Master's Thesis

Implementing Cast Calculus for Label Dependent Lambda Calculus

Thomas Leyh

Examiner: Prof. Dr. Peter Thiemann
Second Assessor: Prof. Dr. Andreas Podelski

Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Programming Languages

March 16, 2022

Writing period

22. 11. 2021 – 23. 05. 2022

Examiner

Prof. Dr. Peter Thiemann

Second Assessor

Prof. Dr. Andreas Podelski

Declaration

Freiburg, 16.03.2022 T. Leyh

Place, Date Signature

Abstract

Dependent typing is a double-edged sword: One the one hand, it allows for more precise types, hence more guarantees of the program's correctness at compile time while also giving more hints about the program's structure and properties when reading the code. On the other hand, the programmer needs to provide these types in the first place, thus more work is required, especially in volatile code bases. A combination of label dependent types with gradual typing—allowing dynamic typing in an otherwise statically typed language—is proposed by Fu et al. [2021]. Building upon the proposal, this thesis provides an implementation of the *Cast Calculus Label Dependent Lambda Calculus* (CCLDLC) in the Haskell programming language. The *Cast Calculus* introduces dynamic type $\star$ as well as an explicit type cast notation $(M : A \Rightarrow B)$. By applying various rules for casts between arbitrary types and the dynamic type these additional constructs integrate into the basic *Label Dependent Lambda Calculus* (LDLC). Extending an existing interpreter, this builds the foundation for full gradual typting.
The accompanying code resides at `https://github.com/leyhline/ldgv/`.

Contents

List of Figures

List of Source Code Listings

1 Introduction

In programming, dependent types are the next step in the unification of language constructs. Not only values and functions but also types are first class constructs. When declaring types, it is now possible to depend on values, e. g. in addition to type *Integer*, one may define a type ($Integer\ n$), only accepting a value $x \in \mathbb{Z}$ when $x \leq n$.

This goes along with practical advantages: more precise typing, consequently better correctness guarantees and more expressive power when describing programs. Nevertheless, such power comes at a cost: As the difference between proof and program grows hazy, type checking will face similar problems as program execution. The additional complexity of fully dependent types implies *Turing completeness*. Hence, type checking will be subject to the *halting problem*.

1.1 Label Dependent Types

Over the years, different approaches to dependent types surfaced, mitigating such disadvantages. This work is based on a subset of "Label Dependent Session Types" by Thiemann and Vasconcelos [2019]. Ignoring session types, we focus on the *Label Dependent Lambda Calculus* (LDLC) it introduces, extending it during this thesis. In general, types in LDLC refer to finite sets of labels. For example, it is easy to define type **Bool** with two labels representing *True* and *False* respectively:

$$\textbf{Bool} := \{\ell_T, \ell_F\} \tag{1}$$

Elimination is done using a **case** construct, matching labels with expressions or types. In equation 2, a function accepting two arguments x and y is given. While x is of type **Bool**, y depends on the value of x, hence y may have type **Int** or type **Bool**. The same holds for the function's body as seen in the whole function's type annotation, given as Π-type. This quickly results in unwieldy type annotations.

$$\begin{aligned}
&\lambda(x : \textbf{Bool})\\
&\quad .\lambda(y : \textbf{case } x\{\ell_T : \textbf{Int}, \ell_F : \textbf{Bool}\})\\
&\qquad .\textbf{ case } x\{\ell_T : 17 + y, \ell_F : \text{not } y\}\\
&:\Pi(x : \textbf{Bool})\\
&\quad .\Pi(y : \textbf{case } x\{\ell_T : \textbf{Int}, \ell_F : \textbf{Bool}\})\\
&\qquad .\textbf{ case } x\{\ell_T : \textbf{Int}, \ell_F : \textbf{Bool}\}
\end{aligned} \tag{2}$$

Label-dependency strikes a middle ground between full dependent typing and its complete absence. Thus, it comes with similar advantages. It is not as complex. The compiler has more options to check type safety, resulting in a more reliable program. Furthermore, types can serve as a machine-readable and machine-verifiable documentation since they give hints about the workings of the code. As a disadvantage, the code above is quite verbose. Adding all these type annotations can be cumbersome for a programmer, especially when still exploring a given task and trying out different approaches.

1.2 Gradual Typing

Gradual Typing mitigates this problem since it allows mixing static and dynamic typing. In their paper, Fu et al. [2021] propose an extension, the *Gradual Label Dependent Lambda Calculus* (GLDLC), introducing the dynamic type $\star$. Annotating terms with $\star$ results in type checking at runtime, still preserving the guarantees of statically typed code at its boundaries. One possibility for rewriting equation 2 is by replacing the **case** construct describing the type of variable y by dynamic type $\star$.

$$\begin{aligned}&\lambda(x : \textbf{ Bool }).\lambda(y : \star).\ \textbf{case}\ x\{\ell_T : 17 + y, \ell_F : \text{ not } y\}\\ :&\Pi(x : \textbf{ Bool }).\Pi(y : \star).\ \textbf{case}\ x\{\ell_T : \textbf{ Int }, \ell_F : \textbf{ Bool }\}\end{aligned} \tag{3}$$

The original paper lists more advantages and applications of GLDLC, whereas this work focuses on implementation aspects. Therefore, the methods used for evaluating type $\star$ are of interest. This leads to an intermediate representation, describing the cast from arbitrary type A to $\star$ and from $\star$ to an arbitrary type B. These casts are part of the *Cast Calculus Label Dependent Lambda Calculus* (CCLDLC). The *Cast Calculus* adds an explicit type cast expression $(M : A \Rightarrow B)$, casting the value of evaluated expression M of type A to type B. Naturally, type $\star$ may occur on both sides. Rewriting equation 3 results in a more verbose and more explicit formulation.

$$\begin{aligned}&\lambda(x : \textbf{ Bool }).\lambda(y : \star)\\ &\quad .\ \textbf{case}\ x\{\ell_T : 17 + (y : \star \Rightarrow \textbf{ Int }), \ell_F : \text{ not } (y : \star \Rightarrow \textbf{ Bool })\}\\ :&\Pi(x : \textbf{ Bool }).\Pi(y : \star)\\ &\quad .\ \textbf{case}\ x\{\ell_T : \textbf{ Int }, \ell_F : \textbf{ Bool }\}\end{aligned} \tag{4}$$

In CCLDLC, a cast must always happen in such an explicit manner. The same is true for function application. When passing an expression for variable y, there must be an explicit cast to the dynamic type, e. g. $(\ell_F : \textbf{Bool} \Rightarrow \star)$.

We contribute an implementation of CCLDLC by extending LDGV, an existing interpreter for Label Dependent Session Types in the Haskell programming language.
The full source code is available at `https://github.com/leyhline/ldgv/`.

First, we describe the *Label Dependent Lambda Calculus*, its rules and its interpreter LDGV since these provide the foundation for our extension. Next, we formally describe the rules of CCLDLC including a section on the current implementation's structure and a top-level overview of necessary adjustments. Finally, we document the implementation in a detailed manner.

1.3 Notation

Since this work is about building an implementation, there needs to be a distinction between concepts and code. Often, both will be quite similar. Concepts like formulas and rules are conveyed using mathematical notation. This is not necessarily identical to the actual syntax of LDGV.

$x := \ell_T$	variable definition
$A := \{\ell_T\}$	type definition

Then again, actual code precisely reflects the capabilities of LDGV. One can feed it verbatim to the interpreter. It is written in a monospaced font.

Listing 1.1: LDGV: Source code example

```
val x = 'foo
type A : ~un = { 'foo }
```

2 Foundations

Before diving into the *Cast Calculus* we introduce the foundation, the *Label Dependent Lambda Calculus* (LDLC) and its implementation LDGV.

