsf{let}\ \texttt{x} := \texttt{e},\ E\rangle \Downarrow [\texttt{x} \mapsto \texttt{D}]E}$$

$$\text{[DATAREC]:}\quad \frac{[\texttt{x}_1 \mapsto \texttt{D}_1 \vec{\to} \texttt{D}_2,\ \texttt{x}_2 \mapsto \texttt{D}_1]E \overset{\Delta}{\vdash} \texttt{e} ::_{\textit{TExp}} \texttt{D}_2}{\Delta \vdash \langle \textsf{letrec}\ \texttt{x} : \texttt{D}_1 \vec{\to} \texttt{D}_2 := \texttt{x}_2\ .\ \texttt{e},\ E\rangle \Downarrow [\texttt{x}_1 \mapsto \texttt{D}_1 \vec{\to} \texttt{D}_2]E}$$

$$\text{[ALL-DATA]:}\quad \frac{\forall i : \Delta \vdash \langle \texttt{Df}_i,\ E_{i-1}\rangle \Downarrow E_i}{\Delta \vdash \langle \overset{n}{\overrightarrow{\texttt{Df}_i}},\ E_0\rangle \Downarrow E_n}$$

Similarly to type aggregation, value aggregation is performed through ***BindData***, with the following signature:

$$\textbf{\textit{BindData}} : \bar{\Delta} \vdash \overrightarrow{\textsf{Def}} \times \mathit{Env} \Downarrow \mathit{Env}$$

The rules defining ***BindData*** are as follows:

$$\text{[DATA]:}\quad \frac{e \vdash \texttt{e} \Downarrow_{\textit{Exp}} \omega}{\Delta \vdash \langle \textsf{let}\ \texttt{x}_1 := \texttt{e},\ e\rangle \Downarrow [\texttt{x} \mapsto \omega]e}$$

$$\text{[DATAREC]:}\quad \Delta \vdash \langle \textsf{letrec}\ \texttt{x}_1 : \_ := \texttt{x}_2\ .\ \texttt{e},\ e\rangle \Downarrow [\texttt{x} \mapsto \langle\!\langle e, \texttt{x}_1, \lambda \texttt{x}_2.\texttt{e}\rangle\!\rangle]e$$

$$\text{[ALL-DATA]:}\quad \frac{\forall i : \Delta \vdash \langle \texttt{Df}_i,\ e_{i-1}\rangle \Downarrow e_i}{\Delta \vdash \langle \overset{n}{\overrightarrow{\texttt{Df}_i}},\ e_0\rangle \Downarrow e_n}$$

2.7.3 Transition system definitions

Transition system definition binding is handled through transition system ***BindTSys***. Its initial configuration is an abstract syntax tree pruned to include only transition system definitions. It updates both the type and semantic environments of transition systems. It has the following signature:

$$\textbf{\textit{BindTSys}} : \overrightarrow{\textsf{Def}} \Downarrow \mathit{Env}^{\Downarrow} \times \mathit{Env}_T^{\Downarrow}$$

Its only rule specifies defined over a vector of transition system definitions. When iterating over every component in order of declaration, bind the current transition system variable to its value and types in the previously generated respective

environments. Formally, the rule is:

$$\text{[ALL-SYSTEMS]:}\quad \frac{\begin{array}{c}\forall i : e_i^{\Downarrow} = [\mathtt{x}_i^{\mathtt{T}} \mapsto \mathtt{T}_i]e_{i-1}^{\Downarrow} \\ \forall i : E_i^{\Downarrow} = [\mathtt{x}_i^{\mathtt{T}} \mapsto (\mathtt{D}_{1i}\ \mathtt{|\text{-}}\ \mathtt{D}_{2i} \mathtt{=\!>}\ \mathtt{D}_{3i})]E_{i-1}^{\Downarrow}\end{array}}{\overrightarrow{\mathsf{system}\ \mathtt{D}_{1i}\ \vdash\ \mathtt{D}_{2i}\ \Downarrow\ \mathtt{D}_{3i} : \mathtt{x}_i^{\mathtt{T}} := \mathtt{T}_i}^{n} \Downarrow \langle e_n^{\Downarrow}, E_n^{\Downarrow}\rangle}$$

where n is the number of transition system definitions in a specification, and $\forall\ \mathtt{x}^{\mathtt{T}}$, it is the case that $e_0^{\Downarrow}(\mathtt{x}^{\mathtt{T}})$ and $E_0^{\Downarrow}(\mathtt{x}^{\mathtt{T}})$ are initially undefined.

Following binding, transition systems can be checked for type-correctness. This is achieved through ***AllTTSystem***, with the following signature:

$$\textbf{\textit{AllTTSystem}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \overrightarrow{\mathsf{Def}} ::^{\checkmark}$$

It is defined according to the following rule:

$$\text{[TYPE-CHECK-SYSTEMS]:}\quad \frac{\forall i : E, [\phi \mapsto \mathtt{D}_{1i}\ \vdash\ \mathtt{D}_{2i}\ \Downarrow\ \mathtt{D}_{3i}]E^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{T}_i ::^{\checkmark}}{E, E^{\Downarrow} \overset{\Delta}{\vdash} \overrightarrow{\mathsf{system}\ \mathtt{D}_{1i}\ \vdash\ \mathtt{D}_{2i}\ \Downarrow\ \mathtt{D}_{3i} : \mathtt{x}_i^{\mathtt{T}} := \mathtt{T}_i}^{n} ::^{\checkmark}}$$

2.7.4 Evaluations

The last elements in the spine to be examined are evaluations. For type checking, ***AllTEval*** is used, with the following signature:

$$\textbf{\textit{AllTEval}} : Env_T \times Env_T^{\Downarrow} \overset{\Delta}{\vdash} \overrightarrow{\mathsf{Eval}} ::^{\checkmark}$$

It type checks all evaluations by using the previously generated environments for domains, types of variables, and types of transition systems. The rule:

$$\text{[TYPE-CHECK-ALL-EVALS]:}\quad \frac{\forall i : E, E_T^{\Downarrow} \overset{\Delta}{\vdash} \mathtt{Ev}_i ::^{\checkmark}_{\textbf{\textit{TEval}}}}{E, E_T^{\Downarrow} \overset{\Delta}{\vdash} \overrightarrow{\mathtt{Ev}_i}^{n} ::^{\checkmark}}$$

If all evaluations are well-typed, they can be carried out, using the semantic environments for variables and transition systems. The transition system is ***AllEval***, with the signature:

$$\textbf{\textit{AllEval}} : Env \times Env^{\Downarrow} \vdash \overrightarrow{\mathsf{Eval}} \Downarrow \overrightarrow{\Omega}$$

It evaluates to a vector of values, where each of its component is the result of the corresponding evaluation. The rule is given as:

$$\text{[ALL-EVALS]:}\quad \frac{\forall i : \mathrm{Ev}_i \Downarrow_{\boldsymbol{Eval}} \omega_i}{e, e^{\Downarrow} \vdash \overrightarrow{\mathrm{Ev}_i}^{n} \Downarrow \overrightarrow{\omega_i}^{n}}$$

Chapter 3

Design decisions and implementation

Having established the metalanguage semantics, this chapter provides the reasoning behind some of the design decisions, as well as a brief overview of the implementation.

3.1 Pedagogical considerations

One of the stated goals of the metalanguage is to be used in a pedagogical context. As such, the syntax of transition systems has been constructed to resemble that of natural semantics found in textbooks as much as possible.

3.1.1 Separating syntax and domain definitions

From a purely practical perspective, syntax definitions can be trivially simulated through the use of union domain types. However, considering the pedagogical nature of the metalanguage, there is an advantage in giving dedicated constructs for abstract syntax definitions and formation rules. This brings their syntax closer to corresponding textbook depictions, and highlights their role in a specification. In addition, when it comes to typesetting definitions, the LaTeX pretty-printer behaves differently depending on whether union domains were initially a syntax definition, or not.

3.1.2 Syntactic support for specifying binding models

Other highlights include a separation between binding environment and the initial configurations. Given the pattern matching capabilities of configurations, coupled

with product domains, these can be trivially represented as a pair in the initial configuration. However, the distinction brings it to parity with textbook representations of judgements used in operational semantics. This separation would be further justified if small-steps semantics would be considered as a potential extension of the metalanguage.

3.1.3 Using the functional paradigm

As highlighed in sections 1.2 and 2.4.3.2, the expressions used in the metalanguage for expressing general behaviors not necessarily dependent on transition systems are in the functional paradigm. This paradigm was chosen specifically to help make the metalanguage more declarative, and since it is more likely for 4^{th} semester students to be less resistant to using it. It also aligns with textbook representations of specifications, where mathematical definitions are used for describing machinery such as states or environments. To take further advantage of this, additional constructs, like binding update, are introduced.

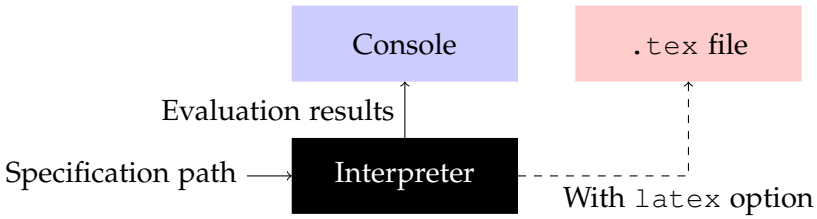
3.1.4 Departures from textbook representations

One exception is the fact that rules are given in reverse order starting with the conclusion, followed by the premises. While this change is a departure from typical pen-and-paper representations of natural semantics, it is the author's opinion that writing rules starting with the conclusion generally feels more natural. This is specifically because, this way, pairing with labels is more direct, and it seems easier to typeset rules, given the rather limited format that text can be expressed in for most text editors used for programming, as opposed to richer text renderers like LaTeX. The generated LaTeX is rendered in a more traditional way by comparison.

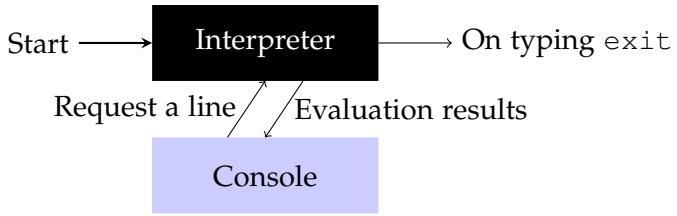
3.2 Implementation

The metalanguage is implemented in Haskell, as a Stack[24] project. Stack is a package and dependency manager for Haskell, built on top of Cabal[4], with many advanced features, such as streamlined execution and building pipelines, and localized, project-specific GHC[7] installations.

Starting with a brief overview of the execution workflow, the interpreter is a simple executable, which can be interacted with in two ways. The first involves giving a file path, representing the input specification, as an argument, evaluating it and, if `latex` was given as an additional argument, generating a `.tex` file from the specification. The second way involves running it without any arguments, in which case the execution will invoke a simplistic, state-less REPL. The REPL can take one-line specification and evaluate them ad-hoc. The first way is intended for



(a) Evaluating a file



(b) Interactive mode

Figure 3.1: Execution overview: 3.1a when giving a file path as a parameter; 3.1b when ommiting the file path, running in interactive mode.

serious uses, while the second is intended for quickly testing short phrases in the metalanguage. Figure 3.1 gives an overview of both workflows.

The following sections will examine the architecture of the project used to generate the executable, starting with the front-end of the interpreter and progressing towards the back-end.

3.2.1 Front-end

The interpreter syntax is expressed through the **L**abelled **B**NF **G**rammar **F**ormalism[3] (LBNF). BNF Converter (BNFC) is a parser generator that supports LBNF grammars, interpreting them and generating language front-end implementations in multiple languages as its back-end. Supported languages include Haskell[10], Java[13], and C++[25], among others.

Since the semantics of the metalanguage are implemented in Haskell, it outputs equivalent Alex[2] and Happy[8] files. BNFC also generates a module defining an AST, based on the LBNF grammar.

Figure 3.2 gives an overview of the architecture of the front-end. Whenever there are changes in the LBNF grammar, BNFC has to be executed to generate the new `.x` and `.y` files, which are then executed with Alex and Happy, respectively, to generate the lexing and parsing modules used by the interpreter. Following this, the project needs to be re-built with Stack.

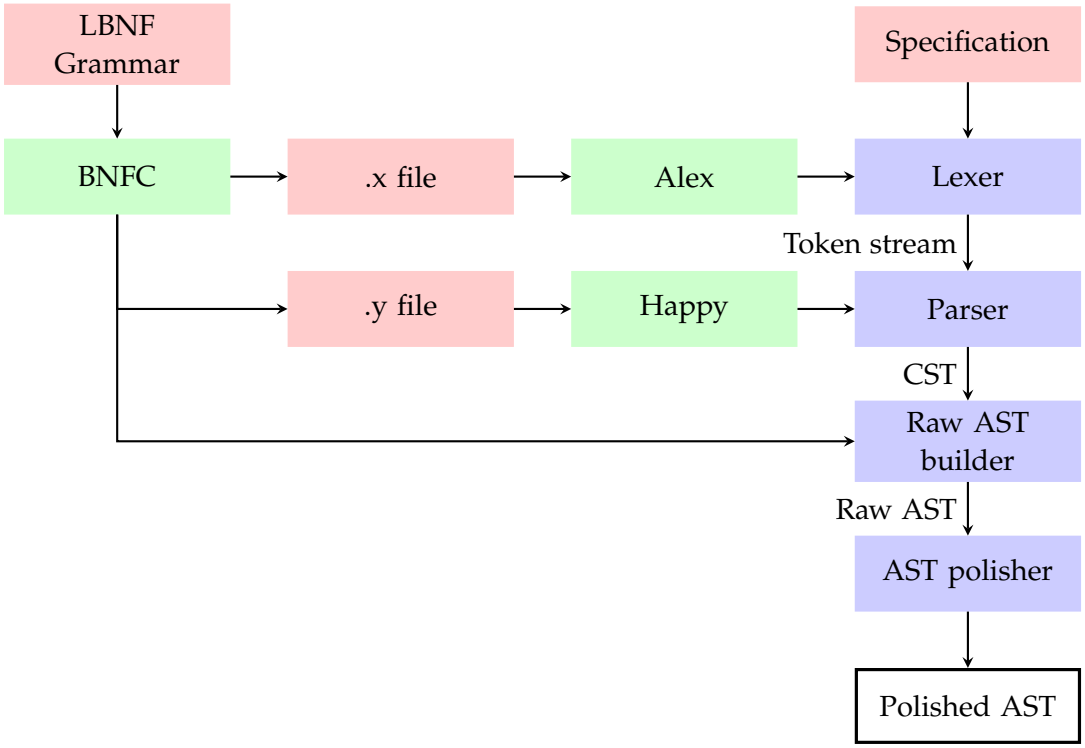


Figure 3.2: Interpreter front-end

Whenever a specification is given as input, it follows a typical interpreter workflow, lexing, parsing and building an AST. While the auto-generated abstract syntax module provided by BNFC is reasonable interpretation, trimming down the concrete syntax considerably, it is not yet in perfect agreement with the abstract syntax presented in chapter 2. As such, it requires some additional post-processing, such as injections of ϵ terms, which cannot be expressed syntactically, removal of superfluous nodes, and transforming specific formations given as lists into binary trees.

3.2.2 Evaluating a specification

The evaluation of a secification is performed through multiple passes over the polished AST. It has the following steps:

1. *Domain sanity checks*: it involves creating the domain environment and a tag table, mapping tags to their parent and attached domain, checking for free domain variables, and checking that named domains are well-formed and, in the case of unions, well-ordered. Figure 3.3 provides an overview of this process.

2. *Binding and type checking data*: it establishes the type environment and sequentially checks every expression in data declarations for type safety.
3. *Binding and type checking transition systems and evaluations*: in this step, transition system are appropriately bound by their names in the type environment. This is followed by type checking them. With these bindings, and those for data, all evaluations can now also be type checked. Figure 3.4 gives an overview of this step in conjunction with step 2.
4. *Evaluating data and configurations*: if type-safe, a specification can be evaluated. Data is evaluated sequentially, following the declaration order, and building the semantic environment along the way. This is followed by performing all declared evaluations. All evaluation results are printed to the console, listing the binding environment (if any), and the initial and final configurations as values. In the case of expression evaluations, a transition is shown from the initial expression to the resulting value. Figure 3.5 gives on overview of the evaluation process.

3.2.3 Back-end

If `latex` is provided as an option, then a `.tex` file with the same name as the input specification is created in folder `dist`, at the location the tool was run. The generated file will begin by including some boilerplate commands and packages, which can be directly copied into a LaTeX report. These commands and packages were, in fact, used throughout this document as well. It is then followed by all domain, transisiton system, abstract syntax and data definitions. Each of these constructs is typeset differently. The contents of this file can be used in a report, although, depending on the needs and size of specific expressions, might need some additional manual type-setting on the students' part. Additionally, the abstract syntax is currently not as well supported as in LETOS[9],

3.2.4 Limitations and weaknesses

3.2.4.1 Rule selection

One of the key limitations of this particular natural semantics implementation is the rule selection mechanism. The rules are investigated in the order in which they are declared. As such, when rules overlap in the pattern matching process, they need to be declared in order from the most specific to the most general.

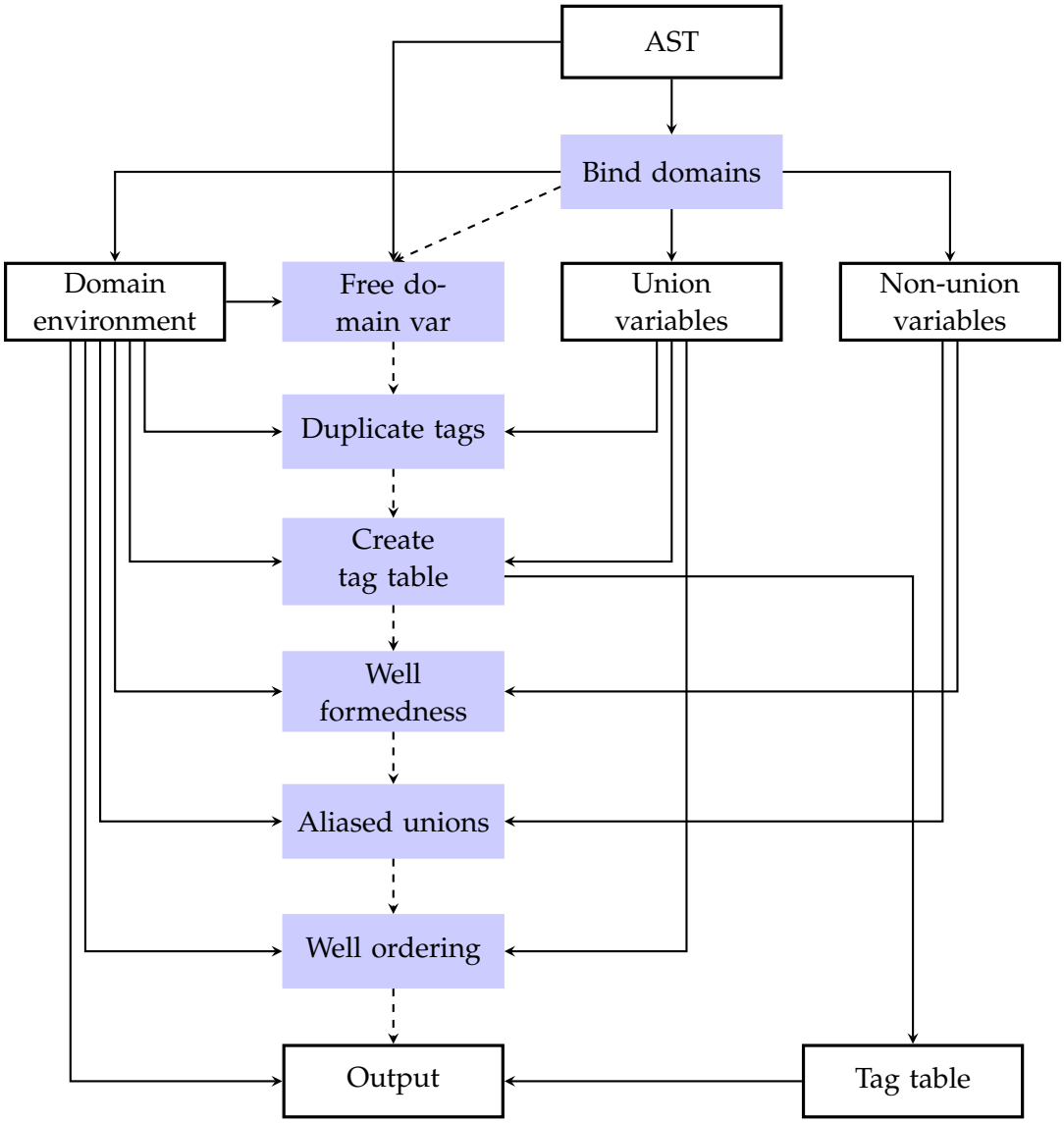


Figure 3.3: Specification evaluation - checking domain definition correctness

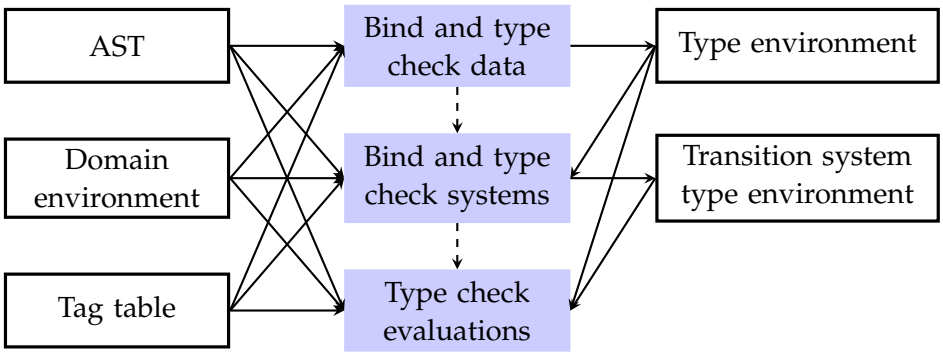


Figure 3.4: Specification type checking

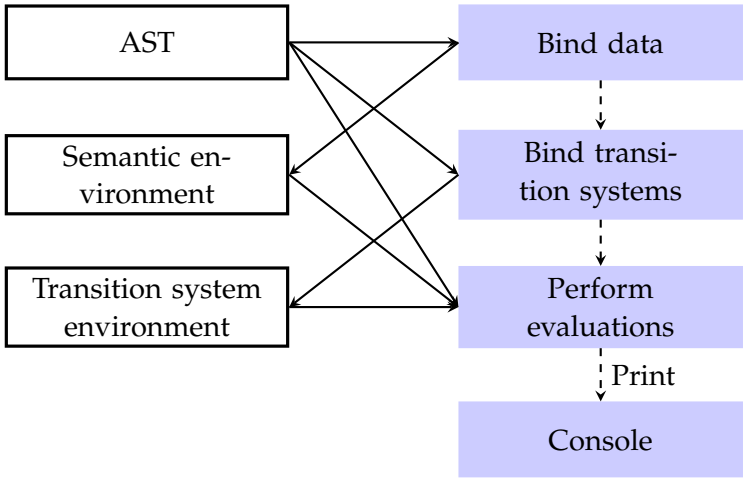


Figure 3.5: Specification evaluations

Example 3.1 (Overlapping rules)

$$\mathtt{g} ::= \textbf{term}\ \mathtt{c} \mid ...$$

$$\text{[GENERAL]:}\quad \mathtt{g} \Downarrow v$$

$$\text{[CONCRETE]:}\quad \textbf{term}\ \mathtt{c} \Downarrow v'$$

An equivalent metalanguage implementation would be:

```
syntax g = term of c | ...

...
    [[ GENERAL ]]: g ==> v;

    [[ CONCRETE ]]: term[c] ==> v';
```

Following the order of these rules in a specification implementation will result in a situation where GENERAL would apply for any term g before CONCRETE, even if said term is actually of the form **term** c.

Another consequence of this approach is that any rule following a diverging rule in the declaration order would not be applied, even if, from a purely theoretical perspective, there would have eventually been a successful application.

Example 3.2 (Diverging rules)

$$\text{[DIVERGE]:}\quad \frac{\mathtt{T} \Downarrow v}{\mathtt{T} \Downarrow v}$$

$$\text{[HALT]:}\quad \mathtt{T} \Downarrow 0$$

In a natural semantics, rule application is non-deterministic, and, if any rule produces a closed derivation tree, then the final configuration of said rule is the result of an evaluation. In the above rules, a semantics is defined for T, evaluating to v' by applying HALT at some point. As such, we can disregard an infinite application of DIVERGE.

However, a one-to-one metalanguage transcription of said rules would be given as:

```
[[ DIVERGE ]]: T ==> v \\
    T ==> v;

[[ HALT ]]: T ==> v';
```

In the current implementation, the rule selection strategy during an evaluation would result in applying DIVERGE first every time for any `T`, and causing the interpretation to recurse infinitely, never reaching HALT.

3.2.4.2 Taxing memory requirements

Another weakness comes in the form of performance when it comes to memory management, specifically around function closures, a lot of these issues stemming from the underlying implementation in Haskell. Since data in Haskell is immutable[11], whenever the semantic or type environments require an update or preservation, they are copied in full. Perhaps the biggest offender is the binding update construct, which will copy the environment at the time of binding, in separate instances for both the input and output expressions. These issues can potentially be addressed or mitigated with a different strategy for expressing binding updates, either by evaluating the expressions beforehand, and extending the syntax and semantics of expressions with values, or by using a different formation rule where the environment only needs to be copied once.

3.2.4.3 Error tracing

While LBNF grammars and BNFC allowed very quick deployment of the front-end of the metalanguage, through many of its features, there were issues with developing the error pipeline in such a way that token position is also used in messages. One of the features LBNF is applying the keyword `position` to a token definition, allowing to capture its position in a parsed file. The problem lies in the fact that, if another literal matching the same regular expression is used somewhere in the specification, parse errors are introduced as a result of the confusion between the two lexemes.

Example 3.3 (Error pipeline complications)
One of the main selling points of this formalism was ease of on-boarding through some advanced parsing features like `terminator` and `separator`, for quickly expressing common patterns in production rules. As an example, `separator`

`nonempty Rule ";"` would automatically generate the following production rules:

```
Rules ::= Rule ";" Rules;
Rules ::= Rule;
```

The problem is that, according to the specifications of LBNF, a separator can only be given as a string literal, as opposed to any token, including custom defined ones. Consider a token given as `position token Semi {";"}`. In its current implementation, BNFC assumes the one declared with `position` and the one from `separator` are different tokens, where one supersedes the other. This leads the lexer to resolve all occurrences of `";"` to the same definition, further leading to spurious parse errors.

3.2.4.4 Generated LaTeX

The generated LaTeX is serviceable, but there are many potential improvements that can be made, taking LETOS and Typol as inspiration. Perhaps the biggest disadvantage at the moment is that `.text` file generation is completely agnostic of the actual evaluation. One particularly notable example is that, while for definitions there is a distinction made between unions and syntax definitions, the same cannot be said for patterns expressed in rules. Currently, syntax constructors are printed in the same manner as tags, which can lead to some confusion and a requirement for additional manual typesetting of rules.

Chapter 4

Conclusion and future work

The metalanguage provides an initial tool-set for expressing the semantics of languages through natural semantics. Being primarily addressed at students working on semester language projects, a key focus was the development of a syntax in line with textbooks on natural semantics. The implementation allows checking specifications for well-formedness and type safety, performing ad-hoc evaluations, to ensure behavioral correctness in semantics, and generating reasonable quality LaTeX code.

The expression semantics can also serve as a gentle introduction to the functional paradigm, for 4^{th} semester students that are mostly unacquainted with it. However, while it includes some features, like higher-order functions and pattern matching for configurations, it lacks the advanced features of a fully-fledged functional language, like its implementation language, Haskell. Examples include a polymorphic type system and lazy evaluation.

Section 3.2.4 provides a comprehensive account of many of the shortcomings with the current implementation. More efforts can be made towards improving the writability, pretty printing when it comes to the expressiveness of abstract syntax. In the current iteration, abstract syntax formation rules are given through the tag system, enforcing an exclusively prefix notation. Introducing mixfix[6] term constructors, or some template mechanism, would bring the metalanguage closer to textbook representations. Additional work is required for error signaling, in support of the pedagogical aspects.

But perhaps the most useful missing feature would be specification transpilation, targeting popular general-purpose languages, such as Java, Haskell, C++ or C#.

Bibliography

[1] *Aalborg University - Languages & Compilers*. https://moduler.aau.dk/course/2019-2020/DSNCSITK203?lang=en-GB.

[2] *Alex: A lexical analyser generator for Haskell*. https://www.haskell.org/alex/.

[3] *BNFC*. https://bnfc.digitalgrammars.com/.

[4] *Cabal*. https://www.haskell.org/cabal/.

[5] Arthur Charguéraud. "Pretty-Big-Step Semantics". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 41–60. DOI: 10.1007/978-3-642-37036-6_3. URL: https://doi.org/10.1007/978-3-642-37036-6_3.

[6] Nils Anders Danielsson and Ulf Norell. "Parsing Mixfix Operators". In: *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Vol. 5836. Lecture Notes in Computer Science. Springer, 2008, pp. 80–99. DOI: 10.1007/978-3-642-24452-0_5. URL: https://doi.org/10.1007/978-3-642-24452-0_5.

[7] *GHC*. https://www.haskell.org/ghc/.

[8] *Happy: The parser generator for Haskell*. https://www.haskell.org/happy/.

[9] Pieter H. Hartel. "LETOS - a lightweight execution tool for operational semantics". In: *Softw. Pract. Exp.* 29.15 (1999), pp. 1379–1416. DOI: 10.1002/(SICI)1097-024X(19991225)29:15\<1379::AID-SPE286\>3.0.CO;2-V. URL: https://doi.org/10.1002/(SICI)1097-024X(19991225)29:15\<1379::AID-SPE286\>3.0.CO;2-V.

[10] *Haskell*. https://www.haskell.org/.

[11] *Haskell Garbage Collection*. https://wiki.haskell.org/GHC/Memory_Management.

[12] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN: 978-0-521-14709-5. URL: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/transitions-and-trees-introduction-structural-operational-semantics.

[13] *Java*. https://www.java.com/en/.

[14] *JavaScript*. https://www.javascript.com/.

[15] Gilles Kahn. "Natural Semantics". In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39. ISBN: 3-540-17219-X. DOI: 10.1007/BFb0039592. URL: https://doi.org/10.1007/BFb0039592.

[16] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988. ISBN: 0-13-110370-9. URL: https://en.wikipedia.org/wiki/The_C_Programming_Language.

[17] Peter Lee and Uwe F. Pleban. "A Realistic Compiler Generator Based on High-Level Semantics". In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 284–295. DOI: 10.1145/41625.41651. URL: https://doi.org/10.1145/41625.41651.

[18] *Miranda*. http://miranda.org.uk/.

[19] Lawrence C. Paulson. "A Semantics-Directed Compiler Generator". In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. Ed. by Richard A. DeMillo. ACM Press, 1982, pp. 224–233. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582178. URL: https://doi.org/10.1145/582153.582178.

[20] Mikael Pettersson. "RML - A New Language and Implementation for Natural Semantics". In: *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September 14-16, 1994, Proceedings*. Ed. by Manuel V. Hermenegildo and Jaan Penjam. Vol. 844. Lecture Notes in Computer Science. Springer, 1994, pp. 117–131. DOI: 10.1007/3-540-58402-1_10. URL: https://doi.org/10.1007/3-540-58402-1_10.

[21] *Python*. https://www.python.org/.

[22] Georgian-Vlad Saioc. *Tools for Generating Interpreters and Compilers - A state of the Art*. 2020.

[23] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[24] *Stack*. `https://docs.haskellstack.org/en/stable/README/`.

[25] Bjarne Stroustrup. *The C++ programming language - special edition (3. ed.)* Addison-Wesley, 2007. ISBN: 978-0-201-70073-2.

[26] *Top 7 Programming Languages Of 2020*. `https://www.codingdojo.com/blog/top-7-programming-languages-of-2020`.

[27] *Typol*. `http://www-sop.inria.fr/members/Laurent.Hascoet/papers/CDDHK85.html`.