2.1 Label Dependent Lambda Calculus

The LDLC is the foundation of this work. Its main aspect is the gain of additional simplicity in context of session types. The introduction of labels results in decoupled operations when sending and receiving functions, hence being more lightweight regarding its semantics. See Thiemann and Vasconcelos [2019]. However, session types are not of special relevance when introducing casts. For brevity, we will omit the handling of session types, focusing on labels instead.

A list of language constructs is found in figures 1 to 3. There is support for common functionality like variables, expressions and types,[1] functions and pairs as well as numerical operations and values. In general—as implied by its name—it is based on *Simply Typed Lambda Calculus* (details can be found in most introductory literature, e.g. Pierce [2002]). As in many functional languages, there is also the unit type **Unit** encompassing its only value "()".

Another two language concepts are more interesting: First and foremost, the support for labels ℓ. Labels can have arbitrary names. A mathematical set of distinct labels forms a type. Moreover, there is a recursor over natural

[1]Variable names for expressions have to start with a lowercase letter, e.g. `val x = ()`; for types its an uppercase letter: `type X : Unit`.

$$
\begin{array}{rlr}
M, N, P ::= & x \mid () \mid \ell \mid x \in \mathbb{R} & \\
& \mid M \circ N & \circ \in \{+, -, \times, \div\} \\
& \mid \ \textbf{case}\ M\{\overline{\ell : N_\ell}^{\ell \in L}\} & \text{label expression matching} \\
& \mid \ \textbf{rec}\ V\ M\ x.f.N & \mathbb{N}\text{ recursor} \\
& \mid \lambda(x : A).M & \text{abstraction} \\
& \mid M\ N & \text{application} \\
& \mid \ \textbf{let}\ x = M\ \textbf{in}\ N & \text{bind in } N \\
& \mid \ \textbf{let}\ \langle x, y\rangle = M\ \textbf{in}\ N & \text{bind pair in } N \\
& \mid \langle x = M, N\rangle & \text{pair construction} \\
& \mid \ \textbf{fst}\ M \mid \ \textbf{snd}\ M & \text{pair elimination}
\end{array}
$$

Figure 1: Expressions in LDLC

$$
\begin{array}{rlr}
A, B ::= & t \mid \ \textbf{Unit} & \\
& \mid \ \textbf{Int} \ \mid \ \textbf{Nat} \ \mid \ \textbf{Double} & \\
& \mid \{\ell_1, \dots, \ell_n\} & \text{finite label set} \\
& \mid \Sigma(x : A)B & \text{pair type} \\
& \mid \Pi(x : A)B & \text{function type} \\
& \mid \ \textbf{case}\ M\{\overline{\ell : A_\ell}^{\ell \in L}\} & \text{label type matching} \\
& \mid \ \textbf{rec}\ V\ A\ t.B & \text{recursor type}
\end{array}
$$

Figure 2: Types in LDLC

numbers **rec** $V\ M\ x.f.N$. This allows for recursively constructing expressions by using base expression M for $n = 0$ and otherwise expression N with appropriate substitutions for x and f.

2.1.1 Labels and Label Sets

The main feature of LDLC is its use of labels. A label evaluates to itself, thus it forms its own value. But what about its type? Any finite non-empty set of labels L is a type. For instance, it is sufficient to declare a boolean type as seen in equation 1.

$$V, W ::= () \mid \ell \mid x \in \mathbb{R} \mid \langle V, W \rangle$$

Figure 3: Values in LDLC

The minimal type is the singleton type of its corresponding label. For label ℓ this is $\{\ell\}$. Apart from that there might be a bottom type $\perp$ which corresponds to a hypothetical empty label set. Neither type $\perp$ nor the empty label set are valid types in LDLC.

Predominantly, labels appear in context of **case** terms. These may occur as expressions or as types and act as eliminators for labels. They provide a mapping from each label of the corresponding label set to an expression or a type. Hence the name of the calculus, *Label Dependent* Lambda Calculus. The type of an expression may depend on a label value.

2.1.2 Recursor over Natural Numbers

In addition to label dependency, there is another construct used for modeling dependency on natural numbers: $\textbf{rec}\ V\ M\ x.f.N$

Value V must be a natural number value which can also be written recursively using the constructors Z (for zero) and $S(V)$ for the successor of V. An encoding for a natural number is given by $\overline{n} = S(\ldots S(Z))$. When evaluating a recursor, there is a case differentiation by value V.

If $V = Z$ the whole expression is reduced to M, discarding its latter term. Otherwise, for $V = S(V')$, the **rec** expression is reduced to expression N with the following two substitutions: variable x in N is substituted by V', the predecessor of $S(V')$; variable f by $\textbf{rec}\ V'\ M\ x.f.N$, the result of the "previous" evaluation of the recursor.

On the type level it is similar when evaluating a recursor type $\textbf{rec}\ V\ A\ t.B$. For $V = Z$, the resulting type is simply A. As for $V = S(V')$, the resulting type is given by B after substituting type identifier t in B with $\textbf{rec}\ V'\ A\ t.B$.

$$\begin{aligned}
\textbf{rec}\ Z\ M\ x.f.N &\longrightarrow M \\
\textbf{rec}\ S(V)\ M\ x.f.N &\longrightarrow N[V/x][\ \textbf{rec}\ V\ M\ x.f.N/f] \\
\textbf{rec}\ Z\ A\ t.B &\longrightarrow A \\
\textbf{rec}\ S(V)\ A\ t.B &\longrightarrow B[\ \textbf{rec}\ V\ A\ t.B/t]
\end{aligned}$$

Figure 4: Nat. recursor expression reduction and type reduction

Both reductions are concisely listed in figure 4.

2.1.3 Subtyping

Without subtyping, a type system lacks flexibility. Assuming a lambda expression $\lambda(x : \textbf{ Int }).M$, what will happen when applying an expression of type **Nat**? Obviously, there is a difference between natural numbers and integers. But at the same time, every natural number is also an integer. ($\mathbb{N} \subset \mathbb{Z}$) By formulating consistent rules—not necessarily involving subset relations—a type system will accept a wider range of types. Especially in object-oriented languages supporting inheritance this is practically a necessary feature.[2] Subtyping for dependent types entails some peculiarities that were studied in various papers during the last three decades, e.g. by Aspinall and Compagnoni [2001].

In figure 5 we give an excerpt of relevant subtyping rules of LDLC. Handling of function types (A-Sub-Pi) and pair types (A-Sub-Sigma) is of special relevance since they are composite types, thus they need specification on how to recursively apply subtyping to their component types. For CCLDLC these rules gain additional relevance since they are necessary not only for type checking but now for type casts during runtime, too.

Since a label type is a set of labels we can use set comparisons for subtyping. If one label type L is a subset of another label type L', then L

[2]A common example is class *Dog* inheriting from class *Animal*; then methods accepting arguments of type *Animal* will also accept type *Dog*.

$$\textsc{A-Sub-Unit}\quad \Gamma \Vdash \textbf{Unit} \leq \textbf{Unit} \qquad\qquad \textsc{A-Sub-Label}\quad \frac{L \subseteq L'}{\Gamma \Vdash L \leq L'}$$

$$\textsc{A-Sub-Pi}\quad \frac{\Gamma \Vdash A' \leq A \qquad \Gamma, x : A' \Vdash B \leq B'}{\Gamma \Vdash \Pi(x : A)B \leq \Pi(x : A')B'}$$

$$\textsc{A-Sub-Sigma}\quad \frac{\Gamma \Vdash A \leq A' \qquad \Gamma, x : A \Vdash B \leq B'}{\Gamma \Vdash \Sigma(x : A)B \leq \Sigma(x : A')B'}$$

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

Figure 5: Subtyping rules in LDGV (excerpt) w. r. t. type environment Γ

is a subtype of L'. Thus, there is no L that is a subtype of the empty set $\emptyset$ and equivalently of bottom type $\bot$. Subtyping in LDLC is closed under transitivity.