Appendix A

Well-formedness algorithms

A.1 Non-unions

Well-formedness of non-unions involves unfolding their definitions and analyzing the dependencies between domain variables. The semantics is given by the transition system ***WF*** with the following signature:

$$\textbf{\textit{WF}}: \ \bar{\Delta} \vdash \mathsf{Dom} \times P \times \mathsf{Var} \Downarrow P$$

The member of Var denotes the name of the domain that is currently unfolded (the "host"), and members of P bind domain variables to the set of their parents. The domain P is defined as:

$$P = \mathsf{Var} \to \mathcal{P}(\mathsf{Var})$$

The rules of ***WF*** ensure that a specification has only well-formed domains. Several axioms and rules cover the cases where no domain variables, or variables bound to union domain are involved. Basic domains are considered inherently well-formed. Function and product domains are considered well-formed if their constituents are well-formed. Union domains are a special case and considered well-formed by

default in this algorithm[1]. The axioms and rules are:

$$\text{[CONST]:}\quad \Delta \vdash \langle \mathtt{B},\ p,\ \mathtt{x} \rangle \Downarrow p$$

$$\text{[UNION-VAR]:}\quad \Delta \vdash \langle \mathtt{x}_1,\ p,\ \mathtt{x}_2 \rangle \Downarrow p \quad \text{if } \Delta(\mathtt{x}_1) \in \mathbf{Dom}_{\bar{\cup}}$$

$$\text{[FUNCTION]:}\quad \frac{\Delta \vdash \langle \mathtt{D}_1,\ p,\ \mathtt{x} \rangle \Downarrow p'' \qquad \Delta \vdash \langle \mathtt{D}_2,\ p'',\ \mathtt{x} \rangle \Downarrow p'}{\Delta \vdash \langle \mathtt{D}_1 \bar{\to} \mathtt{D}_2,\ p,\ \mathtt{x} \rangle \Downarrow p'}$$

$$\text{[PRODUCT]:}\quad \frac{\Delta \vdash \langle \mathtt{D}_1,\ p,\ \mathtt{x} \rangle \Downarrow p'' \qquad \Delta \vdash \langle \mathtt{D}_2,\ p'',\ \mathtt{x} \rangle \Downarrow p'}{\Delta \vdash \langle \mathtt{D}_1 \bar{\times} \mathtt{D}_2,\ p,\ \mathtt{x} \rangle \Downarrow p'}$$

VAR is the rule describing how "guest" domain variables pointing to non-unions are handled. If the "host" variable is the same as the "guest", the domain is self-referencing, meaning it is ill-formed. Another case is if the "guest" variable is one of the parents of the "host". If the constraints are satisfied, ***WF*** proceeds by referring to the bound domain of the "guest", assigning the "guest" as the new "host", and updating the parent environment, where the "guest" adds to its own parents the "host" and all of its parents.

$$\text{[VAR]:}\quad \frac{\Delta \vdash \langle \Delta(\mathtt{x}_1),\ [\mathtt{x}_1 \mapsto p(\mathtt{x}_1) \cup p(\mathtt{x}_2) \cup \{\mathtt{x}_2\}]p,\ \mathtt{x}_1 \rangle \Downarrow p'}{\Delta \vdash \langle \mathtt{x}_1,\ p,\ \mathtt{x}_2 \rangle \Downarrow p'} \quad \begin{array}{l} \text{if } \Delta(\mathtt{x}_1) \notin \mathbf{Dom}_{\bar{\cup}}, \\ \text{if } \mathtt{x}_1 \notin p(\mathtt{x}_2) \\ \text{if } \mathtt{x}_1 \neq \mathtt{x}_2 \end{array}$$

A.2 Unions

The well-ordered relation is represented by $\downarrow$. Basic domains are considered inherently well-ordered. Well-ordered product and function domains are formed from well-ordered constituents. Variables bound to well-ordered non-union domains are themselves considered well-ordered. These cases are expressed by the following

[1]Their restrictions are handled in section A.2

axioms and rules:

$$\text{[CONST]:}\quad \Delta \vdash\!\downarrow \mathtt{B}$$

$$\text{[FUNCTION]:}\quad \frac{\Delta \vdash\!\downarrow \mathtt{D}_1 \qquad \Delta \vdash\!\downarrow \mathtt{D}_2}{\Delta \vdash\!\downarrow \mathtt{D}_1 \bar{\to} \mathtt{D}_2}$$

$$\text{[PRODUCT]:}\quad \frac{\Delta \vdash\!\downarrow \mathtt{D}_1 \qquad \Delta \vdash\!\downarrow \mathtt{D}_2}{\Delta \vdash\!\downarrow \mathtt{D}_1 \bar{\times} \mathtt{D}_2}$$

$$\text{[NON-UNION-VAR]:}\quad \frac{\Delta \vdash\!\downarrow \Delta(\mathtt{x})}{\Delta \vdash\!\downarrow \mathtt{x}} \quad \text{if } \Delta(\mathtt{x}) \notin \mathbf{Dom}_{\bar{\cup}}$$

For a union domain to be well-ordered, at least one of its tags needs to refer to a well-ordered domain..[2] Formally, this rule is expressed as:

$$\text{[UNION-VAR]:}\quad \frac{\exists\ \mathtt{t},\ \mathtt{D}\ \text{s. t.}\ \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x})) \wedge \downarrow \mathtt{D}}{\Delta \vdash\!\downarrow \mathtt{x}} \quad \text{if } \Delta(\mathtt{x}) \in \mathbf{Dom}_{\bar{\cup}}$$

[2]Here the choice is presented as being non-deterministic. In the actual implementation, a historical record of visited tags is kept, ensuring no tag is visited more than once downstream in the same traversal. This safeguards termination of the algorithm.

Appendix B

General expression semantics

This section covers the run-time semantics and type system of commonplace language constructs, such as arithmetic and boolean operations, or functional elements like anonymous functions, application, or local declarations.

B.1 Run-time semantics

B.1.1 Arithmetic expressions

Arithmetic operations are the part of the metalanguage that allow users to express the mathematical meaning behind language constructs in a specification. Arithmetic operations include addition, subtraction, multiplication, division, modulo, and additive inversion.

For brevity, the semantics of binary arithmetic operators is given in a more generalized form. Consider the following set of pairs between arithmetic operators (composition between division and rounding down for /) in prefix notation, and members of σ_A:

$$\mathscr{A}_{op} = \{(+,\ +), (-,\ \text{-}), (\cdot,\ \textbf{*}), (/ \circ \lfloor\_\rfloor,\ /), (mod,\ \textbf{\%})\}$$

The following generalized rule describes the semantics of arithmetic operators[1]:

$$\text{[ARITHMETIC-BIN]:}\quad \frac{e \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad e \vdash \mathsf{e}_2 \Downarrow \omega_2}{e \vdash \mathsf{e}_1\ \textbf{op}\ \mathsf{e}_2 \Downarrow op(\omega_1, \omega_2)} \qquad \forall\ (op,\ \textbf{op}) \in \mathscr{A}_{op}$$

[1] Division and modulo include side-conditions disallowing division by 0

For additive inversion, the operational semantics is:

$$\text{[INV]:} \quad \frac{e \vdash \mathsf{e} \Downarrow \omega}{e \vdash \texttt{-}\mathsf{e} \Downarrow -\omega}$$

B.1.2 Relational expressions

Relational expressions permit comparisons between the values of expressions in the metalanguage. Operations of this nature include checking for equality or inequality between values, as well as comparisons if the expressions belong to ordered sets.

Similarly to arithmetic expressions, relational operators are expressed through usage of the set:

$$\mathscr{R}_{op} = \{(=,\ =), (\neq,\ \texttt{!=}),\ (\leq,\ \texttt{<=}), (\geq,\ \texttt{>=}), (<,\ <), (>,\ >)\}$$

with their semantics given by the generalized rule:

$$\text{[RELATIONAL]:} \quad \frac{e \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad e \vdash \mathsf{e}_2 \Downarrow \omega_2}{e \vdash \mathsf{e}_1\ \mathbf{op}\ \mathsf{e}_2 \Downarrow op(\omega_1, \omega_2)} \qquad \forall\ (op,\ \mathbf{op}) \in \mathscr{R}_{op}$$

B.1.3 Boolean expressions

The metalanguage also allows logical operations, namely conjunction, disjunction and negation, the semantics of which being:

$$\text{[AND]:} \quad \frac{env \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad env \vdash \mathsf{e}_2 \Downarrow \omega_2}{env \vdash \mathsf{e}_1\ \texttt{\&}\ \mathsf{e}_2 \Downarrow \omega_1 \wedge \omega_2}$$

$$\text{[OR]:} \quad \frac{env \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad env \vdash \mathsf{e}_2 \Downarrow \omega_2}{env \vdash \mathsf{e}_1\ \texttt{|}\ \mathsf{e}_2 \Downarrow \omega_1 \vee \omega_2}$$

$$\text{[NEG]:} \quad \frac{env \vdash \mathsf{e} \Downarrow \omega}{env \vdash \texttt{!}\ \mathsf{e} \Downarrow \neg\omega}$$

B.1.4 String expressions

The only allowed operation for strings is concatenation. The corresponding rule is:

$$\text{[CONCAT]:} \quad \frac{e \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad e \vdash \mathsf{e}_2 \Downarrow \omega_2}{e \vdash \mathsf{e}_1\ \texttt{++}\ \mathsf{e}_2 \Downarrow \omega_1\omega_2}$$

B.1.5 Functional elements

The semantics of anonymous functions is given by the axiom:

$$\text{[LAMBDA]:}\quad e \vdash \textsf{lam}\ \mathtt{x} : \_\ .\ \mathtt{e} \Downarrow \langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle$$

meaning any λ-expression evaluates to a closure between environment *env* and itself.

Local declarations allow the bindings of variables to values within the scope of one expression. That expression is evaluated with the updated environment, and the resulting value is propagated as the evaluation value of the let-expression. The semantics is denoted by the rule:

$$\text{[LET]:}\quad \frac{e \vdash \mathtt{e}_1 \Downarrow \omega' \qquad [\mathtt{x} \mapsto \omega']e \vdash \mathtt{e}_2 \Downarrow \omega}{e \vdash \textsf{let}\ \mathtt{x} := \mathtt{e}_1\ \textsf{in}\ \mathtt{e}_2 \Downarrow \omega}$$

The metalanguage also allows local declarations to be recursive, albeit in a more restricted fashion, only allowing direct definitions of functions. A recursive declaration involves $\mathtt{x}_1$ as the name of the declared variable, $\mathtt{x}_2$ denoting the input variable, and an expression acting as the body of the function. Together with the current environment, these are used to construct the recursive closure to which $\mathtt{x}_1$ will be bound when evaluating the inner expression $\mathtt{e}_2$. The rule is defined as:

$$\text{[LETREC]:}\quad \frac{[\mathtt{x}_1 \mapsto \langle\!\langle e, \mathtt{x}_1, \lambda \mathtt{x}_2.\mathtt{e}_1 \rangle\!\rangle]e \vdash \mathtt{e}_2 \Downarrow \omega}{e \vdash \textsf{letrec}\ \mathtt{x}_1 : \_ := \mathtt{x}_2\ .\ \mathtt{e}_1\ \textsf{in}\ \mathtt{e}_2 \Downarrow \omega}$$

Function application requires that the value to be applied is a closure. The argument is then evaluated and the variable representing the input of the λ-expression is mapped to this value in the environment of the closure. It involves two rules, one for each type of closure:

$$\text{[APP-1]:}\quad \frac{e \vdash \mathtt{e}_1 \Downarrow \langle\!\langle e', \lambda \mathtt{x}.\mathtt{e}' \rangle\!\rangle \qquad e \vdash \mathtt{e}_2 \Downarrow \omega' \qquad [\mathtt{x} \mapsto \omega']e' \vdash \mathtt{e}' \Downarrow \omega}{e \vdash \textsf{app}\ \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \omega}$$

$$\text{[APP-2]:}\quad \frac{\begin{array}{c} e \vdash \mathtt{e}_1 \Downarrow \langle\!\langle e', \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e}' \rangle\!\rangle \qquad e \vdash \mathtt{e}_2 \Downarrow \omega' \\ [\mathtt{x} \mapsto \langle\!\langle e', \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e}' \rangle\!\rangle, \mathtt{x}' \mapsto \omega']e' \vdash \mathtt{e}' \Downarrow \omega \end{array}}{e \vdash \textsf{app}\ \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \omega}$$

Metalanguage expressions also include conditional statements. If the conditional expression evaluates to *tt*, the result is that of the expression on the **then** side. A result of *ff* will result in the evaluation of the **else** branch. The rules for conditional

statements are:

$$\text{[IF-TRUE]:}\quad \frac{e \vdash \mathsf{e}_1 \Downarrow tt \qquad e \vdash \mathsf{e}_2 \Downarrow \omega}{e \vdash \textbf{if } \mathsf{e}_1 \textbf{ then } \mathsf{e}_2 \textbf{ else } \mathsf{e}_3 \Downarrow \omega}$$

$$\text{[IF-FALSE]:}\quad \frac{e \vdash \mathsf{e}_1 \Downarrow ff \qquad e \vdash \mathsf{e}_3 \Downarrow \omega}{e \vdash \textbf{if } \mathsf{e}_1 \textbf{ then } \mathsf{e}_2 \textbf{ else } \mathsf{e}_3 \Downarrow \omega}$$

B.1.6 Operations involving pairs

Pair formations can be expressed in the metalanguage. Each member of a pair is itself an expression, and the result is a pair value, where each of its members is the evaluation result of the corresponding expression. The rule is:

$$\text{[PAIR]:}\quad \frac{e \vdash \mathsf{e}_1 \Downarrow \omega_1 \qquad e \vdash \mathsf{e}_2 \Downarrow \omega_2}{e \vdash (\mathsf{e}_1,\ \mathsf{e}_2) \Downarrow (\omega_1,\ \omega_2)}$$

Conversely, the individual values of members can also be extracted from pairs through operators **head** and **tail**:

$$\text{[HEAD]:}\quad \frac{e \vdash \mathsf{e} \Downarrow (\omega_1,\ \omega_2)}{e \vdash \textbf{head } \mathsf{e} \Downarrow \omega_1}$$

$$\text{[TAIL]:}\quad \frac{e \vdash \mathsf{e} \Downarrow (\omega_1,\ \omega_2)}{e \vdash \textbf{tail } \mathsf{e} \Downarrow \omega_2}$$

B.2 Type system

B.2.1 Arithmetic expressions

For arithmetic expressions, the type rules specify that both members of a binary operation must be integers, and the resulting type is itself an integer. The type rules are:

$$\text{[ARITHMETIC-BIN]:}\quad \frac{E \overset{\Delta}{\vdash} \mathsf{e}_1 :: \textbf{int} \qquad E \overset{\Delta}{\vdash} \mathsf{e}_2 :: \textbf{int}}{E \overset{\Delta}{\vdash} \mathsf{e}_1 \textbf{ op } \mathsf{e}_2 :: \textbf{int}} \qquad \forall\ \textbf{op} \in \sigma_A$$

Conversely, the same principle applies for additive inversion, its type rule being:

$$\text{[INV]:}\quad \frac{E \overset{\Delta}{\vdash} \mathsf{e} :: \textbf{int}}{E \overset{\Delta}{\vdash} \text{-}\mathsf{e} :: \textbf{int}}$$

B.2.2 Relation expressions

Relational equations are defined between identical non-empty basic types[2]. The resulting type is that of booleans.

$$\text{[RELATIONAL]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{B} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{B}}{E \overset{\Delta}{\vdash} \mathtt{e}_1 \ \mathbf{op}\ \mathtt{e}_2 :: \mathbf{bool}} \qquad \begin{array}{c}\forall\ \mathbf{op} \in \sigma_R \\ \text{if } \mathtt{B} \not\equiv \epsilon\end{array}$$

B.2.3 Boolean expressions

A boolean expression yields a boolean type, and it is valid only if the sub-expressions are booleans themselves. This applies both to binary and unary boolean expressions.

$$\text{[BOOLEAN-BIN]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathbf{bool} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathbf{bool}}{E \overset{\Delta}{\vdash} \mathtt{e}_1 \ \mathbf{op}\ \mathtt{e}_2 :: \mathbf{bool}} \quad \forall\ \mathbf{op} \in \sigma_B$$

$$\text{[NEG]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathbf{bool}}{E \overset{\Delta}{\vdash} \mathbf{!}\ \mathtt{e} :: \mathbf{bool}}$$

B.2.4 String expressions

String concatenation requires that its sub-terms be of the string type, and the result is, likewise, a string type. The type rule is:

$$\text{[CONCAT]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathbf{str} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathbf{str}}{E \overset{\Delta}{\vdash} \mathtt{e}_1 \mathbin{++} \mathtt{e}_2 :: \mathbf{str}}$$

B.2.5 Functional elements

For λ-expression declarations, the domain of the input is specified as an annotation, whereas the output domain is inferred from its body expression. The type judgement is:

$$\text{[LAMBDA]:}\quad \frac{[\mathtt{x} \mapsto \mathtt{D}_1]E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_2}{E \overset{\Delta}{\vdash} \mathbf{lam}\ \mathtt{x} : \mathtt{D}_1\ \mathtt{.}\ \mathtt{e} :: \mathtt{D}_1 \rightarrow \mathtt{D}_2}$$