2.2 LDGV Interpreter

LDGV stands for "Label Dependency" and "Gay & Vasconcelos", the authors of multiple papers about Session Types, e. g. Gay and Vasconcelos [2010]. In this work's context, it is the name of an interpreter for LDLC and its extension CCLDLC. It is written in the Haskell programming language.

Its syntax is similar to the calculus from the previous section. We give an excerpt in listing 2.1. In general, keywords and structure are identical safe for labels. To uniquely identify and tokenize a label it needs to be a combination of alphanumeric characters, prefixed by an apostrophe. Furthermore, as opposed to the underlying LDLC, there is basic string support. A combination of characters enclosed by quotation marks `"` is read as string expression.

The syntax for types is given in listing 2.2. Writing out function and pair types differs from the calculus' notation. Function types of multiple

Listing 2.1: LDGV: Expressions (excerpt) with expressions `M,N,P`, variable `x` and type `A`

```
()              -- unit
M + N           -- and other arithmetics
10              -- numbers
"foo bar"       -- strings
x               -- variables
'foo            -- labels
case M {'foo: N, 'bar: P}
fn(x: A) M      -- lambda abstraction
<x = M, N>      -- pair
```

arguments are built by recursion, e.g. `(x:A) -> (y:B) -> C`.

For an actual program there need to be statements, especially declarations. They create the global environment, binding expressions to variables and types to type identifiers.[3] When running the interpreter on a given program, it outputs the evaluation result of the expression bound to variable `main`. Furthermore, there are type assertions. The full list of statements is given in listing 2.3.

Translating equation 2 into a valid LDGV program results in the sample program in figure 2.4. The variable's type signatures are optional. Variable `main` gives an example application and does fully evaluate to value `'F` (representing boolean *False*).

[3]For type identifiers, the programmer has to supply a *kind.* For brevity, we omit the introduction of kinds, generally assuming kind `~un` in declarations.

Listing 2.2: LDGV: Types (excerpt) with expression `M`, variable `x`, type identifier `T` and types `A,B`

```
Unit
Int
Nat
Double
String
T                  -- type identifier
{'foo, 'bar}       -- label type
case M {'foo: A, 'bar: B}
(x: A) -> B        -- function type
[x: A, B]          -- pair type
```

Listing 2.3: LDGV: Statements with expression `M` and types `A,B`

```
val x = M           -- variable declaration
val x : A           -- variable signature declaration
type T : ~un = A    -- type declaration
A <: B              -- subtype assertion
A =: B              -- type equivalence assertion
```

Listing 2.4: LDGV: Sample program

```
type Bool : ~un = {'T, 'F}

val not : (a: Bool) -> Bool
val not   (a: Bool) = (case a {'T: 'F, 'F: 'T})

val f : (x: Bool)
        -> (y: case x {'T: Int, 'F: Bool})
           -> case x {'T: Int, 'F: Bool}
val f = fn(x: Bool)
          fn(y: case x {'T: Int, 'F: Bool})
            case x {'T: 17+y, 'F: not y}

val main = f 'F 'T
```

3 Cast Calculus

The *Cast Calculus Label Dependent Lambda Calculus* is an extension to regular LDLC as introduced in section 2.1. It adds an explicit cast between two types, a dynamic type $\star$ as well as related language constructs. CCLDLC is but an intermediate representation between LDLC and a gradual typing extension GLDLC. With gradual typing, type casts involving the dynamic type are hidden and handled internally. In other words, a program written in GLDLC is internally converted to CCLDLC by making type casts explicit. The actual cast reductions happens in the latter calculus. This work is solely about implementing the *Cast Calculus* without going into conversion from the *Gradual Calculus.*

3.1 Expressions and Types

CCLDLC adds expressions for cast and blame as well as the dynamic type, top type and bottom type; see figure 6. The bottom type's single purpose is as type for the blame expression (**blame** $: \bot$). The top type is only used for unfolding the bottom type, evaluating $\bot$ to a function of any argument

$$\begin{aligned} M &::= \;\dots \mid M : A \Rightarrow B \mid \textbf{blame} \\ A &::= \;\dots \mid \star \mid \bot \mid \top \end{aligned}$$

Figure 6: CCLDLC extensions for expression M and types A, B.

$\Pi(y : \top)\bot$, the *smallest function type*. Every type A is a subtype of $\top$ ($A \leq \top$) and $\bot$ is a subtype of every type B ($\bot \leq B$). Hence, the smallest function type we just described contains functions accepting any argument, then always evaluating to **blame**.

3.2 Values

Values are extended in a less intuitive way, see figure 7. First, there is a *dynamic wrapper* $V : G \Rightarrow \star$, representing a cast of value V from a ground type G to dynamic type $\star$. The whole cast term is a value itself. A ground type indicates an application of dynamic arguments to a top level type constructor as well as subtyping capabilities. Hence, label types are ground types, together with the unit type, built-in numeric types and corresponding singleton types. There are also ground types for function types where argument and return type are unknown ($\Pi(x : \star)\star$) and for pair types with unknown component types ($\Sigma(x : \star)\star$). Here, "unknown type" means unknown before runtime, thus we use the dynamic type $\star$.

Next, a function closure where the evaluation environment ρ is explicitly part of the value. Environment ρ is an ordered mapping of variable names to values. It is necessary for substitutions in expressions with bound variables on evaluation. Providing the environment as part of a function value implies deferred evaluation. Instead of strictly substituting variables in the function's body, full evaluation is delayed by supplying ρ for looking up values when further evaluation is necessary. In CCLDLC, we simultaneously use ρ for mapping type identifiers to types for similar reasons in context of cast reductions. This is possible because in the language's syntax, variables and type identifiers necessarily have different names: variables starting with lowercase, type identifiers with an uppercase letter.

And finally a function type cast for value V of a specific function type to another one. This is necessary for reducing function casts and is shown in

$$
\begin{array}{rll}
V ::= & \dots \mid (V : G \Rightarrow \star) & \text{dynamic wrapper} \\
& \mid (\rho, \lambda(x : A)M) & \text{closure} \\
& \mid (V : (\rho, \Pi(x : A)A') \Rightarrow (\rho, \Pi(x : B)B')) & \text{function type cast} \\
G ::= & \mathbf{Unit} \mid L & \\
& \mid \mathbf{Nat} \mid \mathbf{Int} \mid \mathbf{Double} & \\
& \mid \mathbf{S}\{V : \mathbf{Unit}\} \mid \mathbf{S}\{V : L\} & \\
& \mid \mathbf{S}\{V : \mathbf{Nat}\} \mid \mathbf{S}\{V : \mathbf{Int}\} & \\
& \mid \mathbf{S}\{V : \mathbf{Double}\} & \\
& \mid \Pi(x : \star)\star \mid \Sigma(x : \star)\star & \\
\rho ::= & \cdot \mid \rho, x = V \mid \rho, t = A &
\end{array}
$$

Figure 7: Values V and ground types G in CCLDLC with environment ρ

greater detail when describing the implementation.

3.3 Evaluation and Cast Reduction

Interpreting an LDLC program (assuming successful type checking) implies evaluation of the expression at the program's entry point.[1] In general, this involves recursively evaluating sub-expressions. Assuming a correct program with finite execution time, interpretation yields a final value. In contrast, CCLDLC needs to handle casts and types on top of this.

- As in LDLC, interpretation requires the evaluation of expression M to a value V in environment ρ as defined in the last section: $\rho \vdash M \Downarrow V$
- Additionally, the evaluation of type A to *head normal form* (HNF) A°: $\rho \vdash' A \Downarrow A^\circ$
- A program in CCLDLC returns a value, too. Hence, reducing a cast from a cast expression to an ordinary value is necessary: $(M : A \Rightarrow B) \Downarrow V$

[1] Reminder: in LDGV this is the expression that was bound to variable `main`.

We now introduce additional evaluation rules, especially for handling cast expressions. Basically, such a *cast reduction* is written $(V : A^\circ \Rightarrow B^\circ) \Downarrow V'$, and turns a value V of type A° into value V' of type B°. In contrast, when the evaluation reaches **blame** , it simply fails. This failure needs to be handled by the implementation. The whole set of cast reduction rules as given by Fu et al. [2021] is listed in figure 8.

Figure 6 did describe *Cast Calculus* extensions to the language's expressions. Here, the cast expression $(M : A \Rightarrow B)$ consists of an expression M of type A and a target type B. The CAST-REDUCE rule describes the procedure for evaluating the expression's components before cast reduction. In environment ρ, the embedded expression is evaluated $(M \downarrow V)$ resulting in a value. On successful cast reduction this is also the whole reduction's result. As for the types A and B, we do evaluate them to *head normal form* $(A \downarrow A^\circ;\ B \downarrow B^\circ)$.

A lambda expression is in *head normal form* (HNF) if and only if it is of the form

$$\lambda x_1.\lambda x_2 \ldots \lambda x_n.(v\ M_1\ M_2\ \ldots\ M_m)$$

where $n, m \geq 0$;

v is a variable (x_i), a data object, for a built-in function;

and $(v\ M_1\ M_2\ \ldots\ M_p)$ is not a redex for any $p \leq m$.

— Peyton Jones [1987], p. 199

Using the types from CCLDLC, we can derive all forms of HNF types by type evaluation. For a corresponding list see figure 9. After evaluating the components of the cast expression, the actual cast reduction is invoked, resulting in a single value.

Some rules make use of *type matching* from HNF types to ground types

Cast-Reduce
$$\frac{\rho \vdash M \downarrow V \qquad \rho \vdash' A \downarrow A^\circ \qquad \rho \vdash' B \downarrow B^\circ \qquad (V : A^\circ \Rightarrow B^\circ) \Downarrow V'}{\rho \vdash (M : A \Rightarrow B) \Downarrow V'}$$

Cast-Is-Value
$$\frac{V = (V' : A^\circ \Rightarrow B^\circ)}{V \Downarrow V}$$

Cast-Dyn-Dyn
$$\frac{A^\circ \leq \star}{(V : A^\circ \Rightarrow \star) \Downarrow V}$$

Cast-Sub
$$\frac{G \leq H}{(V : G \Rightarrow H) \Downarrow V}$$

Cast-Fail
$$\frac{A^\circ \rhd G \qquad B^\circ \rhd H \qquad G \not\leq H}{(V : A^\circ \Rightarrow B^\circ) \Downarrow \mathbf{blame}}$$

Cast-Bot
$$(V : A^\circ \Rightarrow \bot) \Downarrow \mathbf{blame}$$

Cast-Collapse
$$\frac{G \leq H}{((V : G \Rightarrow \star) : \star \Rightarrow H) \Downarrow V}$$

Cast-Collide
$$\frac{G \not\leq H}{((V : G \Rightarrow \star) : \star \Rightarrow H) \Downarrow \mathbf{blame}}$$

Cast-Factor-Left
$$\frac{\begin{array}{c} A^\circ \rhd G \qquad A^\circ \neq G \\ (V : A^\circ \Rightarrow G) \Downarrow V' \end{array}}{(V : A^\circ \Rightarrow \star) \Downarrow (V' : G \Rightarrow \star)}$$

Cast-Factor-Right
$$\frac{\begin{array}{c} B^\circ \rhd H \qquad B^\circ \neq H \\ (V : \star \Rightarrow H) \Downarrow V' \\ (V' : H \Rightarrow B^\circ) \Downarrow V'' \end{array}}{(V : \star \Rightarrow B^\circ) \Downarrow V''}$$

Cast-Pair
$$\frac{\begin{array}{c} \rho \vdash' A \downarrow A^\circ \qquad \rho' \vdash' A' \downarrow A'^\circ \qquad (V : A^\circ \Rightarrow A'^\circ) \Downarrow V' \\ \rho, x = V \vdash' B \downarrow B^\circ \qquad \rho', x = V' \vdash' B' \downarrow B'^\circ \qquad (W : B^\circ \Rightarrow B'^\circ) = W' \end{array}}{(\langle V, W \rangle : (\rho, \Sigma(x : A)B) \Rightarrow (\rho', \Sigma(x : A')B')) \Downarrow \langle V', W' \rangle}$$

Cast-Function
$$\frac{\begin{array}{c} \rho_\circ \vdash M \downarrow (V : (\rho, \Pi(x : A)B) \Rightarrow (\rho', \Pi(x : A')B')) \qquad \rho_\circ \vdash N \downarrow W' \\ \rho \vdash' A \downarrow A^\circ \qquad \rho' \vdash' A' \downarrow A'^\circ \qquad (W' : A'^\circ \Rightarrow A^\circ) \Downarrow W \\ \rho', x = W' \vdash' B' \downarrow B'^\circ \qquad \rho, x = W \vdash' B \downarrow B^\circ \\ \rho_\circ \vdash (VW : B^\circ \Rightarrow B'^\circ) \downarrow U' \end{array}}{\rho_\circ \vdash MN \downarrow U'}$$

Figure 8: Rules for cast reduction ($\Downarrow$) in CCLDLC.

$$
\begin{aligned}
A^\circ ::= \bot \mid \star \mid \textbf{Unit} \mid L \mid \textbf{S}\{V : A^\circ\} \\
\mid \textbf{Nat} \mid \textbf{Int} \mid \textbf{Double} \\
\mid (\rho, \Pi(x:A)B) \mid (\rho, \Sigma(x:A)B)
\end{aligned} \tag{5}
$$

Figure 9: Head normal form (HNF) of CCLDLC types

$$
\begin{aligned}
\textbf{Unit} &\rhd \textbf{Unit} \\
L &\rhd L \\
\textbf{Nat} &\rhd \textbf{Nat} \\
\textbf{Int} &\rhd \textbf{Int} \\
\textbf{Double} &\rhd \textbf{Double} \\
(\rho, \Pi(x:A)B) &\rhd \Pi(x:\star)\star \\
(\rho, \Sigma(x:A)B) &\rhd \Sigma(x:\star)\star \\
\textbf{S}\{V : A\} &\rhd \textbf{S}\{V : A\}
\end{aligned}
$$

Figure 10: Type matching from HNF type A° to ground type G

using the $\rhd$ operator. These are Cast-Fail, Cast-Factor-Left and Cast-Factor-Right. Note that type matching does not work *from* ground types *to* HNF types since the latter are more general.

3.4 Program Structure

LDGV is built from of a number of modules. There is a clear hierarchy as seen in the dependency graph in figure 11. Each functionality encompasses an independent group of modules:

- Module `Parsing`, including tokenization
- Module `Typechecker`, including modules for typechecking expressions and another one for subtyping in particular.
- Module `Interpreter` with data types for environment and values.

Most modules depend on the `Syntax` module. It provides data types for language constructs like expressions, types and literals. When invoking the program's functionality, the relevant module functions are invoked in order, e.g. when running the interpreter it (1) parses the source code, (2) runs the type checker on the AST[2] and on success, it finally (3) interprets the AST's `main` function if available and returns its value.

Since the module groups are separated, only the Interpreter knows about values, though there is no dependency on the type checker. The interpreter for LDLC does not need to know about types, its only concern is the evaluation of expressions, i.e. after type checking, *type erasure* occurs, simplifying the implementation (see: Crary et al. [2002]).

For the cast calculus, most adaptions happen in the interpreter, implementing runtime casts. Parts of the aforementioned simplicity regarding type erasure have to be abandoned to introduce *type passing* wheresoever necessary.