[2]Comparisons between non-integers are handled by the underlying Haskell implementation

In a function application, the type of the expression being applied must be a function, with the domains of formal and actual parameters matching. If these conditions are met, the resulting type is that of the inner expression of the function. Formally, it is expressed as:

$$\text{[APP]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}' \qquad E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}' \rightarrow \mathtt{D}}{E \overset{\Delta}{\vdash} \textsf{app}\ \mathtt{e}_1\ \mathtt{e}_2 :: \mathtt{D}}$$

For local declarations, the type of a declared variable, x, results from inferring the type of its bound expression. Then, the inner expression has its type evaluated, with the type environment updated at x. The type of the inner expression serves as the type of the local declaration construct.

$$\text{[LET]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D} \qquad [\mathtt{x} \mapsto \mathtt{D}]E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}'}{E \overset{\Delta}{\vdash} \textsf{let}\ \mathtt{x} := \mathtt{e}_1\ \textsf{in}\ \mathtt{e}_2 :: \mathtt{D}'}$$

The difference between non-recursive and recursive definitions is that, for the latter, the locally declared variable requires a type annotation, which necessarily needs to be an arrow domain. Considering the restriction on how recursive variables are declared, type checking functions as follows:

1. Before type checking the body of the recursive function, $\mathtt{x}_1$ and $\mathtt{x}_2$ are bound to the type annotation and left-hand domain in the arrow type, respectively. This new type environment is used to check the body of the recursive function for type safety.
2. If this succeeds, type environment is again updated at $\mathtt{x}_1$ with the annotation, and used to type check the inner expression.

$$\text{[LETREC]:}\quad \frac{[\mathtt{x}_1 \mapsto \mathtt{D}_1 \rightarrow \mathtt{D}_2,\ \mathtt{x}_2 \mapsto \mathtt{D}_1]E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}_2 \qquad [\mathtt{x}_1 \mapsto \mathtt{D}_1 \rightarrow \mathtt{D}_2]E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}'}{E \overset{\Delta}{\vdash} \textsf{letrec}\ \mathtt{x}_1 : \mathtt{D}_1 \rightarrow \mathtt{D}_2 := \mathtt{x}_2\ .\ \mathtt{e}_1\ \textsf{in}\ \mathtt{e}_2 :: \mathtt{D}'}$$

In a conditional statement, the conditional expression must be a boolean type, and the two branches must yield equivalent types. If this is satisfied, the yielded type is that of the branches. Formally the type rule is:

$$\text{[IF]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \textsf{bool} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_3 :: \mathtt{D}}{E \overset{\Delta}{\vdash} \textsf{if}\ \mathtt{e}_1\ \textsf{then}\ \mathtt{e}_2\ \textsf{else}\ \mathtt{e}_3 :: \mathtt{D}}$$

B.2.6 Operations involving pairs

Whenever a pair is formed, its type is a product type where its members are the types of the children. Formally, the type rule is:

$$\text{[PAIR]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}_1 \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}_2}{E \overset{\Delta}{\vdash} (\mathtt{e}_1,\ \mathtt{e}_2) :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2}$$

The domain of an expression extracted from a **head** or **tail** operation on a pair is the domain of the respective child. The type rules for extracting a member of a pair is:

$$\text{[HEAD]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2}{E \overset{\Delta}{\vdash} \textbf{head}\ \mathtt{e} :: \mathtt{D}_1}$$

$$\text{[TAIL]:}\quad \frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2}{E \overset{\Delta}{\vdash} \textbf{tail}\ \mathtt{e} :: \mathtt{D}_2}$$

Appendix C

Proofs

C.1 Domain equivalence is an equivalence relation

Proving reflexivity, symmetry and transitivity of domain equivalence is achieved through induction on the height of derivation trees[12].

C.1.1 Reflexivity

Domain equivalence is reflexive if, for every $\mathtt{D}$, the following relation holds:

$$\Delta \vdash \mathtt{D} \equiv \mathtt{D} \tag{C.1}$$

To prove this, we must prove that for any $\mathtt{D}$, there exists a derivation tree according to the rules of $\equiv$ for some Δ.

The proof involves induction on the structure of $\mathtt{D}$. The structure of $\mathtt{D}$ correlates with a tree, where the tree is formed as follows:

- According to the formation rules of $\mathtt{D}$, where non-recursive term constructors are leaves, and recursive ones are nodes, with each occurrence of $\mathtt{D}$ within the constructor forming a branch.
- Given a Δ, if $\mathtt{D}$ is of the form $\mathtt{x}$ such that $\Delta(\mathtt{x}) \notin \mathbf{Dom}_{\bigcup}$, then a node is formed for $\mathtt{x}$ and its only branch points to the root of the tree formed by $\Delta(\mathtt{x})$.

If the height of the tree is $n = 0$, there are two cases to consider:

1. $\mathtt{D}$ is of the form $\mathtt{B}$, meaning we can apply axiom CONST. The equivalence holds for every construct formed from $\mathtt{B}$, since, for each of its respective concrete formations, $\mathtt{B} = \mathtt{B}$.

2. $\mathtt{D}$ is of the form $\mathtt{x}$ such that $\Delta(\mathtt{x}) \in \mathbf{Dom}_{\bar{\bigcup}}$. The derivation tree is formed by applying axiom UNION. The equivalence holds, considering that $\mathtt{x} = \mathtt{x}$.

We assume that C.1 holds for all $\mathtt{D}$, where the tree formed from the structure of $\mathtt{D}$ has height $j \leq n$. For a tree of height $n+1$, the following cases are considered:

1. The form of $\mathtt{D}$ is $\mathtt{D}_1 \bar{\times} \mathtt{D}_2$. According to the induction hypothesis, the reflexive equivalences for $\mathtt{D}_1$ and $\mathtt{D}_2$ hold. By application of rule PRODUCT, the following derivation is obtained:

$$\frac{\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_1 \qquad \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}_2}{\Delta \vdash \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \equiv \mathtt{D}_1 \bar{\times} \mathtt{D}_2}$$

   We can apply the same method to obtain the result for $\mathtt{D}_1 \bar{\to} \mathtt{D}_2$.

2. The form of $\mathtt{D}$ is $\mathtt{x}$, such that $\Delta(\mathtt{x}) \notin \mathbf{Dom}_{\bar{\bigcup}}$. According to the induction hypothesis, we have that $\Delta \vdash \Delta(\mathtt{x}) \equiv \Delta(\mathtt{x})$. Using this as a premise, we can reach conclusion $\Delta \vdash \mathtt{x} \equiv \mathtt{x}$ by successively using rules VAR-1 and VAR-2 as follows:

$$\frac{\dfrac{\Delta \vdash \Delta(\mathtt{x}) \equiv \Delta(\mathtt{x})}{\Delta \vdash \Delta(\mathtt{x}) \equiv \mathtt{x}}}{\Delta \vdash \mathtt{x} \equiv \mathtt{x}}$$

This concludes the proof of reflexivity for domain equivalence.

C.1.2 Symmetry

For domain equivalence to be symmetric, for every $\mathtt{D}_1$, $\mathtt{D}_2$, the following relation must hold:

$$\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_2 \implies \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}_1 \tag{C.2}$$

It should be noted that, for all cases where the left-hand side of the implication does not hold, the implication as a whole is true. As such, the following cases need only consider the situations where the left-hand side holds, and prove the right-hand side holds as well.

The proof involves induction on the height of $\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_2$. Where $n = 0$, the cases are as follows:

1. Rule CONST was used, meaning $\mathtt{D}_1$ is of the form $\mathtt{B}_1$, and $\mathtt{D}_2$ is of the form $\mathtt{B}_2$. As a result, the following relation needs to hold:

$$\Delta \vdash \mathtt{B}_1 \equiv \mathtt{B}_2 \implies \Delta \vdash \mathtt{B}_2 \equiv \mathtt{B}_1$$

   Considering the definition of CONST, this is true if $\mathtt{B}_1 = \mathtt{B}_2 \implies \mathtt{B}_2 = \mathtt{B}_1$, for every $\mathtt{B}_1$ and $\mathtt{B}_2$. This follows from the symmetry of = over algebraic data types.

2. Rule UNION was applied, leading to the following case implication:

$$\Delta \vdash \mathtt{x}_1 \equiv \mathtt{x}_2 \implies \Delta \vdash \mathtt{x}_2 \equiv \mathtt{x}_1$$

According to the definition of UNION, $\mathtt{x}_1 = \mathtt{x}_2 \implies \mathtt{x}_2 = \mathtt{x}_1$ needs to be true for the relation to hold. Again, this is satisfied according to the symmetry of $=$.

We assume that relation C.2 holds for derivation trees of height $j \leq n$. For trees of height $n+1$, the following cases are considered:

1. Rule PRODUCT was used. This implies that $\mathtt{D}_1 = \mathtt{D}_1' \bar{\times} \mathtt{D}_2'$ and $\mathtt{D}_2 = \mathtt{D}_1'' \bar{\times} \mathtt{D}_2''$. The case implication is as follows:

$$\frac{\Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_1'' \quad \Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_2''}{\Delta \vdash \mathtt{D}_1' \bar{\times} \mathtt{D}_2' \equiv \mathtt{D}_1'' \bar{\times} \mathtt{D}_2''} \implies \frac{\Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_1' \quad \Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_2'}{\Delta \vdash \mathtt{D}_1'' \bar{\times} \mathtt{D}_2'' \equiv \mathtt{D}_1' \bar{\times} \mathtt{D}_2'}$$

According to induction hypothesis, the following implications are true:

$$\Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_1'' \implies \Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_1'$$
$$\Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_2'' \implies \Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_2'$$

Since $\Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_1'$ and $\Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_2'$ hold, then the equivalence on the right-hand side of the case implication also holds, by application of rule PRODUCT.
We can operate similarly in the case of functions, and obtain positive results.

2. Rule VAR-1 is used, hence $\mathtt{D}_2 = \mathtt{x}$, and the case implication is:

$$\frac{\Delta \vdash \mathtt{D} \equiv \Delta(\mathtt{x})}{\Delta \vdash \mathtt{D} \equiv \mathtt{x}} \implies \frac{\Delta \vdash \Delta(\mathtt{x}) \equiv \mathtt{D}}{\Delta \vdash \mathtt{x} \equiv \mathtt{D}}$$

According to induction hypothesis, we have that:

$$\Delta \vdash \mathtt{D} \equiv \Delta(\mathtt{x}) \implies \Delta \vdash \Delta(\mathtt{x}) \equiv \mathtt{D}$$

Since $\Delta \vdash \Delta(\mathtt{x}) \equiv \mathtt{D}$ holds, for the right-hand side of the case implication to hold, there needs to exist a derivation tree. We obtain this tree by using rule VAR-2.
If rule VAR-2 is used, then $\mathtt{D}_1 = \mathtt{x}$. The procedure is similar, except $\mathtt{x}$ or $\Delta(\mathtt{x})$ on the one hand, and $\mathtt{D}$ on the other, swap positions around $\equiv$ in all relations, and the conclusion on the right-hand side of the implication is obtained through application of VAR-1.

Accordingly, we can conclude that domain equivalence is symmetric.

C.1.3 Transitivity

For domain equivalence to be transitive, for every $\mathtt{D}_1$, $\mathtt{D}_2$ and $\mathtt{D}_3$, the following relation needs to hold:

$$\Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_2 \wedge \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}_3 \implies \Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_3 \tag{C.3}$$

Like symmetry, we can begin by excluding all cases where the left-hand side of C.3 does not hold.

As before, the proof relies on induction on the height of derivation trees for the two equivalences on the left-hand side of C.3. The n used in the induction represents the maximum between the two involved trees. When $n = 0$, the cases to consider are:

1. All three domains are constant domains, meaning the case implication is:

   $$\Delta \vdash \mathtt{B}_1 \equiv \mathtt{B}_2 \wedge \Delta \vdash \mathtt{B}_2 \equiv \mathtt{B}_3 \implies \mathtt{B}_1 \equiv \mathtt{B}_3$$

   and it holds if:

   $$\mathtt{B}_1 = \mathtt{B}_2 \wedge \mathtt{B}_2 = \mathtt{B}_3 \implies \mathtt{B}_1 = \mathtt{B}_3$$

   This is the case for any $\mathtt{B}$, according to the transitivity property of =.

2. All three domains are variables pointing to union domains according to Δ, hence the case implication is:

   $$\Delta \vdash \mathtt{x}_1 \equiv \mathtt{x}_2 \wedge \Delta \vdash \mathtt{x}_2 \equiv \mathtt{x}_3 \implies \mathtt{x}_1 \equiv \mathtt{x}_3$$

   Whether the implication holds depends on the following:

   $$\mathtt{x}_1 = \mathtt{x}_2 \wedge \mathtt{x}_2 = \mathtt{x}_3 \implies \mathtt{x}_1 = \mathtt{x}_3$$

   As with constant domains, the transitivity property of = assures this.

We assume that transitivity holds for all trees of height $j \leq n$. Thus, for $n + 1$, the following cases are considered:

1. We assume that $\mathtt{D}_1$, $\mathtt{D}_2$, and $\mathtt{D}_3$ are not variables. We need only consider products and functions, since the base cases cover basic domains and variables pointing towards unions. For products, if one or both of the equivalences has a tree of height $n + 1$, the case implication is:

   $$\frac{\Delta \vdash \mathtt{D}'_1 \equiv \mathtt{D}'_2 \qquad \Delta \vdash \mathtt{D}''_1 \equiv \mathtt{D}''_2}{\Delta \vdash \mathtt{D}'_1 \bar{\times} \mathtt{D}''_1 \equiv \mathtt{D}'_2 \bar{\times} \mathtt{D}''_2} \wedge \frac{\Delta \vdash \mathtt{D}'_2 \equiv \mathtt{D}'_3 \qquad \Delta \vdash \mathtt{D}''_2 \equiv \mathtt{D}''_3}{\Delta \vdash \mathtt{D}'_2 \bar{\times} \mathtt{D}''_2 \equiv \mathtt{D}'_3 \bar{\times} \mathtt{D}''_3}$$

   $$\implies \frac{\Delta \vdash \mathtt{D}'_1 \equiv \mathtt{D}'_3 \qquad \Delta \vdash \mathtt{D}''_1 \equiv \mathtt{D}''_3}{\Delta \vdash \mathtt{D}'_1 \bar{\times} \mathtt{D}''_1 \equiv \mathtt{D}'_3 \bar{\times} \mathtt{D}''_3}$$

According to the induction hypothesis, the following implications hold:

$$\Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_2' \wedge \Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_3' \implies \Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_3'$$
$$\Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_2'' \wedge \Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_3'' \implies \Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_3''$$

We, thus, know that $\Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_3'$, and $\Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_3''$ hold. Since these form the premises of the tree on the right-hand side, we can obtain the derivation tree by applying rule PRODUCT.

Similar results can be obtained in the function space by applying FUNCTION, instead.

2. We assume that domains $\mathtt{D}_1$ and $\mathtt{D}_3$ are variables. The case implication is:

$$\frac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_2}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_2} \wedge \frac{\Delta \vdash \mathtt{D}_2 \equiv \Delta(\mathtt{x}_3)}{\Delta \vdash \mathtt{D}_2 \equiv \mathtt{x}_3} \implies \frac{\dfrac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \Delta(\mathtt{x}_3)}{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{x}_3}}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{x}_3}$$

Regardless of which tree is of height $n+1$, the trees formed from each premise have a height of at most n. The induction hypothesis states that:

$$\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_2 \wedge \Delta \vdash \mathtt{D}_2 \equiv \Delta(\mathtt{x}_3) \implies \Delta \vdash \Delta(\mathtt{x}_1) \equiv \Delta(\mathtt{x}_3)$$

Knowing the right-hand side in the above holds, by applying rules VAR-1 and VAR-2, we obtain a derivation tree for the right-hand side of the case implication, thus satisfying it for any $\mathtt{D}_2$.

3. When assuming that $\mathtt{D}_1$ and $\mathtt{D}_2$ are variables, the case implication is:

$$\frac{\dfrac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \Delta(\mathtt{x}_2)}{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{x}_2}}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{x}_2} \wedge \frac{\Delta \vdash \Delta(\mathtt{x}_2) \equiv \mathtt{D}_3}{\Delta \vdash \mathtt{x}_2 \equiv \mathtt{D}_3} \implies \frac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_3}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_3}$$

Since the trees formed from premises are at most of height n, then, according to the induction hypothesis, we have that:

$$\Delta \vdash \Delta(\mathtt{x}_1) \equiv \Delta(\mathtt{x}_2) \wedge \Delta \vdash \Delta(\mathtt{x}_2) \equiv \mathtt{D}_3 \implies \Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_3$$

Considering the above, and that the left-hand side premises in the case implication hold, then the right-hand side of the above implication holds. By applying rule VAR-2, we obtain a derivation tree for the conclusion on the right-hand side of the case implication.