But first things first, we implement preliminary language constructs, starting with the most basic module: In the `Syntax` module, we add a constructor for the cast $(M : A \Rightarrow B)$ to the data type for expressions, accepting an expression M and two types A and B. We also add two argumentless types: bottom type $\bot$ and dynamic type $\star$. In section 3.1 we mentioned the top type $\top$, which we do not implement, because there actually is no expression of type $\top$. When declaring a function taking an argument of arbitary type, the programmer will use the dynamic type $\lambda(x : \star)M$. This implies that variable x is handled at runtime. A function $\lambda(x : \top)M$ is only sound if $M =$ **blame** , which is not necessary since we archive the same outcome by unfolding.

On the same note, there is no expression for explicitly invoking **blame**. As it indicates a cast failure, the implementation will handle blame. Since

[2]AST: Abstract Syntax Tree; a graph representation of an expression or program.

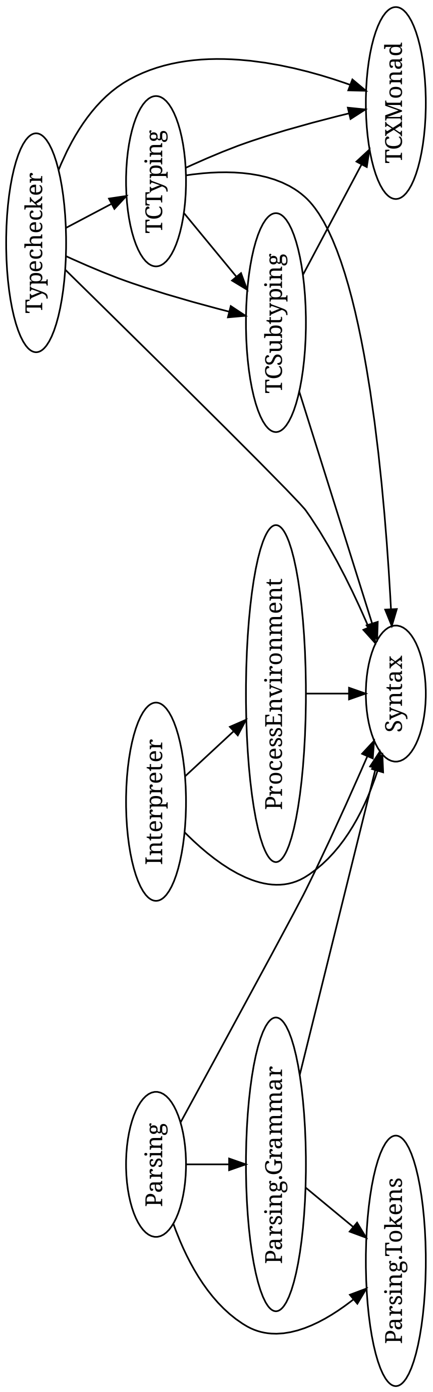

Figure 11: Module dependencies of LDGV, simplified without utility modules

Listing 3.1: LDGV: New language constructs for CCLDLC

```
Bot              -- bottom type
*                -- dynamic type
exp  : A => B    -- cast expression
```

the programmer is not involved in the specification of type casting rules there is no need to allow its usage in the language.

The modules connected to `Parsing` need to reflect these modifications. We introduce tokens for the dynamic type and the bottom type and extend the grammar accordingly. We also add an expression for casts. See listing 3.1.

The module group forming the `Typechecker` needs to handle the new types. $\perp$ is made a subtype to everything and the subtyping rules for $\star$ are implemented *without introducing transitivity*. Even though we permit a cast from any type A to $\star$ and $\star$ to any type B this does tell us nothing about subtyping relations between A and B. Finally, we check that for an expression $(M : A \Rightarrow B)$ the encapsulated expression M is of type A while the cast itself is handled at runtime by the interpreter.

The `Interpreter` needs most adjustments. The evaluation rules from section 3.3 are applied at runtime. Since there is no direct dependency between typechecker and interpreter, we also exchange no data, i.e. the results of typechecking are disregarded by the interpreter to avoid entanglement. While the typechecker asserts type safety within cast expressions, the actual cast reduction is always deferred to the interpretation stage.

We extend the `ProcessEnvironment` module in a way that introduces the necessary types, i.e. *HNF types* and *ground types*. In the `Interpreter` module itself, we add functions for type evaluation (to already existing support for expressions and statements) and finally cast reduction. The resulting values are handled as before, making this a conservative extension. The next chapter details these translations from formal rules to actual code.

4 Implementation

The actual implementation uses the Haskell programming language and related tools. There are adjustments in all three module groups as classified in section 3.4. Furthermore, we extend the data types in the fundamental `Syntax` module with additional expressions and types from section 3.1, thus making these terms known to the remaining program.

Listing 4.1: Haskell: Extensions to `Syntax.hs`

```
data Exp = ...
  | Cast Exp Type Type   -- M : A => B

data Type = ...
  | TBot   -- bottom type
  | TDyn   -- dynamic type *
```

4.1 Parsing

As usual, there is a tokenization step before the actual parsing, turning a string into tokens. For this purpose, we use the tool *Alex: A lexical analyser generator for Haskell*[1], maintained by Simon Marlow, as it nicely integrates with Haskell. We add tokens for the bottom type and the double arrow used in cast expressions. In our implementation, we use `*` as identifier for the dynamic type. There already exists a corresponding token since it was used for multiplication in the base language.

[1] `https://www.haskell.org/alex/`

Listing 4.2: Haskell: Additional tokens (`Parsing/Tokens.x`)

```
data Token = ...
  | TBot
  | DoubleArrow

tokens :-
  ...
  Bot      { tok $ const TBot }
  "_|_"    { tok $ const TBot }
  "=>"     { tok $ const DoubleArrow }
```

For parsing, we use *Happy: The Parser Generator for Haskell*[2], also maintained by Simon Marlow. Parsing is slightly more complicated since we need to add the dynamic type $\star$, bottom type and the cast expression $M : A \Rightarrow B$.

Listing 4.3: Haskell: Extensions to `Parsing/Grammar.y`

```
%token
  ...
  Bot    { T _ T.TBot }
  '=>'   { T _ T.DoubleArrow }

Exp = ...
  | Exp ':' Typ '=>' Typ  { Cast $1 $3 $5 }

ATyp = ...
  | '*'  { TDyn }
  | Bot  { TBot }
```

4.2 Typechecker

Since terms involving the newly introduced dynamic type are handled at runtime there is no need for extensive changes to the typechecker. Nevertheless,

[2]`https://www.haskell.org/happy/`

we need to integrate the new syntax. Otherwise, typechecking would fail, not because of type errors but out of "ignorance".

4.2.1 Type Synthesis

In general, type synthesis $\Gamma \vdash M \Rightarrow A; \Delta$ takes environment Γ and expression M to synthesize its type A and the environment after Δ. We adjust the corresponding function for the following expressions: (1) application, (2) pair construction and for built-in functions (3) `fst` and (4) `snd` (both eliminating pairs, yielding the first or second element). In code, it is complemented by a monad transformer, as seen in its annotation in listing 4.4, though most of its components are not relevant here.

For application $M\ N$, after synthesizing $\Gamma \vdash M \Rightarrow A; \Delta$ and if $A = \star$, we assert that expression N also has dynamic type $\star$. For this purpose, we extend the matching when handling applications inside the `tySynth` function, see listing 4.5.

An expression involving pairs is "**let** <x, y> = M **in** N", where M has a pair type. First, the components of M are bound to x and y, finally evaluating N. For typecheckig, we synthesize type A from M. Afterwards, we synthesize type B from N after extending the environment: $\Delta_A, x : \star, y : \star \vdash N \Rightarrow B$ results in the whole expression's type. This is depicted in listing 4.6.

The built-in functions `fst` and `snd` simply return the dynamic type $\star$ upon type synthesis since a pair type $\Sigma(x : \star)\star$ implies that both components are of type $\star$, too.

4.2.2 Subtyping

Subtyping regarding the dynamic type happens in the context of **case**, used for eliminating label types. For dynamic type $\star$ we need to unfold such **case** constructs by checking that each type in its label matching unfolds to $\star$. For this purpose, we extend the existing `unfold` function as seen in

Listing 4.4: Haskell: `tySynth` function type annotation

```haskell
tySynth :: TEnv -> Exp -> TCM (Type, TEnv)

type TCM a =
  WriterT [Constraint] (ExceptT String (State Caches)) a

data Caches = Caches{subCache::Cache, eqvCache::Cache}
type Cache = [(Type, Type)]

data Constraint = Type :<: Type
```

Listing 4.5: Haskell: Application type synthesis (`TCTyping.hs`)

```haskell
tySynth :: TEnv -> Exp -> TCM (Type, TEnv)
tySynth te (App e1 e2) = do
  (tf, te1) <- tySynth te e1
  tfu <- unfold te1 tf
  case tfu of
    -- ...
    TDyn -> do
      te2 <- tyCheck te1 e2 TDyn
      return (TDyn, te2)
```

Listing 4.6: Haskell: Pair type synthesis (`TCTyping.hs`)

```haskell
tySynth te (LetPair x y e1 e2) = do
  (tp, te1) <- tySynth te e1
  tpu <- unfold te1 tp
  case tpu of
    TPair {} -> -- ...
    TDyn -> tySynth
      ((y, (Many, TDyn)) : (x, (Many, TDyn)) : te1)
      e2
```

Listing 4.7: Haskell: Unfolding to $\star$ in case terms (`TCSubtyping.hs`)

```haskell
unfold :: TEnv -> Type -> TCM Type
unfold tenv (TCase val@(Var x) cases) = do
  tyx <- varlookupUnfolding x tenv
  let f :: (String, Type) -> TCM Type
      f (lab, labtyp) = unfold
        (("*unfold*", TEqn val (Lit $ LLab lab) tyx) : tenv)
        labtyp
  results <- case tyx of
    TLab {} -> -- ...
    TDyn -> mapM f cases
  -- if all results are TDyn, return TDyn
```

listing 4.7. If this succeeds, then type checking accepts the outcome, deferring the actual cast to runtime.

Furthermore, we have to consider casts in context of **case** types. Given a type **case** $(x : D \Rightarrow L)\ \{\overline{\ell : A_\ell}^{\ell \in L}\}$ with variable x and label type L we need to implement subtyping in relation to other types. The following rules describe these relations when checking if **case** is a subtype or supertype.

A-Sub-Case-CL

$$\frac{L' = \{\ell \in L \mid \Gamma \Vdash_G \{\ell\} \leq D\} \qquad (\forall \ell \in L') \quad D_\ell = \mathit{cast}(\{\ell\} \Rightarrow D) \quad \Gamma, x : D_\ell, \Delta \Vdash A_\ell \leq B}{\Gamma, x : D, \Delta \Vdash \textbf{case}\ (x : D \Rightarrow L)\{\overline{\ell : A_\ell}^{\ell \in L}\} \leq B}$$

A-Sub-Case-CR

$$\frac{L' = \{\ell \in L \mid \Gamma \Vdash_G \{\ell\} \leq D\} \qquad (\forall \ell \in L') \quad D_\ell = \mathit{cast}(\{\ell\} \Rightarrow D) \quad \Gamma, x : D_\ell, \Delta \Vdash A \leq B_\ell}{\Gamma, x : D, \Delta \Vdash A \leq \textbf{case}\ (x : D \Rightarrow L)\{\overline{\ell : B_\ell}^{\ell \in L}\}}$$

$$\mathit{cast}(\{\ell\} \Rightarrow B) = \begin{cases} \{\ell\} & \text{if } (V : A \Rightarrow B) \Downarrow W \\ \bot & \text{if } (V : A \Rightarrow B) \Downarrow \textbf{blame} \end{cases}$$

Listing 4.8: Haskell: Subtyping for casts in case terms (`TCSubtyping.hs`)

```haskell
subtype' tenv
  (TCase (Cast (Var x) t1 (TLab ls2)) cases)
  tyy2 =
    case lookup x tenv of
      Just (_,TLab [l]) -> do
        t <- lablookup l cases
        subtype tenv t tyy2
      _ -> do  -- A-Sub-Case-CL
        let ls' = case t1 of
              TDyn    -> ls2
              TLab ls -> filter (`elem` ls) ls2
              _       -> []
            subtypeCast l = do
              let tenv' = (x,(Many,TLab [l])) : tenv
              cty <- lablookup l cases
              subtype tenv' cty tyy2
        results <- mapM subtypeCast ls'
        return $ foldr1 klub results
```

When implementing these rules, we assume that type D is either a label type or the dynamic type. This is because in *Gradual LDLC*—as indicated by "$\Vdash_G$"—there are subtyping rules for these cases. For label type L we can establish subtyping if $L \subseteq D$ and for dynamic type $D = \star$ every type A is a subtype of D. Function *cast* checks if a type cast is possible, returning a singleton type on success. Since a successful subtyping check for $\{\ell\} \leq D$ already indicates a successful cast $\{\ell \Rightarrow D\}$ we omit an explicit implementation of the *cast* function.

Furthermore, we implement additional logic regarding these rules: If typing environment Γ already contains a binding to a singleton label $x : \{\ell\}$ then this is utilized for resolving the **case** type, directly checking subtyping $A_\ell \leq B$ or $A \leq B_\ell$ respectively. Otherwise, variable x would serve no purpose. Deliberately ignoring its binding would render aforementioned logic redundant, hence the **case** type would not be resolved. The corresponding code for A-Sub-Case-CL is given in listing 4.8. Rule A-Sub-Case-CL is nearly identical.

4.3 Interpreter

In basic LDGV, types are not relevant in context of evaluation. The interpreter works on the assumption that prior type checking did succeed. In our extension, cast expressions are always handled at runtime, even if both types in a cast $M : A \Rightarrow B$ are not dynamic ($A \neq \star, B \neq \star$). For correct handling of the dynamic type and type reduction of cast expressions we introduce new data types and extend the existing data type for values. Finally, we implement the rules of section 3.3.

In chapter 3 we also listed rules for handling singleton types $\mathbf{S}\{A\}$ as given in the original paper. The implementation does not support explicit singleton types in its grammar. Therefore, from this point onwards, singleton types will not be considered.

Listing 4.9: Haskell: HNF type definition (`ProcessEnvironment.hs`)

```haskell
data NFType
  = NFBot
  | NFDyn
  | NFFunc FuncType
  | NFPair FuncType
  | NFGType GType

data FuncType = FuncType PEnv String Type Type
```

4.3.1 Type Passing

Since type reduction requires *head normal form* we introduce an explicit data type `NFType` in listing 4.9. Label type L is implemented as a set of strings. Similarities between function type $(\rho, \Pi(x : A)B)$ and pair type $(\rho, \Sigma(x : A)B)$ are depicted using the `FuncType` data type.

The constructor `NFGType` is a special case. It takes an argument of type `GType`—the data type implementing *ground types*—which we define in listing 4.10. This type dependency depicts that every ground type is also in head normal form—but not the other way around—while making comparison operations between `NFType` and `GType` possible. At the same time, defining ground types as a separate data type allows for more precise function signatures. Even so, we can not perfectly model the relationship since Haskell itself is not dependently typed. A function type $(\cdot, \Pi(x : \star)\star)$ can be written as `NFGType GFunc "x"` or as `NFFunc (FuncType [] "x" TDyn TDyn)`. The latter is semantically correct but must not occur in most contexts. We have to resort to helper function `equalsType` and additional logic to resolve such cases.