The case where $\mathtt{D}_2$ and $\mathtt{D}_3$ are variables observes a similar procedure, concluding through application of rule VAR-1, instead.

4. When assuming that just $\mathtt{D}_1$ is a variable, there are two possible sub-cases, depending on the relative heights of the trees formed from the equivalence involving $\mathtt{x}_1$, and the other equivalence in the conjunction.

   - If the tree involving $\mathtt{x}_1$ is of height $n+1$ and strictly taller, the case implication is:

   $$\frac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_2}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_2} \wedge \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}_3 \implies \frac{\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_3}{\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_3}$$

   The height of the tree formed from premise on the left-hand side is at most n, and the equivalence between $\mathtt{D}_2$ and $\mathtt{D}_3$ is given as having a tree of height at most n. According to the induction hypothesis, the following implication holds:

   $$\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_2 \wedge \Delta \vdash \mathtt{D}_2 \equiv \mathtt{D}_3 \implies \Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_3$$

   It follows that $\Delta \vdash \Delta(\mathtt{x}_1) \equiv \mathtt{D}_3$ holds, and, by applying rule VAR-2, we obtain a derivation tree for the right-hand side of the case implication, thus satisfying it.

   - The second case covers situations where the tree not involving $\mathtt{D}_1$ is of height $n+1$, instead. We can assume that $\mathtt{D}_2$ and $\mathtt{D}_3$ are not variables, since that has already been covered by cases 2. and 3. Hence, the case implication can be one of two forms, depending on whether $\mathtt{D}_2$ and $\mathtt{D}_3$ are product or function domains. When assuming product, the case implication is:

   $$\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_2' \bar{\times} \mathtt{D}_2'' \wedge \frac{\Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_3' \qquad \Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_3''}{\Delta \vdash \mathtt{D}_2' \bar{\times} \mathtt{D}_2'' \equiv \mathtt{D}_3' \bar{\times} \mathtt{D}_3''}$$

   $$\implies \Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_3' \bar{\times} \mathtt{D}_3''$$

   On the left-hand side of the case implication, the height of the trees for $\Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_3'$ and $\Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_3''$ are at most n. Since we know that the left-hand side of the case implication holds, it is expected that $\Delta_\perp(\Delta, \mathtt{x}_1) = \mathtt{D}_1' \bar{\times} \mathtt{D}_1''$. The height of the derivation tree for $\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_2' \bar{\times} \mathtt{D}_2''$ is at most $n+1$. As such, the following equivalence, obtained by repeatedly applying VAR-2 and substituting $\mathtt{x}_1$ for its bound value in Δ:

   $$\Delta \vdash \mathtt{D}_1' \bar{\times} \mathtt{D}_1'' \equiv \mathtt{D}_2' \bar{\times} \mathtt{D}_2''$$

   can only have a derivation tree of at most n. According to the induction hypothesis, the following implications hold:

   $$\Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_2' \wedge \Delta \vdash \mathtt{D}_2' \equiv \mathtt{D}_3' \implies \Delta \vdash \mathtt{D}_1' \equiv \mathtt{D}_3'$$
   $$\Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_2'' \wedge \Delta \vdash \mathtt{D}_2'' \equiv \mathtt{D}_3'' \implies \Delta \vdash \mathtt{D}_1'' \equiv \mathtt{D}_3''$$

Knowing the right-hand side of each implication holds, the following equivalence holds:

$$\Delta \vdash \mathtt{D}_1' \bar{\times} \mathtt{D}_1'' \equiv \mathtt{D}_3' \bar{\times} \mathtt{D}_3''$$

By following in reverse the chain of variable substitutions used by $\Delta_\perp$ to get to the "bedrock" of $\mathtt{x}_1$, we form a chain of premises that ultimately shows that the following equivalence holds:

$$\Delta \vdash \mathtt{x}_1 \equiv \mathtt{D}_3' \bar{\times} \mathtt{D}_3''$$

But this is the conclusion on the right-hand side of the case implication, thus satisfying it.

The case for $\mathtt{D}_3$ can also be derived easily from this one, but involving repeated applications of rule VAR-1, so as to substitute $\mathtt{x}_3$, in the second subcase.

5. The final case involves $\mathtt{D}_2$ being a variable. We can assume that $\mathtt{D}_1$ and $\mathtt{D}_3$ are not variables, since cases 2. and 3. cover these situations. The case implication is, then:

$$\frac{\Delta \vdash \mathtt{D}_1 \equiv \Delta(\mathtt{x}_2)}{\Delta \vdash \mathtt{D}_1 \equiv \mathtt{x}_2} \wedge \frac{\Delta \vdash \Delta(\mathtt{x}_2) \equiv \mathtt{D}_3}{\Delta \vdash \mathtt{x}_2 \equiv \mathtt{D}_3} \implies \Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_3$$

Regardless of which derivation tree from the left-hand side equivalences is of height $n+1$, the heights of the derivation trees of each premise is at most n. According to the induction hypothesis we have that:

$$\Delta \vdash \mathtt{D}_1 \equiv \Delta(\mathtt{x}_2) \wedge \Delta \vdash \Delta(\mathtt{x}_2) \equiv \mathtt{D}_3 \implies \Delta \vdash \mathtt{D}_1 \equiv \mathtt{D}_3$$

Since we know that the left-hand side of the above holds, then the right-hand side holds as well. But this coincides with the right-hand side of the case implication, thus satisfying it.

This concludes the proof for the transitivity of domain equivalence.

C.2 Expression type safety

To prove the correctness of the type system, a type error semantics is introduced, along with some auxiliary notions. These are that of extensions[12] of domains, provided by function ***set***, and of agreement[12] between a semantic environment, e, and a type environment E. In this case, the extension of some domain $\mathtt{D}$ is all ω belonging to the subset of Ω that $\mathtt{D}$ represents, given a well-formed Δ. The environments are in agreement if we have that, $\forall\, \mathtt{x} \in \mathbf{Var}$, where $dom(e) = dom(E)$, then:

$$e(\mathtt{x}) = \omega \implies \omega \in \boldsymbol{set}(\Delta, E(\mathtt{x}))$$

C.2.1 Covering run-time errors

It is important to acknowledge that the error semantics is only defined over errors in types, and not intended to cover diverging computations as a result of **letrec**, or run-time errors resulting from evaluating bottom elements, projecting an expression onto a different tag in the same union, or run-time dependent conditions (such as division by 0). To amend this, some extensions are considered for ***Exp*** for the following sections, when used in defining the error semantics and ***set***, and proving type safety:

- Evaluations of $\bot_{\mathtt{D}}$ are captured, evaluating to themselves.
- Any operations involving a bottom element are changed to instead return a bottom element of the appropriate type, e. g. evaluating $5 = \bot_{\mathsf{int}}$ would be $\bot_{\mathsf{bool}}$, and $\bot_{\mathsf{int} \Rightarrow \mathsf{sym}}(5)$ would be $\bot_{\mathsf{sym}}$.
- Modulo or division by 0 are similarly captured under $\bot_{\mathsf{int}}$.
- Run-time projections on some $\mathtt{t}$, where the projected expression evaluates to $\mathtt{t}'[\omega]$, $\mathtt{t}' \neq \mathtt{t}$, are captured under $\bot_{\mathtt{D}}$, where $\mathtt{t}$ **as** $\mathtt{D}$ is a member of some union of which $\mathtt{t}'$ is also a member.
- Non-terminating behaviors of some expression of type $\mathtt{D}_1 \Rightarrow \mathtt{D}_2$ are captured under $\bot_{\mathtt{D}_2}$.

The definition of Ω is also extended to include a formation rule $\omega ::= \bot_{\mathtt{D}}$, as well as according extensions to the definitions of agreement and of ***set***.

The error semantics is given by the transition system ***ErrExp***, with the signature:

$$\textbf{\textit{ErrExp}} : Env_T \times Env \overset{\Delta}{\vdash} \mathsf{Exp} \Downarrow_{\mathsf{err}}$$

C.2.2 Type extensions

Formally, the function ***set*** has the following signature:

$$\textbf{\textit{set}} : \bar{\Delta} \times \mathsf{Dom} \rightharpoonup \mathcal{P}(\Omega)$$

We assume that the subsets of Ω have been extended with the appropriate bottom element.

The definition of ***set*** over constant domains is as follows:

$$\begin{aligned}
\text{[EMPTY]:} \quad & \boldsymbol{set}(\Delta, \epsilon) = \{\bot_\epsilon\} \\
\text{[INT]:} \quad & \boldsymbol{set}(\Delta, \mathsf{int}) = \mathbb{Z} \\
\text{[SYMBOL]:} \quad & \boldsymbol{set}(\Delta, \mathsf{sym}) = \chi^* \\
\text{[BOOL]:} \quad & \boldsymbol{set}(\Delta, \mathsf{bool}) = \{tt, ff, \bot_{\mathsf{bool}}\} \\
\text{[STRING]:} \quad & \boldsymbol{set}(\Delta, \mathsf{str}) = \Sigma^*
\end{aligned}$$

The results are the subsets of Ω corresponding to the types of basic constant values.

For domain variables, ***set*** finds the extension of its substitution with the "bedrock" environment in the given Δ. It is defined as:

$$\text{[VAR]:} \quad \boldsymbol{set}(\Delta, \mathtt{x}) = \boldsymbol{set}(\Delta, \Delta_\bot(\Delta, \mathtt{x}))$$

For product domains, ***set*** returns all the pairs of values, where each member belongs to the extension of the corresponding sub-domain:

$$\text{[PROD]:} \quad \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\times} \mathtt{D}_2) = \{(\omega_1, \omega_2) \mid \omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}_1), \omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}_2)\}$$

For union domains, the extension includes all the "bare" tags and tag-enclosed values, where, for the latter, the encapsulated value belongs to the extension of the attached domain in the definition of the union.

$$\text{[UNION]:} \quad \boldsymbol{set}(\Delta, \bigcup \overrightarrow{{}^{n} \mathtt{t}_i \ \mathsf{as}\ \mathtt{D}_i}) = \begin{array}{l} \{\mathtt{t}[\omega] \mid \mathtt{t} \in \bigcup\limits_i \{\mathtt{t}_i\},\ \text{s. t. } \omega \in \boldsymbol{set}(\Delta, \mathtt{D}_i)\} \\ \cup\ \{\mathtt{t} \mid \mathtt{t} \in \bigcup\limits_i \{\mathtt{t}_i\},\ \text{s. t. } \mathtt{D}_i = \epsilon\} \end{array}$$

The most involved definition is for function domains. It returns a union between closures (both recursive and non-recursive). In the case of non-recursive closures, it involves all closures such that, given some ω' in $\mathtt{D}_1$, by updating the environment e in the closure, at formal parameter $\mathtt{x}$, inner body $\mathtt{e}$ evalutes to an ω in $\mathtt{D}_2$. The case for recursive closures is similar, but also including an update of e at the declared name with the value of the recursive closure itself.

$$\text{[FUNC]:} \quad \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\to} \mathtt{D}_2) = \begin{array}{l} \{\langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle \mid \text{where if } \omega \in \boldsymbol{set}(\Delta, \mathtt{D}_2) \text{ and } \omega' \in \boldsymbol{set}(\Delta, \mathtt{D}_1) \\ \text{then } [\mathtt{x} \mapsto \omega']e \vdash \mathtt{e} \Downarrow_{\boldsymbol{Exp}} \omega\} \cup \\ \{\langle\!\langle e, \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e} \rangle\!\rangle \mid \text{where } \omega' \in \boldsymbol{set}(\Delta, \mathtt{D}_1) \text{ and } \omega \in \boldsymbol{set}(\Delta, \mathtt{D}_2) \\ \text{then } [\mathtt{x}' \mapsto \omega', \mathtt{x} \mapsto \langle\!\langle e, \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e} \rangle\!\rangle]e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\boldsymbol{Exp}} \omega\} \end{array}$$

The following sections present formally the rules of ***ErrExp***.

C.2.3 Variables and constants

$$\text{[VAR-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{x} \Downarrow_{\mathbf{err}} \qquad \begin{array}{l}\text{if } e(\mathtt{x}) \text{ is undefined}\\ \text{or } e(\mathtt{x}) \notin \boldsymbol{set}(\Delta, E(\mathtt{x}))\end{array}$$

C.2.4 Arithmetic expressions

$$\text{[ARITHMETIC-BIN-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{e}_1\ \mathbf{op}\ \mathtt{e}_2 \Downarrow_{\mathbf{err}} \quad \forall\ \mathbf{op} \in \sigma_A$$
$$\text{if } E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}} \text{, or } E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$$
$$\text{or } e \vdash \mathtt{e}_1 \Downarrow_{\mathit{Exp}} \omega_1 \text{ and } e \vdash \mathtt{e}_2 \Downarrow_{\mathit{Exp}} \omega_2$$
$$\text{but } \omega_1, \omega_2 \notin \boldsymbol{set}(\Delta, \mathbf{int})$$

$$\text{[INV-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{-e} \Downarrow_{\mathbf{err}}$$
$$\text{if } E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\mathbf{err}}$$
$$\text{or } e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \omega \text{ but } \omega \notin \boldsymbol{set}(\Delta, \mathbf{int})$$

C.2.5 Relational expressions

$$\text{[RELATIONAL-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{e}_1\ \mathbf{op}\ \mathtt{e}_2 \Downarrow_{\mathbf{err}} \quad \forall\ op \in \sigma_R$$
$$\text{if } E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}} \text{, or } E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$$
$$\text{or } e \vdash \mathtt{e}_1 \Downarrow_{\mathit{Exp}} \omega_1 \text{ and } e \vdash \mathtt{e}_2 \Downarrow_{\mathit{Exp}} \omega_2$$
$$\text{but } \omega_1, \omega_2 \notin \boldsymbol{set}(\Delta, \mathtt{B})$$

C.2.6 Boolean expressions

$$\text{[BOOLEAN-BIN-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{e}_1\ \mathbf{op}\ \mathtt{e}_2 \Downarrow_{\mathbf{err}} \quad \forall \mathbf{op} \in \sigma_B$$
$$\text{if } E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}} \text{, or } E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$$
$$\text{or } e \vdash \mathtt{e}_1 \Downarrow_{\mathit{Exp}} \omega_1, \text{ and } e \vdash \mathtt{e}_2 \Downarrow_{\mathit{Exp}} \omega_2$$
$$\text{but } \omega_1, \omega_2 \notin \boldsymbol{set}(\Delta, \mathbf{bool})$$

$$\text{[NEG-ERR]:}\quad E, e \overset{\Delta}{\vdash} \mathtt{!e} \Downarrow_{\mathbf{err}}$$
$$\text{if } E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\mathbf{err}} \text{, or } e\mathtt{e} \Downarrow_{\mathit{Exp}} \omega \text{ but } \omega \notin \boldsymbol{set}(\Delta, \mathbf{bool})$$

C.2.7 String concatenation

[CONCAT-ERR]: $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \mathbin{++} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$

if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}}$, or $E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$
or $e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega_1$ and $e \vdash \mathtt{e}_2 \Downarrow_{Exp} \omega_2$
but $\omega_1, \omega_2 \notin \boldsymbol{set}(\Delta, \mathbf{str})$

C.2.8 Functional constructs

[LAMBDA-ERR]: $E, e \overset{\Delta}{\vdash} \mathbf{lam}\ \mathtt{x} : \mathtt{D}\ .\ \mathtt{e} \Downarrow_{\mathbf{err}}$

if $[\mathtt{x} \mapsto \mathtt{D}]E, [\mathtt{x} \mapsto \omega]e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\mathbf{err}}$, s. t. $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$

[APP-ERR]: $E, e \overset{\Delta}{\vdash} \mathbf{app}\ \mathtt{e}_1\ \mathtt{e}_2 \Downarrow_{\mathbf{err}}$

if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}}$, or $E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$
or $e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega_1$ but $\omega_1 \notin \boldsymbol{set}(\Delta, \mathtt{D}_1 \vec{\to} \mathtt{D}_2)$
or $e \vdash \mathtt{e}_2 \Downarrow_{Exp} \omega_2$ and $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}_1 \vec{\to} \mathtt{D}_2)$
but $\omega_2 \notin \boldsymbol{set}(\Delta, \mathtt{D}_1)$

[LET-ERR]: $E \overset{\Delta}{\vdash} \mathbf{let}\ \mathtt{x} := \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2 \Downarrow_{\mathbf{err}}$

if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}}$
or if $e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega$ where $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$
then $[\mathtt{x} \mapsto \mathtt{D}]E, [\mathtt{x} \mapsto \omega]e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$

[LETREC-ERR]: $E, e \overset{\Delta}{\vdash} \mathbf{letrec}\ \mathtt{x}_1 : \mathtt{D} := \mathtt{x}_2\ .\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2 \Downarrow_{\mathbf{err}}$
if $\Delta \vdash \mathtt{D} \not\equiv \mathtt{D}_1 \vec{\to} \mathtt{D}_2$
or $\Delta \vdash \mathtt{D} \equiv \mathtt{D}_1 \vec{\to} \mathtt{D}_2$
but, for $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D})$ and $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}_1)$
$[\mathtt{x}_1 \mapsto \mathtt{D}, \mathtt{x}_2 \mapsto \mathtt{D}_1]E, [\mathtt{x} \mapsto \omega_1,\ \mathtt{x}_2 \mapsto \omega_2]e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}}$
or $[\mathtt{x}_1 \mapsto \mathtt{D}]E, [\mathtt{x}_1 \mapsto \omega_1]e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$
or $[\mathtt{x}_1 \mapsto \omega_1,\ \mathtt{x}_2 \mapsto \omega_2]e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega,$ but $\omega \notin \boldsymbol{set}(\Delta, \mathtt{D}_2)$

[IF-ERR]: $E, e \overset{\Delta}{\vdash} \textsf{if } \mathtt{e}_1 \textsf{ then } \mathtt{e}_2 \textsf{ else } \mathtt{e}_3 \Downarrow_{\textbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\textbf{err}}$, or $E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\textbf{err}}$, or $E, e \overset{\Delta}{\vdash} \mathtt{e}_3 \Downarrow_{\textbf{err}}$
or $e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega_1$ but $\omega_1 \notin \boldsymbol{set}(\Delta, \textsf{bool})$
or $e \vdash \mathtt{e}_2 \Downarrow_{Exp} \omega_2$, s. t. $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D})$ for some $\mathtt{D}$
but $e \vdash \mathtt{e}_3 \Downarrow_{Exp} \omega_3$ and $\omega_3 \notin \boldsymbol{set}(\Delta, \mathtt{D})$

[UPDATE-ERR]: $E, e \overset{\Delta}{\vdash} [\ \mathtt{e}_1 \textsf{ to } \mathtt{e}_2\ ]\ \mathtt{e}_3 \Downarrow_{\textbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\textbf{err}}$, or $E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\textbf{err}}$, or $E.e \overset{\Delta}{\vdash} \mathtt{e}_3 \Downarrow_{\textbf{err}}$
or $e \vdash \mathtt{e}_1 \Downarrow_{Exp} \omega_1$, but $\omega_1 \notin \boldsymbol{set}(\Delta, \mathtt{B})$, where $\mathtt{B} \neq \epsilon$
or $e \vdash \mathtt{e}_2 \Downarrow_{Exp} \omega_2$ s. t. $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D})$
and $e \vdash \mathtt{e}_3 \Downarrow_{Exp} \omega_3$ but $\omega_3 \notin \boldsymbol{set}(\Delta, \mathtt{B} \rightarrow \mathtt{D})$

C.2.9 Operations on tags

[INJECT-1-ERR]: $E \overset{\Delta}{\vdash} \mathtt{t}\ [\ \epsilon\ ] \Downarrow_{\textbf{err}}$
if $\nexists\ \mathtt{x}$ such that $\exists_{\mathtt{t}}(\mathtt{t}, \epsilon, \Delta(\mathtt{x}))$

[INJECT-2-ERR]: $E, e \overset{\Delta}{\vdash} \mathtt{t}\ [\ \mathtt{e}\ ] \Downarrow_{\textbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\textbf{err}}$ or $e \vdash \mathtt{e} \Downarrow_{Exp} \omega$ and $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$
but $\nexists\ \mathtt{x}$ s. t. $\exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}))$

[PROJECT-ERR]: $E, e \overset{\Delta}{\vdash} \mathtt{e} \gg \mathtt{t} \Downarrow_{\textbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\textbf{err}}$, or where $e \vdash \mathtt{e} \Downarrow_{Exp} \omega$
but $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$ where $\mathtt{D} \notin \textbf{Dom}_{\bar{\cup}}$
or $\mathtt{D} \in \textbf{Dom}_{\bar{\cup}}$ but $\nexists\ \mathtt{D}' \neq \epsilon$ s. t. $\exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}', \mathtt{D}))$

[IS-TAG-ERR]: $E, e \overset{\Delta}{\vdash} \mathtt{e} \textsf{ is } \mathtt{t} \Downarrow_{\textbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\textbf{err}}$, or $\nexists\ \mathtt{x}^{\mathtt{D}}$ s. t. $\exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}^{\mathtt{D}}))$ for some $\mathtt{D}$
or $e \vdash \mathtt{e} \Downarrow_{Exp} \omega$ but $\omega \notin \boldsymbol{set}(\Delta, \mathtt{x}^{\mathtt{D}})$ s. t. $\exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}^{\mathtt{D}}))$

C.2.10 Operations on pairs

[PAIR-ERR]: $E.e \overset{\Delta}{\vdash} (\mathtt{e}_1,\ \mathtt{e}_2) \Downarrow_{\mathbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e}_1 \Downarrow_{\mathbf{err}}$ or $E, e \overset{\Delta}{\vdash} \mathtt{e}_2 \Downarrow_{\mathbf{err}}$

[HEAD-ERR]: $E, e \overset{\Delta}{\vdash} \mathbf{head}\ \mathtt{e} \Downarrow_{\mathbf{err}}$
if $E \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\mathbf{err}}$
or $e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \omega$ but $\omega \notin \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\times} \mathtt{D}_2)$

[TAIL-ERR]: $E, e \overset{\Delta}{\vdash} \mathbf{tail}\ \mathtt{e} \Downarrow_{\mathbf{err}}$
if $E, e \overset{\Delta}{\vdash} \mathtt{e} \Downarrow_{\mathbf{err}}$
or $e \vdash \mathtt{e} \Downarrow_{\mathit{Exp}} \omega$ but $\omega \notin \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\times} \mathtt{D}_2)$

C.2.11 Proof

Theorem C.2.1 (Expression type safety) *Whenever the semantic and type environments agree, we have that if* $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}$ *then* $E, e \overset{\Delta}{\vdash} \mathtt{e} \not\Downarrow_{\mathbf{err}}$ *(error prevention), and* $e \vdash \mathtt{e} \Downarrow \omega$*, where* $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$ *(type preservation). Expressed formally:*

$$\begin{array}{l} dom(e) = dom(E) \\ \forall\, \mathtt{x} \in dom(e), \\ e(\mathtt{x}) \in \boldsymbol{set}(\Delta, E(\mathtt{x})) \end{array} \quad \vdash \quad E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D} \implies \begin{array}{l} E, e \overset{\Delta}{\vdash} \mathtt{e} \not\Downarrow_{\mathbf{err}} \wedge \\ e \vdash \mathtt{e} \Downarrow \omega,\ s.\ t.\ \omega \in \boldsymbol{set}(\Delta, \mathtt{D}) \end{array}$$

The proof involves induction over the derivation trees of ***TExp***. For a tree of height $n = 0$, we have the following cases, involving the axioms of ***TExp***:

1. CONST:

$$E \overset{\Delta}{\vdash} \mathtt{c} :: \mathscr{T}(\mathtt{c})$$

Error prevention - the proof is trivial, considering no error transition is defined for evaluations of constants.

Type preservation - according to the definitions of $\mathscr{V}$, $\mathscr{T}$, and ***set***, it follows that the values returned for each constant literal belong to the correct extension. Regarding $\bot_{\mathtt{D}}$, we extend $\mathscr{V}$ as follows:

$$\mathscr{V}(\bot_{\mathtt{D}}) = \bot_{\mathtt{D}}$$

Considering that $\mathscr{T}(\bot_{\mathtt{D}}) = \mathtt{D}$, and that, according to the definition extensions introduced in section C.2.1, we have that $\bot_{\mathtt{D}} \in \textbf{\textit{set}}(\Delta, \mathtt{D})$, it is clear to see the type is preserved.

2. VAR:

$$E \overset{\Delta}{\vdash} \mathtt{x} :: E(\mathtt{x}) \quad \text{if } E(\mathtt{x}) \text{ is defined}$$

Error prevention - an error transition is defined if x is not defined in e. But according to the assumption of agreement, we have that if e(x is undefined, then $E(\mathtt{x})$ is also undefined. But the type rule clearly disallows this, thus preventing an error transition.

Type preservation - according to the agreement assumption, we have that when $e(\mathtt{x}) = \omega$, then $\omega \in \textbf{\textit{set}}(\Delta, E(\mathtt{x}))$.

3. INJECT-1:

$$E \overset{\Delta}{\vdash} \mathtt{t}\ [\ \epsilon\ ] :: \mathtt{x} \quad \text{if } \exists_{\mathtt{t}}(\mathtt{t}, \epsilon, \Delta(\mathtt{x}))$$

Error prevention - the situation covered by this case is "bare" tag injection. The type system clearly disallows usages of tags which have no associated parent union in the given Δ, or of which the attached domain is not empty, preventing such a run-time error.

Type preservation - when a union with "bare" tag t exists, a well-formed specification also implies that the union has a name $\mathtt{x}^{\mathtt{D}}$. Thus, the evaluation would result in a value of the form $\mathtt{t} \in \textbf{\textit{set}}(\Delta, \mathtt{x}^{\mathtt{D}})$, ensuring type preservation.

We state the induction hypothesis, according to which trees of height $j \leq n$ satisfy type safety. We must now prove for trees of height $n+1$ that type safety is preserved. For brevity, we can start by stating that an error transition is not defined over any sub-term as long as there continues to be an agreement between the type and semantic environments, in accordance with the induction hypothesis. Considering this, the cases to prove are:

1. ARITHMETIC-BIN:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \textbf{int} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \textbf{int}}{E \overset{\Delta}{\vdash} \mathtt{e}_1\ \textbf{op}\ \mathtt{e}_2 :: \textbf{int}} \quad \forall\ \textbf{op} \in \sigma_A$$

Error prevention - an error transition is defined when $e \vdash \mathtt{e}_1 \Downarrow \omega_1$, but $\omega_1 \notin \textbf{\textit{set}}(\Delta, \textbf{int})$, or $e \vdash \mathtt{e}_2 \Downarrow \omega_2$, but $\omega_2 \notin \textbf{\textit{set}}(\Delta, \textbf{int})$. But, according to the induction hypothesis we have that $E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \textbf{int} \implies e \vdash \mathtt{e}_1 \Downarrow \omega$ such

that $\omega \in \textbf{\textit{set}}(\Delta, \textbf{int})$, and similarly for $\mathtt{e}_2$, preventing a type-unsafe arithmetic expression from executing.

Type preservation - according to the induction hypothesis, if the sub-expressions are well-typed, then $\omega_1,\ \omega_2 \in \textbf{\textit{set}}(\Delta, \textbf{int})$. It immediately follows that $op(\omega_1, \omega_2) \in \textbf{\textit{set}}(\Delta, \textbf{int})$ for any arithmetic operation, satisfying the type rule and preserving the type.

2. The following cases are very similar to the above: INV, RELATIONAL, CONCAT, BOOLEAN-BIN and NEG, where only the operator, the basic domain and the number of sub-terms involved (between unary and binary operations) are different.

3. LAMBDA:

$$\frac{[\mathtt{x} \mapsto \mathtt{D}_1]E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_2}{E \overset{\Delta}{\vdash} \textbf{lam}\ \mathtt{x} : \mathtt{D}_1\ .\ \mathtt{e} :: \mathtt{D}_1 \to \mathtt{D}_2}$$

Error prevention - in this case, an error can occur in the semantics if the body of the expression produces an error with E updated at $\mathtt{x}$ with $\mathtt{D}_1$, and e with some arbitrary $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{D}_1)$.

Considering an application to an arbitrary $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{D}_1)$, e is updated at $\mathtt{x}$ with ω, hence agreement between environments is preserved. Thus, we can safely apply the induction hypothesis, and rule out run-time type errors for λ-expressions with well-typed bodies.

Type preservation - the axiom applied during evaluation would be:

$$e \vdash \textbf{lam}\ \mathtt{x} : \mathtt{D}_1\ .\ \mathtt{e} \Downarrow \langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle$$

Given $\omega' \in \textbf{\textit{set}}(\Delta, \mathtt{D}_1)$, where e is updated at $\mathtt{x}$ with ω', since agreement is preserved, according to the induction hypothesis, we have that $[\mathtt{x} \mapsto \mathtt{D}_1]E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_2 \implies [\mathtt{x} \mapsto \omega']e \vdash \mathtt{e} \Downarrow \omega$, such that $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{D}_2)$. From e, $\mathtt{x}$, and $\mathtt{e}$, we can therefore construct closure $\langle\!\langle e, \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle \in \textbf{\textit{set}}(\Delta, \mathtt{D}_1 \to \mathtt{D}_2)$. The type is successfully preserved by the type judgement.

4. APP:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}' \qquad E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}' \to \mathtt{D}}{E \overset{\Delta}{\vdash} \textbf{app}\ \mathtt{e}_1\ \mathtt{e}_2 :: \mathtt{D}}$$

Error prevention - according to the induction hypothesis, we have that $E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}' \implies e \vdash \mathtt{e}_2 \Downarrow \omega_2$, such that $\omega_2 \in \textbf{\textit{set}}(\Delta, \mathtt{D}')$ and, $E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}' \to \mathtt{D} \implies$

$e \vdash \mathtt{e}_1 \Downarrow \omega_1$ such that $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}' \to \mathtt{D})$. If the type judgement is satisfied, then none of the conditions for an error semantics apply.

Type preservation - as above, we apply the induction hypothesis, in which case, for well-typed $\mathtt{e}_1$, we have that $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}' \to \mathtt{D})$. Consider an arbitrary $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}')$. According to the definition of ***set*** over function domains, there are two possible forms for ω_1:

- In the first case, $\omega_1 = \langle\!\langle e', \lambda \mathtt{x}.\mathtt{e}' \rangle\!\rangle$ and applying it to ω_2 will lead to the following evaluation by way of rule APP-1:

  $$[\mathtt{x} \mapsto \omega_2]e' \vdash \mathtt{e}' \Downarrow \omega$$

  But, since we know that $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}' \to \mathtt{D})$, then, according to the definition of ***set***, we have that $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$, preserving the type.

- In the second case, $\omega_1 = \langle\!\langle e', \mathtt{x}, \lambda \mathtt{x}'.\mathtt{e}' \rangle\!\rangle$ and applying it to ω_2 will lead to the following evaluation by way of rule APP-2:

  $$[\mathtt{x} \mapsto \omega_1, \mathtt{x}' \mapsto \omega_2]e' \vdash \mathtt{e}' \Downarrow \omega$$

  Considering that we know $\omega_1 \in \boldsymbol{set}(\mathtt{D}' \to \mathtt{D})$, then we have that $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$, again preserving the type.

5. LET:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D} \qquad [\mathtt{x} \mapsto \mathtt{D}]E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}'}{E \overset{\Delta}{\vdash} \textsf{let } \mathtt{x} := \mathtt{e}_1 \textsf{ in } \mathtt{e}_2 :: \mathtt{D}'}$$

Error prevention - for $\mathtt{e}_1$, we can safely apply the induction hypothesis for type safety. We have that a type safe $\mathtt{e}_1$ will evaluate to some $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$. In such a case, considering the updates to both environments with respect to $\mathtt{x}$, agreement is preserved between environments when type checking of $\mathtt{e}_2$. We can apply the induction hypothesis to rule out an error transition over $\mathtt{e}_2$, thus also preventing it for the let expression.

Type preservation - Considering the aforementioned agreement preservation, we can safely apply the induction hypothesis to obtain that, if $[\mathtt{x} \mapsto \mathtt{D}]E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}'$, then $[\mathtt{x} \mapsto \omega]e \vdash \mathtt{e}_2 \Downarrow \omega'$, such that $\omega' \in \boldsymbol{set}(\Delta, \mathtt{D}')$. Since ω' is in the conclusion of the LET rule when evaluating, and that $\mathtt{D}'$ is used in the conclusion of the type judgement, then the type is preserved.