Subtyping checks are implemented in listing 4.11 introducing typeclass `Subtypeable` with instances for ground types `GType`—mostly with trivial definitions—and partially for HNF types `NFType`. The latter is solely necessary

Listing 4.10: Haskell: Ground type definition (`ProcessEnvironment.hs`)

```haskell
data GType
  = GUnit
  | GLabel LabelType
  | GFunc String
  | GPair String
  | GNat
  | GInt
  | GDouble
  | GString

type Label = String
type LabelType = Set Label

equalsType :: NFType -> GType -> Bool
equalsType (NFFunc (FuncType _ _ s TDyn TDyn)) (GFunc s') =
  s == s'
equalsType (NFPair (FuncType _ _ s TDyn TDyn)) (GPair s') =
  s == s'
equalsType (NFGType gt1) gt2 = gt1 == gt2
equalsType _ _ = False
```

Listing 4.11: Haskell: Subtypeable typeclass (`ProcessEnvironment.hs`)

```
class Subtypeable t where
  isSubtypeOf :: t -> t -> Bool

instance Subtypeable GType where
  isSubtypeOf GUnit GUnit = True
  isSubtypeOf (GLabel ls1) (GLabel ls2) =
    ls1 `Set.isSubsetOf` ls2
  isSubtypeOf (GFunc _) (GFunc _) = True
  isSubtypeOf (GPair _) (GPair _) = True
  isSubtypeOf GNat GNat = True
  isSubtypeOf GInt GInt = True
  isSubtypeOf GDouble GDouble = True
  isSubtypeOf GString GString = True
  isSubtypeOf _ _ = False

instance Subtypeable NFType where
  isSubtypeOf NFBot _ = True
  isSubtypeOf NFDyn NFDyn = True
  -- ...
  isSubtypeOf _ _ = False
```

for implementing the Cast-Dyn-Dyn rule.

We also extend the value type according to section 3.1. We add function closures, including environment ρ, variable name and the expression in the function's body. Furthermore function casts and the dynamic wrapper $(V : G \Rightarrow \star)$ (see listing 4.12).

We use these new and extended data types to handle CCLDLC's types during the interpretation state, especially for type inference in cast expressions. At last, we need a mapping to HNF types used in type casts. We introduce the `evalType` function in listing 4.13 to evaluate types to their HNF form. Mostly, this evaluation is trivial. For function and pair types, the environment is made part of the type. Most involved is evaluation of the **case** x $\{\ell_i : A_i\}$

Listing 4.12: Haskell: Value type extensions (`ProcessEnvironment.hs`)

```haskell
data Value = -- ...
  | VFunc PEnv String Exp
  | VFuncCast Value FuncType FuncType
  | VDynCast Value GType
```

type since case types are never in HNF. We match label $x \downarrow \ell_j$ on the type mapping, finally evaluating $A_j \downarrow A^\circ$. If possible, we evaluate to ground types and wrap them using the `NFGType` constructor.

4.3.2 Cast Reduction

The goal of type inference at runtime is to either reduce casts in a way that evaluation halts at a value, or to abort evaluation with **blame** . In our implementation, the latter throws an exception. During this section we describe the implementation of the rules stated in figure 7.

We extended the language with cast expression $M : A \Rightarrow B$. However, a cast reduction has to have form $V : A^\circ \Rightarrow B^\circ$. Hence, we use rule Cast-Reduce to evaluate expression M to value V and types A and B to HNF A° and B° respectively. For that purpose, we extend the existing `eval` function according to listing 4.14. This function implements the interpreter's main logic, mapping all the language's expressions (see figure 1) to values. The `InterpretM` monad from the function's signature represents evaluation environment ρ, providing the *Reader* monad's interface.[3]

Cast reduction $V : A^\circ \Rightarrow B^\circ \Downarrow V'$ is implemented in the `reduceCast` function. Its type signature matches the term as it requires a value and two types in HNF. However, upon failure it will not return **blame** directly but encapsulates its result into a `Maybe` type. Then, the calling function needs to call **blame** upon an empty result (e. g. see listing 4.14).

[3]`https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-Reader.html`

Listing 4.13: Haskell: Type evaluation (`Interpreter.hs`)

```haskell
evalType :: Type -> InterpretM NFType
evalType = \case
  TUnit   -> return $ NFGType GUnit
  TNat    -> return $ NFGType GNat
  TInt    -> return $ NFGType GInt
  TDouble -> return $ NFGType GDouble
  TString -> return $ NFGType GString
  TBot    -> return NFBot
  TDyn    -> return NFDyn
  TLab ls -> return $ NFGType $ GLabel ls
  TFun  _ _ TDyn TDyn -> return $ NFGType GFunc
  TFun  _ s t1 t2 -> do
    env <- ask
    return $ NFFunc $ FuncType env s t1 t2
  TPair _ _ TDyn TDyn -> return $ NFGType GPair
  TPair _ s t1 t2 -> do
    env <- ask
    return $ NFPair $ FuncType env s t1 t2
  TCase exp labels -> interpret' exp >>= \(VLabel l) ->
    let entry = find (\(l', _) -> l == l') labels
    in maybe (return NFBot) (evalType . snd) entry
```

Listing 4.14: Haskell: Cast expression evaluation (`Interpreter.hs`)

```haskell
eval :: Exp -> InterpretM Value
eval cast@(Cast e t1 t2) = do
  v <- interpret' e
  nft1 <- evalType t1
  nft2 <- evalType t2
  maybe (blame cast) return (reduceCast v nft1 nft2)
```

Listing 4.15: Haskell: Rule CAST-IS-VALUE (`Interpreter.hs`)

```haskell
reduceCast :: Value -> NFType -> NFType -> Maybe Value
reduceCast v t1 t2 = castIsValue v t1 t2
                 <|> reduceCast' v t1 t2

castIsValue :: Value -> NFType -> NFType -> Maybe Value
castIsValue v (NFGType gt) NFDyn = Just $ VDynCast v gt
castIsValue v (NFFunc ft1) (NFFunc ft2) =
  Just $ VFuncCast v ft1 ft2
castIsValue v (NFFunc ft1) (NFGType (GFunc y)) =
  Just $ VFuncCast v
    ft1 (FuncType [] y TDyn TDyn)
castIsValue v (NFGType (GFunc x)) (NFFunc ft2) =
  Just $ VFuncCast v
    (FuncType [] x TDyn TDyn) ft2
castIsValue v (NFGType (GFunc x)) (NFGType (GFunc y)) =
  Just $ VFuncCast v
    (FuncType [] x TDyn TDyn) (FuncType [] y TDyn TDyn)
castIsValue _ _ _ = Nothing
```

The basic CAST-IS-VALUE rule returns $V : A^\circ \Rightarrow B^\circ$ as it is if it already forms a value. This occurs for $A^\circ = G, B^\circ = \star$ (`VDynCast`) or for function casts, i.e. $A^\circ = (\rho, \Pi(x : A)A'), B^\circ = (\rho, \Pi(x : B)B')$ (`VFuncCast`). When implementing the latter we have to consider the ground type `GFunc` that is also a valid function type but is on a different level in the type hierarchy. Therefore, more code is necessary to handle conversion. If the CAST-IS-VALUE rule does not apply we check the remaining rules via distinct function `reduceCast'` as *alternative* (indicated by the `<|>` operator). The code is given in listing 4.15.

For some rules we match HNF types to ground types as defined in figure 10. The actual code of listing 4.16 is more concise than the definition because of the way we encapsulated `GType` and since we do not consider singleton types.

Listing 4.16: Haskell: Type matching $A^\circ \rhd G$ (`Interpreter.hs`)

```
matchType :: NFType -> Maybe GType
matchType = \case
  NFFunc (FuncType _ _ x _ _) -> Just $ GFunc x
  NFPair (FuncType _ _ x _ _) -> Just $ GPair x
  NFGType gt -> Just gt
  _ -> Nothing
```

Listing 4.17: Haskell: Rules Cast-Dyn-Dyn and Factor-Left (`Interpreter.hs`)

```
reduceCast' v t NFDyn =
  if t `isSubtypeOf` NFDyn
  then Just v --Cast-Dyn-Dyn
  else do
    gt <- matchType t
    v' <- reduceCast v t (NFGType gt)
    Just $ VDynCast v' gt
```

The remaining rules necessary for a full implementation of CCLDLC , except these for function and pair handling, are handled by the `reduceCast'` function we describe in the next paragraphs.