6. LETREC:

$$\frac{[\mathtt{x}_1 \mapsto \mathtt{D}_1 \to \mathtt{D}_2,\ \mathtt{x}_2 \mapsto \mathtt{D}_1]E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}_2 \qquad [\mathtt{x}_1 \mapsto \mathtt{D}_1 \to \mathtt{D}_2]E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}'}{E \overset{\Delta}{\vdash} \textsf{letrec } \mathtt{x}_1 : \mathtt{D}_1 \to \mathtt{D}_2 := \mathtt{x}_2 \,.\, \mathtt{e}_1 \textsf{ in } \mathtt{e}_2 :: \mathtt{D}'}$$

Error prevention - an error transition is defined if the domain annotation of the letrec is not a function domain, but since the type rule disallows this, then an error transition is prevented on these grounds. The second case is when we have $\mathtt{D}_1 \overset{-}{\to} \mathtt{D}_2$, and, for $e' = [\mathtt{x}_1 \mapsto \omega_1, \mathtt{x}_2 \mapsto \omega_2]e$, where $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}_1 \overset{-}{\to} \mathtt{D}_2)$ and $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}_1)$, respectively, then $e' \vdash \mathtt{e}_1 \Downarrow \omega$, such that $\omega \notin \boldsymbol{set}(\Delta, \mathtt{D}_2)$. However, since agreement is preserved, by applying the induction hypothesis, if we have a well-typed $\mathtt{e}_1$, then $\omega \in \boldsymbol{set}(\Delta, \mathtt{D}_2)$. The only remaining case would be an error transition defined over $\mathtt{e}_2$. We observe that, since $\mathtt{x}_1$ would be bound to ω_1 at run-time, , and $\mathtt{D}_1 \overset{-}{\to} \mathtt{D}_2$ during type checking, the agreement is preserved and we can apply the induction hypothesis to rule it out.

Type preservation - type preservation can be proven by using the above proven preservation of agreement and following the same procedure as LET.

7. IF:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \textbf{bool} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_3 :: \mathtt{D}}{E \overset{\Delta}{\vdash} \textbf{if } \mathtt{e}_1 \textbf{ then } \mathtt{e}_2 \textbf{ else } \mathtt{e}_3 :: \mathtt{D}}$$

Error prevention - we have that an error transition is defined if the guard condition is not a boolean, or if the two branches belong to different extensions. In the first case, we can apply the induction hypothesis, showing that $E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \textbf{bool} \implies e \vdash \mathtt{e}_1 \Downarrow \omega_1$, such that $\omega_1 \in \boldsymbol{set}(\Delta, \textbf{bool})$. In the second, we have that, $E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D} \implies e \vdash \mathtt{e}_2 \Downarrow \omega_2$ such that $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D})$, and $E \overset{\Delta}{\vdash} \mathtt{e}_3 :: \mathtt{D} \implies e \vdash \mathtt{e}_2 \Downarrow \omega_3$ such that $\omega_3 \in \boldsymbol{set}(\Delta, \mathtt{D})$. For the type judgement to succeed, ω_2 and ω_3 must belong to the same extension, but this would prevent an error transition at run-time.

Type preservation - the type judgement concludes with type D, and, as previously shown, the values from evaluating the branches belong to the extension of D. This concludes type preservation.

8. UPDATE:

$$\frac{\begin{array}{c} E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{B} \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D} \\ E \overset{\Delta}{\vdash} \mathtt{e}_3 :: \mathtt{B} \overset{-}{\to} \mathtt{D} \end{array}}{E \overset{\Delta}{\vdash} [\ \mathtt{e}_1 \textbf{ to } \mathtt{e}_2\ ]\ \mathtt{e}_3 :: \mathtt{B} \overset{-}{\to} \mathtt{D}} \quad \text{if } \mathtt{B} \neq \epsilon$$

Error prevention - we can apply the induction hypothesis on $\mathtt{e}_1$, $\mathtt{e}_2$ and $\mathtt{e}_3$, obtaining $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{B})$, $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D})$, and $\omega_3 \in \boldsymbol{set}(\Delta, \mathtt{B} \overset{-}{\to} \mathtt{D})$ if they are well-typed, thus preventing an error transition.

Type preservation - to ensure type preservation, the evaluation of an update must be some $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{B} \rightarrow \mathtt{D})$. Depending on the value of ω_3, there are two cases to analyze:

- In the first case, $\omega_3 = \langle\!\langle e', \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle$. Therefore, the evaluation result would be $\omega = \langle\!\langle e', \lambda \mathtt{x}.\textbf{if } \mathtt{x} = \{e, \mathtt{e}_1\} \textbf{ then } \{e, \mathtt{e}_2\} \textbf{ else } \mathtt{e} \rangle\!\rangle$. To show that $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{B} \rightarrow \mathtt{D})$, we need to analyze the result of evaluating the body of the enclosed lambda expression and show that the result is $\omega'' \in \textbf{\textit{set}}(\Delta, \mathtt{D})$. For any $\omega' \in \textbf{\textit{set}}(\Delta, \mathtt{B})$, and $e'' = [\mathtt{x} \mapsto \omega']e'$, we can construct derivation trees, the first for when the guard is true, the other for when it is false:

$$\frac{\dfrac{e'' \vdash \mathtt{x} \Downarrow \omega' \quad e \vdash \mathtt{e}_1 \Downarrow \omega_1}{e'' \vdash \mathtt{x} = \{e, \mathtt{e}_1\} \Downarrow tt} \qquad \dfrac{e \vdash \mathtt{e}_2 \Downarrow \omega_2}{e'' \vdash \{e, \mathtt{e}_2\} \Downarrow \omega_2}}{e'' \vdash \textbf{if } \mathtt{x} = \{e, \mathtt{e}_1\} \textbf{ then } \{e, \mathtt{e}_2\} \textbf{ else } \mathtt{e} \Downarrow \omega_2}$$

$$\frac{\dfrac{e'' \vdash \mathtt{x} \Downarrow \omega' \quad e \vdash \mathtt{e}_1 \Downarrow \omega_1}{e'' \vdash \mathtt{x} = \{e, \mathtt{e}_1\} \Downarrow ff} \qquad e'' \vdash \mathtt{e} \Downarrow \omega''}{e'' \vdash \textbf{if } \mathtt{x} = \{e, \mathtt{e}_1\} \textbf{ then } \{e, \mathtt{e}_2\} \textbf{ else } \mathtt{e} \Downarrow \omega''}$$

  On the **else** branch, the original body of ω_3 is evaluated, and, since we know $\omega_3 \in \textbf{\textit{set}}(\Delta, \mathtt{B} \rightarrow \mathtt{D})$, then $\omega'' \in \textbf{\textit{set}}(\Delta, \mathtt{D})$. On the **then** branch, we instead evaluate closure $\{e, \mathtt{e}_2\}$. However, as previously established through the induction hypothesis, for a well-typed $\mathtt{e}_2$ we have that $e \vdash \mathtt{e}_2 \Downarrow \omega_2$, and $\omega_2 \in \mathtt{D}$. In both cases, evaluating the if statement results in a value in the extension of $\mathtt{D}$. This implies that $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{B} \rightarrow \mathtt{D})$.

- In the second case, $\omega_3 = \langle\!\langle e', \mathtt{x}', \lambda \mathtt{x}.\mathtt{e} \rangle\!\rangle$. The evaluation result would be $\omega = \langle\!\langle e', \mathtt{x}', \lambda \mathtt{x}.\textbf{if } \mathtt{x} = \{e, \mathtt{e}_1\} \textbf{ then } \{e, \mathtt{e}_2\} \textbf{ else } \mathtt{e} \rangle\!\rangle$. The procedure follows similar steps as the previous one, evaluating the body of the closure for arbitrary inputs, except we update environment e' as follows:

$$e'' = [\mathtt{x} \mapsto \omega',\ \mathtt{x}' \mapsto \langle\!\langle e', \mathtt{x}', \lambda \mathtt{x}.\textbf{if } \mathtt{x} = \{e, \mathtt{e}_1\} \textbf{ then } \{e, \mathtt{e}_2\} \textbf{ else } \mathtt{e} \rangle\!\rangle]e'$$

  Except the updates at e'', the constructed derivation trees are identical to the non-recursive closure, and similar results follow about the preservation of types.

9. INJECT-2:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}}{E \overset{\Delta}{\vdash} \mathtt{t}\ [\ \mathtt{e}\ ] :: \mathtt{x}} \quad \text{if } \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}))$$

   Error prevention - an error transition is defined whenever $e \vdash \mathtt{e} \Downarrow \omega$, where $\omega \in \textbf{\textit{set}}(\Delta, \mathtt{D}$, but there does not exist a variable bound to a union in the domain environment such that tag $\mathtt{t}$ encapsulates $\mathtt{D}$. By virtue of our induction

hypothesis we have that $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D} \implies e \vdash \mathtt{e} \Downarrow \omega$, such that $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$. If the side condition is also satisfied, then there is no error transition.

Type preservation - whenever e is well-typed, we have that tag injection yields some value of the form $\mathtt{t}[\omega]$, which belongs to $\boldsymbol{set}(\Delta, \mathtt{x})$, where x is bound to the union in Δ which has tag t.

10. PROJECT:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x}}{E \overset{\Delta}{\vdash} \mathtt{e} \gg \mathtt{t} :: \mathtt{D}} \quad \text{if } \exists\ \mathtt{D}, \text{ s.t. } \mathtt{D} \neq \epsilon \wedge \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x}))$$

Error prevention - an error transition is defined whenever the inner expression does not belong to the extension of some union, or if it does, but said union has no tag t attached to some non-empty D. By applying the induction hypothesis, we have that $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x} \implies e \vdash \mathtt{e} \Downarrow \omega$, such that $\omega \in \boldsymbol{set}(\Delta, \mathtt{x})$, and, if the other conditions are also satisfied, no error transition can occur.

Type preservation - a well-typed e will evaluate to some $\omega' \in \boldsymbol{set}(\Delta, \mathtt{x})$, which is of the form $\mathtt{t}'$ or $\mathtt{t}'[\omega]$. In the first case, then $\mathtt{t} \neq \mathtt{t}'$, since projection on a "bare" tag would have been statically prevented by the type checker. It would evaluate to $\bot_{\mathtt{D}} \in \boldsymbol{set}(\Delta, \mathtt{D})$, satisfying type preservation. If $\mathtt{t} = \mathtt{t}'$, then according to the induction hypothesis we have that $\omega \in \boldsymbol{set}(\Delta, \mathtt{D})$, otherwise, we have $\bot_{\mathtt{D}} \in \boldsymbol{set}(\Delta, \mathtt{D})$. Both preserve the type.

11. IS-TAG:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x}}{E \overset{\Delta}{\vdash} \mathtt{e}\ \textbf{is}\ \mathtt{t} :: \textbf{bool}} \quad \begin{array}{l}\text{if } \Delta(\mathtt{x}) \in \mathbf{Dom}_{\bar{\bigcup}} \\ \text{and } \exists_{\mathtt{t}}(\mathtt{t}, \mathtt{D}, \Delta(\mathtt{x})) \text{ for some } \mathtt{D}\end{array}$$

Error prevention - an error transition is defined whenever there doesn't exist any x in $dom(\Delta)$, such that $\Delta(\mathtt{x}) \in \mathbf{Dom}_{\bar{\bigcup}}$ and t is a member of it, or e evaluates to ω not in the extension of x. By applying the induction hypothesis, we have that $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{x} \implies e \vdash \mathtt{e} \Downarrow \omega$ such that $\omega \in \boldsymbol{set}(\Delta, \mathtt{x})$. If the type judgement succeeds according to the side conditions as well, then there is no error transition.

Type preservation - when checking the tag of a well-typed expression, the succesful type judgement yields **bool**. Therefore, at evaluation time, the value should be in $\boldsymbol{set}(\Delta, \textbf{bool})$, which is the case, being the result of $\mathtt{t} = \mathtt{t}'$.

12. PAIR:

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}_1 \qquad E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}_2}{E \overset{\Delta}{\vdash} (\mathtt{e}_1,\ \mathtt{e}_2) :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2}$$

Error prevention - an error transition is defined only when the sub-terms have an error transition themselves. But, by applying the induction hypothesis, we have that well-typed sub-terms will prevent an error transition on pair formation.

Type preservation - we have that $E \overset{\Delta}{\vdash} \mathtt{e}_1 :: \mathtt{D}_1 \implies e \vdash \mathtt{e}_1 \Downarrow \omega_1$ such that $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}_1)$, and $E \overset{\Delta}{\vdash} \mathtt{e}_2 :: \mathtt{D}_2 \implies e \vdash \mathtt{e}_2 \Downarrow \omega_2$, such that $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}_2)$. The result of the evaluating a pair formation is (ω_1, ω_2) which belongs to $\boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\times} \mathtt{D}_2)$.

13. HEAD and TAIL: We will follow both in parallel, using HEAD as a general example.

$$\frac{E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2}{E \overset{\Delta}{\vdash} \textsf{head}\ \mathtt{e} :: \mathtt{D}_1}$$

Error prevention - an error transition is defined if e is not in the extension of some product type. However, by applying the induction hypothesis, we have that $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \implies e \vdash \mathtt{e} \Downarrow \omega$, such that $\omega \in \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\times} \mathtt{D}_2)$, in which case there is no error transition.

Type preservation - we have that $E \overset{\Delta}{\vdash} \mathtt{e} :: \mathtt{D}_1 \bar{\times} \mathtt{D}_2 \implies e \vdash \mathtt{e} \Downarrow \omega'$ such that $\omega' \in \boldsymbol{set}(\Delta, \mathtt{D}_1 \bar{\to} \mathtt{D}_2)$, in which case $\omega' = (\omega_1, \omega_2)$ and applying **head** yields $\omega_1 \in \boldsymbol{set}(\Delta, \mathtt{D}_1)$, preserving the type. In the case of TAIL, we have $\omega_2 \in \boldsymbol{set}(\Delta, \mathtt{D}_2)$, again preserving the type.

With all of these cases covered, we can conclude type safety for expressions.

Appendix D

Sample specifications

D.1 Bims

Below is an implementation of the specifications for the simple language ***Bims***[12] (**B**asic **im**perative **s**tatements). The language involves simple arithmetic expressions and statements.

```
domain State = Symbol -> Int;

syntax Aexp = add of Aexp * Aexp
  | mul of Aexp * Aexp
  | sub of Aexp * Aexp
  | aparens of Aexp
  | aconst of Int
  | var of Symbol;

syntax Bexp = eq of Aexp * Aexp
  | lt of Aexp * Aexp
  | not of Bexp
  | and of Bexp * Bexp
  | bparens of Bexp
  | bconst of Bool;

syntax Stm = skip
  | ass of Symbol * Aexp
  | comp of Stm * Stm
  | ifs of Bexp * Stm * Stm
  | while of Bexp * Stm;
```

```
system Aexp : State |- Aexp ==> Int =
  [[ VAR ]]: s |- var[x] ==> s(x);

  [[ CONST ]]: s |- aconst[n] ==> n;

  [[ ADD ]]: s |- add[a1, a2] ==> v1 + v2 \\
    s |- a1 ==> v1,
    s |- a2 ==> v2;

  [[ MUL ]]: s |- mul[a1, a2] ==> v1 * v2 \\
    s |- a1 ==> v1,
    s |- a2 ==> v2;

  [[ SUB ]]: s |- sub[a1, a2] ==> v1 - v2 \\
    s |- a1 ==> v1,
    s |- a2 ==> v2;

  [[ PARENS ]]: s |- aparens[a] ==> v \\
    s |- a ==> v;
end

system Bexp : State |- Bexp ==> Bool =
  [[ CONST ]]: s |- bconst[b] ==> b;

  [[ EQ ]]: s |- eq[a1, a2] ==> v1 == v2 \\
    s |- a1 =Aexp=> v1,
    s |- a2 =Aexp=> v2;

  [[ LT ]]: s |- lt[a1, a2] ==> v1 < v2 \\
    s |- a1 =Aexp=> v1,
    s |- a2 =Aexp=> v2;

  [[ NOT ]]: s |- not[b] ==> !v \\
    s |- b ==> v;

  [[ PARENS ]]: s |- bparens[b] ==> v \\
    s |- b ==> v;

  [[ AND ]]: s |- and[b1, b2] ==> v1 & v2 \\
    s |- b1 ==> v1,
```

```
    s |- b2 ==> v2;
end

system Stm : Stm * State ==> State =
  [[ SKIP ]]: (skip[], s) ==> s;

  [[ ASS ]]: (ass[x, a], s) ==> [ x -> v ]s \\
    s |- a =Aexp=> v;

  [[ COMP ]]: (comp[S1, S2], s) ==> s' \\
    (S1, s) ==> s'',
    (S2, s'') ==> s';

  [[ IF ]]: (ifs[b, S1, S2], s) ==> s' \\
    s |- b =Bexp=> true,
    (S1, s) ==> s';

  [[ IF ]]: (ifs[b, S1, S2], s) ==> s' \\
    s |- b =Bexp=> false,
    (S2, s) ==> s';

  [[ WHILE ]]: (while[b, S], s) ==> s' \\
    s |- b =Bexp=> true,
    (S, s) ==> s'',
    (while[b, S], s'') ==> s';

  [[ WHILE ]]: (while[b, S], s) ==> s \\
    s |- b =Bexp=> false;
end
```

Below is the generated LaTeX code. It was only slightly adjusted for better pagination:

$$State = \chi^* \to \mathbb{Z}$$

$$\begin{aligned} Aexp ::= & \texttt{add}\ Aexp \times Aexp \\ | & \texttt{mul}\ Aexp \times Aexp \\ | & \texttt{sub}\ Aexp \times Aexp \\ | & \texttt{aparens}\ Aexp \\ | & \texttt{aconst}\ \mathbb{Z} \\ | & \texttt{var}\ \chi^* \end{aligned}$$

$$\begin{aligned} Bexp ::= & \texttt{eq}\ Aexp \times Aexp \\ | & \texttt{lt}\ Aexp \times Aexp \\ | & \texttt{not}\ Bexp \\ | & \texttt{and}\ Bexp \times Bexp \\ | & \texttt{bparens}\ Bexp \\ | & \texttt{bconst}\ \{tt, ff\} \end{aligned}$$

$$\begin{aligned} Stm ::= & \texttt{skip} \\ | & \texttt{ass}\ \chi^* \times Aexp \\ | & \texttt{comp}\ Stm \times Stm \\ | & \texttt{ifs}\ Bexp \times Stm \times Stm \\ | & \texttt{while}\ Bexp \times Stm \end{aligned}$$

$$\boldsymbol{Aexp} : State \vdash Aexp \Downarrow \mathbb{Z}$$

[VAR]: $s \vdash \texttt{var}[x] \Downarrow s(x)$

[CONST]: $s \vdash \texttt{aconst}[n] \Downarrow n$

[ADD]: $$\frac{\begin{array}{c} s \vdash a_1 \Downarrow v_1 \\ s \vdash a_2 \Downarrow v_2 \end{array}}{s \vdash \texttt{add}[a_1, a_2] \Downarrow v_1 + v_2}$$

$$\text{[MUL]:}\quad \frac{s \vdash a_1 \Downarrow v_1 \quad s \vdash a_2 \Downarrow v_2}{s \vdash \texttt{mul}[a_1, a_2] \Downarrow v_1 \cdot v_2}$$

$$\text{[SUB]:}\quad \frac{s \vdash a_1 \Downarrow v_1 \quad s \vdash a_2 \Downarrow v_2}{s \vdash \texttt{sub}[a_1, a_2] \Downarrow v_1 - v_2}$$

$$\text{[PARENS]:}\quad \frac{s \vdash a \Downarrow v}{s \vdash \texttt{aparens}[a] \Downarrow v}$$

$$\textbf{\textit{Bexp}} : \mathit{State} \vdash \mathit{Bexp} \Downarrow \{\mathit{tt}, \mathit{ff}\}$$

$$\text{[CONST]:}\quad s \vdash \texttt{bconst}[b] \Downarrow b$$

$$\text{[EQ]:}\quad s \vdash \texttt{eq}[a_1, a_2] \Downarrow v_1 = v_2 \qquad s \vdash a_1 \Downarrow_{\textbf{\textit{Aexp}}} v_1 \quad s \vdash a_2 \Downarrow_{\textbf{\textit{Aexp}}} v_2$$

$$\text{[LT]:}\quad s \vdash \texttt{lt}[a_1, a_2] \Downarrow v_1 < v_2 \qquad s \vdash a_1 \Downarrow_{\textbf{\textit{Aexp}}} v_1 \quad s \vdash a_2 \Downarrow_{\textbf{\textit{Aexp}}} v_2$$

$$\text{[NOT]:}\quad \frac{s \vdash b \Downarrow v}{s \vdash \texttt{not}[b] \Downarrow \neg v}$$

$$\text{[PARENS]:}\quad \frac{s \vdash b \Downarrow v}{s \vdash \texttt{bparens}[b] \Downarrow v}$$

$$\text{[AND]:}\quad \frac{s \vdash b_1 \Downarrow v_1 \quad s \vdash b_2 \Downarrow v_2}{s \vdash \texttt{and}[b_1, b_2] \Downarrow v_1 \wedge v_2}$$

$$\textbf{\textit{Stm}} : \mathit{Stm} \times \mathit{State} \Downarrow \mathit{State}$$

$$\text{[SKIP]:}\quad \langle \texttt{skip}[], s\rangle \Downarrow s$$

$$\text{[ASS]:}\quad \langle \texttt{ass}[x, a], s\rangle \Downarrow [x \mapsto v]s \qquad s \vdash a \Downarrow_{Aexp} v$$

$$\text{[COMP]:}\quad \frac{\langle S_1, s\rangle \Downarrow s'' \quad \langle S_2, s''\rangle \Downarrow s'}{\langle \texttt{comp}[S_1, S_2], s\rangle \Downarrow s'}$$

$$\text{[IF]:}\quad \frac{\langle S_1, s\rangle \Downarrow s'}{\langle \texttt{ifs}[b, S_1, S_2], s\rangle \Downarrow s'} \qquad s \vdash b \Downarrow_{Bexp} tt$$

$$\text{[IF]:}\quad \frac{\langle S_2, s\rangle \Downarrow s'}{\langle \texttt{ifs}[b, S_1, S_2], s\rangle \Downarrow s'} \qquad s \vdash b \Downarrow_{Bexp} ff$$

$$\text{[WHILE]:}\quad \frac{\langle S, s\rangle \Downarrow s'' \quad \langle \texttt{while}[b, S], s''\rangle \Downarrow s'}{\langle \texttt{while}[b, S], s\rangle \Downarrow s'} \qquad s \vdash b \Downarrow_{Bexp} tt$$

$$\text{[WHILE]:}\quad \langle \texttt{while}[b, S], s\rangle \Downarrow s \qquad s \vdash b \Downarrow_{Bexp} ff$$

D.2 Flan

The language ***Flan***[12] (**Fl**unctional **lan**guage) is a variation on the simple λ-calculus. It served as a foundation for the functional aspects of the metalanguage specification.

```
domain Values = [
  vconst[Fcon] +
  vpair[Values * Values] +
  cloj[Symbol * Fexp * Env] +
  rcloj[Symbol * Symbol * Fexp * Env]
];

domain Env = Symbol -> Values;

syntax Fexp = var of Symbol
  | const of Fcon
```

```
  | pair of Fexp * Fexp
  | app of Fexp * Fexp
  | iff of Fexp * Fexp * Fexp
  | fn of Symbol * Fexp
  | letnr of Symbol * Fexp * Fexp
  | letr of Symbol * Fexp * Fexp;

syntax Fcon = number of Int
  | bconst of Bool
  | plus | times | minus | equal | not | is0;

system Fexp : Env |- Fexp ==> Values =
  [[ VAR ]]: env |- var[x] ==> env(x);

  [[ CONST ]]: env |- const[c] ==> vconst[c];

  [[ PAIR ]]: env |- pair[e1, e2] ==> vpair[v1, v2] \\
    env |- e1 ==> v1,
    env |- e2 ==> v2;

  [[ APP ]]: env |- app[e1, e2] ==> v \\
    env |- e1 ==> cloj[x, e, env'],
    env |- e2 ==> v',
    [x -> v']env' |- e ==> v;

  [[ APP ]]: env |- app[e1, e2] ==> v \\
    env |- e1 ==> vconst[c],
    env |- e2 ==> v',
    (c, v') =apply=> v;

  [[ APP ]]: env |- app[e1, e2] ==> v \\
    env |- e1 ==> rcloj[f, x, e, env'],
    env |- e2 ==> v',
    let rclojure = rcloj[f, x, e, env'],
    [x -> v'][f -> rclojure]env' |- e ==> v;

  [[ IF ]]: env |- iff[b, e1, e2] ==> v \\
    env |- b ==> vconst[bconst[true]],
    env |- e1 ==> v;

  [[ IF ]]: env |- iff[b, e1, e2] ==> v \\
```

```
    env |- b ==> vconst[bconst[false]],
    env |- e2 ==> v;

  [[ FN ]]: env |- fn[x, e] ==> cloj[x, e, env];

  [[ LET ]]: env |- letnr[x, e1, e2] ==> v \\
    env |- e1 ==> v',
    [x -> v']env |- e2 ==> v;

  [[ LETREC ]]: env |- letr[f, e1, e2] ==> v \\
    env |- e1 ==> cloj[x, e, env'],
    [f -> rcloj[f, x, e, env']]env |- e2 ==> v;
end

system apply : Fcon * Values ==> Values =
  [[ PLUS ]]: (plus[], vpair[
      vconst[number[v1]],
      vconst[number[v2]]
    ]) ==> vconst[number[v1 + v2]];

  [[ MINUS ]]: (minus[], vpair[
      vconst[number[v1]],
      vconst[number[v2]]
    ]) ==> vconst[number[v1 - v2]];

  [[ TIMES ]]: (times[], vpair[
      vconst[number[v1]],
      vconst[number[v2]]
    ]) ==> vconst[number[v1 * v2]];

  [[ EQUAL ]]: (equal[], vpair[
      vconst[number[v1]],
      vconst[number[v2]]
    ]) ==> vconst[bconst[v1 == v2]];

  [[ EQUAL ]]: (equal[], vpair[
      vconst[bconst[v1]],
      vconst[bconst[v2]]
    ]) ==> vconst[bconst[v1 == v2]];

  [[ NOT ]]: (not[], vconst[bconst[b]])
```

```
    ==> vconst[bconst[!b]];

  [[ IS-ZERO ]]: (is0[], vconst[number[n]])
    ==> vconst[bconst[n == 0]];
end
```

Below, the generated LaTeX is give. As with **Bims**. it was modified manually slightly to fit better in the page.

$$
\begin{aligned}
\mathit{Values} = \bigcup(&\texttt{vconst}[\mathit{Fcon}]+\\
&\texttt{vpair}[\mathit{Values}\times\mathit{Values}]+\\
&\texttt{cloj}[\chi^*\times\mathit{Fexp}\times\mathit{Env}]+\\
&\texttt{rcloj}[\chi^*\times\chi^*\times\mathit{Fexp}\times\mathit{Env}])
\end{aligned}
$$

$$\mathit{Env} = \chi^* \to \mathit{Values}$$

$$
\begin{aligned}
\mathit{Fexp} ::= {}& \texttt{var}\ \chi^*\\
\mid {}& \texttt{const}\ \mathit{Fcon}\\
\mid {}& \texttt{pair}\ \mathit{Fexp}\times\mathit{Fexp}\\
\mid {}& \texttt{app}\ \mathit{Fexp}\times\mathit{Fexp}\\
\mid {}& \texttt{iff}\ \mathit{Fexp}\times\mathit{Fexp}\times\mathit{Fexp}\\
\mid {}& \texttt{fn}\ \chi^*\times\mathit{Fexp}\\
\mid {}& \texttt{letnr}\ \chi^*\times\mathit{Fexp}\times\mathit{Fexp}\\
\mid {}& \texttt{letr}\ \chi^*\times\mathit{Fexp}\times\mathit{Fexp}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Fcon} ::= {}& \texttt{number}\ \mathbb{Z}\\
\mid {}& \texttt{bconst}\ \{\mathit{tt},\mathit{ff}\}\\
\mid {}& \texttt{plus}\\
\mid {}& \texttt{times}\\
\mid {}& \texttt{minus}\\
\mid {}& \texttt{equal}\\
\mid {}& \texttt{not}\\
\mid {}& \texttt{is0}
\end{aligned}
$$

$$\textbf{\textit{Fexp}} : Env \vdash Fexp \Downarrow Values$$

[VAR]: $env \vdash \texttt{var}[x] \Downarrow env(x)$

[CONST]: $env \vdash \texttt{const}[c] \Downarrow \texttt{vconst}[c]$

[PAIR]: $$\frac{env \vdash e_1 \Downarrow v_1 \quad env \vdash e_2 \Downarrow v_2}{env \vdash \texttt{pair}[e_1, e_2] \Downarrow \texttt{vpair}[v_1, v_2]}$$

[APP]: $$\frac{env \vdash e_1 \Downarrow \texttt{cloj}[x, e, env'] \quad env \vdash e_2 \Downarrow v' \quad [x \mapsto v']env' \vdash e \Downarrow v}{env \vdash \texttt{app}[e_1, e_2] \Downarrow v}$$

[APP]: $$\frac{env \vdash e_1 \Downarrow \texttt{vconst}[c] \quad env \vdash e_2 \Downarrow v'}{env \vdash \texttt{app}[e_1, e_2] \Downarrow v} \qquad \langle c, v' \rangle \Downarrow_{\textbf{\textit{apply}}} v$$

[APP]: $$\frac{env \vdash e_1 \Downarrow \texttt{rcloj}[f, x, e, env'] \quad env \vdash e_2 \Downarrow v' \quad [x \mapsto v']([f \mapsto \texttt{rcloj}[f, x, e, env']]env') \vdash e \Downarrow v}{env \vdash \texttt{app}[e_1, e_2] \Downarrow v}$$

[IF]: $$\frac{env \vdash b \Downarrow \texttt{vconst}[\texttt{bconst}[tt]] \quad env \vdash e_1 \Downarrow v}{env \vdash \texttt{iff}[b, e_1, e_2] \Downarrow v}$$

[IF]: $$\frac{env \vdash b \Downarrow \texttt{vconst}[\texttt{bconst}[ff]] \quad env \vdash e_2 \Downarrow v}{env \vdash \texttt{iff}[b, e_1, e_2] \Downarrow v}$$

[FN]: $env \vdash \texttt{fn}[x, e] \Downarrow \texttt{cloj}[x, e, env]$

[LET]: $$\frac{env \vdash e_1 \Downarrow v' \quad [x \mapsto v']env \vdash e_2 \Downarrow v}{env \vdash \texttt{letnr}[x, e_1, e_2] \Downarrow v}$$

[LETREC]: $$\frac{env \vdash e_1 \Downarrow \texttt{cloj}[x, e, env'] \quad [f \mapsto \texttt{rcloj}[f, x, e, env']]env \vdash e_2 \Downarrow v}{env \vdash \texttt{letr}[f, e_1, e_2] \Downarrow v}$$

$$\textbf{\textit{apply}} : Fcon \times Values \Downarrow Values$$

[PLUS]: $\langle \texttt{plus}[], \texttt{vpair}[\texttt{vconst}[\texttt{number}[v_1]], \texttt{vconst}[\texttt{number}[v_2]]] \rangle \Downarrow \texttt{vconst}[\texttt{number}[v_1 + v_2]]$

[MINUS]: $\langle \texttt{minus}[], \texttt{vpair}[\texttt{vconst}[\texttt{number}[v_1]], \texttt{vconst}[\texttt{number}[v_2]]] \rangle \Downarrow \texttt{vconst}[\texttt{number}[v_1 - v_2]]$

[TIMES]: $\langle \texttt{times}[], \texttt{vpair}[\texttt{vconst}[\texttt{number}[v_1]], \texttt{vconst}[\texttt{number}[v_2]]] \rangle \Downarrow \texttt{vconst}[\texttt{number}[v_1 \cdot v_2]]$

[EQUAL]: $\langle \texttt{equal}[], \texttt{vpair}[\texttt{vconst}[\texttt{number}[v_1]], \texttt{vconst}[\texttt{number}[v_2]]] \rangle \Downarrow \texttt{vconst}[\texttt{bconst}[v_1 = v_2]]$

[EQUAL]: $\langle \texttt{equal}[], \texttt{vpair}[\texttt{vconst}[\texttt{bconst}[v_1]], \texttt{vconst}[\texttt{bconst}[v_2]]] \rangle \Downarrow \texttt{vconst}[\texttt{bconst}[v_1 = v_2]]$

[NOT]: $\langle \texttt{not}[], \texttt{vconst}[\texttt{bconst}[b]] \rangle \Downarrow \texttt{vconst}[\texttt{bconst}[\neg b]]$

[IS-ZERO]: $\langle \texttt{is0}[], \texttt{vconst}[\texttt{number}[n]] \rangle \Downarrow \texttt{vconst}[\texttt{bconst}[n = 0]]$