Rules Cast-Dyn-Dyn and Factor-Left are similar such that they presume a cast *to* dynamic type $\star$. Hence, we implement them together as in listing 4.17. Currently, the only subtypes of $\star$ are $\star$ itself and bottom type $\bot$ which is expressed using typeclass `Subtypeable` (listing 4.11). If the given type is no subtype of $\star$ we apply Factor-Left as is. After type matching $A^\circ \rhd G$ we do not implement the check that $A^\circ \neq G$. Given the function's arguments this would imply a cast $V : G \Rightarrow \star$ that would have been handled by an earlier call to `castIsValue`.

There is no type that is a subtype of the bottom type $\bot$. Hence, a cast to $\bot$ must always results in blame which is expressed in rule Cast-Bot

Listing 4.18: Haskell: Rule Cast-Bot (`Interpreter.hs`)

```
reduceCast' _ _ NFBot = Nothing
```

Listing 4.19: Haskell: Rules Cast-Collapse and Cast-Collide (`Interpreter.hs`)

```
reduceCast' (VDynCast v gt1) NFDyn (NFGType gt2) =
  if gt1 `isSubtypeOf` gt2 then Just v else Nothing
```

(listing 4.18).

It is straightforward to implement rules Cast-Collapse and Cast-Collide. Even though dynamic type $\star$ does not support transitivity w. r. t. subtyping, these rules enable transitivity for ground types only. The code is given in listing 4.19.

Rule Factor-Right describes a cast from $\star$ to another type (listing 4.20). Its utilization of *type matching* $\triangleright$ makes it similar to Factor-Left. A failure results in **blame** since rules with similar structure, namely Cast-Collapse and Cast-Collide, were handled before by pattern matching over more specific arguments.

Finally, we give a simple implementation for rule Cast-Sub which simply checks subtyping between given ground types. On failure, this simply returns `Nothing` and thus implies **blame** without any additional checks (listing 4.21). Also, we completely omit rule Cast-Fail. This is because it is the last rule in context of `reduceCast'`. There are no cases left to examine so any cast not yet covered must fail.

The last two rules are about type reductions in context of pairs and functions. First, we implemnt Cast-Pair. For this purpose we make adjustments to the `eval` function in listing 4.14 which returns monad `InterpretM`. This is necessary since we need to consider the expression's environment ρ. The type annotation of function `reduceCast` does not accept arguments

Listing 4.20: Haskell: Rule FACTOR-RIGHT (`Interpreter.hs`)

```haskell
reduceCast' v NFDyn t = do
  gt <- matchType t
  let nfgt = NFGType gt
  if not (t `equalsType` gt) then do
    v'  <- reduceCast v NFDyn nfgt
    v'' <- reduceCast v' nfgt t
    Just v''
  else
    Nothing
```

Listing 4.21: Haskell: Rule CAST-SUB (`Interpreter.hs`)

```haskell
reduceCast' v (NFGType gt1) (NFGType gt2) =
  if gt1 `isSubtypeOf` gt2 then Just v else Nothing
```

capable of representing the environment. After evaluating to $V : A^\circ \Rightarrow B^\circ$ we catch pair values by pattern matching and pass the cast term to function `reducePairCast` as seen in listing 4.22. Again, we need a helper function to manage overlapping types. With `toNFPair` we intercept ground types, using type constructor `NFPair` instead of `NFGType`. This is necessary since `reducePairCast` only matches the former. Alternatively, one might match against all combinations of both type constructors as in function `castIsValue` from listing 4.15. The actual cast reduction consists of simply reducing both of its components. When reducing the second component, we extend the respective environments with the first components' values (`xEnv`).

Finally, we implement CAST-FUNCTION by—again—extending function `eval` for application $M\ N$, see listing 4.23. Formerly, this did evaluate expressions $M \downarrow V$ and $N \downarrow W$ separately to values. Given V is a function $(\rho, \lambda x.M')$ it extends environment ρ with $x = W$, then evaluating M' in its context. Otherwise, it would fail. For our extension we introduce function

Listing 4.22: Haskell: Rule CAST-PAIR (`Interpreter.hs`)

```
eval cast@(Cast e t1 t2) = do
  v    <- interpret' e
  nft1 <- evalType t1
  nft2 <- evalType t2
  case v of
    -- catch pair values
    VPair {} -> do
      v' <- lift $ reducePairCast
        v (toNFPair nft1) (toNFPair nft2)
      maybe (blame cast) return v'
    -- as before
    _ -> maybe (blame cast) return (reduceCast v nft1 nft2)

toNFPair (NFGType (GPair s)) =
  NFPair (FuncType [] [] s TDyn TDyn)
toNFPair t = t

reducePairCast
  (VPair v w)
  (NFPair (FuncType env  s  t1  t2))
  (NFPair (FuncType env' s' t1' t2')) = do
    mv' <- reduceComponent v (env, t1) (env', t1')
    case mv' of
      Nothing -> return Nothing -- first reduce failed
      Just v' -> do
        let xEnv  = (s,  v)  : env
            xEnv' = (s', v') : env'
        mw' <- reduceComponent w (xEnv, t2) (xEnv', t2')
        return $ liftM2 VPair mv' mw'

reduceComponent v (env, t) (env', t') = do
  nft  <- R.runReaderT (evalType t)  env,
  nft' <- R.runReaderT (evalType t') env'
  return $ reduceCast v nft nft'
```

Listing 4.23: Haskell: Rule CAST-FUNCTION (`Interpreter.hs`)

```
interpretApp :: Value -> Value -> InterpretM Value
interpretApp (VFuncCast v
  (FuncType env  s  t1  t2)
  (FuncType env' s' t1' t2')) w' = do
    env0 <- ask
    let interpretAppCast = do
      nft1  <- R.runReaderT (evalType t1)  env
      nft1' <- R.runReaderT (evalType t1') env'
      w  <- maybe blame return (reduceCast w' nft1' nft1)
      nft2' <- R.runReaderT (evalType t2') ((s', w') : env)
      nft2  <- R.runReaderT (evalType t2)  ((s,  w)  : env)
      u  <-    R.runReaderT (interpretApp v w) env0
      u' <- maybe blame return (reduceCast u nft2 nft2')
      return u'
    lift interpretAppCast
```

`interpretApp` taking values V as W as arguments in listing 4.23. This allows for more involved functionality by matching over its arguments' types. Apart from that, the code mostly reflects the rule, using `runReaderT` from the *Reader* monad to respectively pass the correct environment.

4.4 Function Cast Example

To give an intuition about the reduction of cast expressions regarding the rules and implementation we show a non-trivial example. We evaluate a nested lambda expression taking two arguments x, y. While x expects type **Bool** , the type of y is dynamic. In the function's body, y depends on the value of x, hence a cast dependent on the value of x is written out. If $x = \ell_F$ then y is a function mapping one boolean argument to a boolean result, then applying x. Else, for $x = \ell_T$, y is a function expecting *two* arguments, applying x two successive times. The function is given in CCLDLC formulation (equation 6) as well as in actual code (listing 4.24).

$$
\begin{aligned}
f = &\lambda(x : \textbf{ Bool }).\lambda(y : \star).\ \textbf{case}\ x\{ \\
&\qquad \ell_F : (y : \star \Rightarrow \Pi(a : \textbf{ Bool })\ \textbf{Bool }) \ x, \\
&\qquad \ell_T : (x : \star \Rightarrow \Pi(a : \textbf{ Bool })\Pi(b : \textbf{ Bool })\ \textbf{Bool })\ x\ x\} \\
&:\Pi(x : \textbf{ Bool }).\Pi(y : \star).\ \textbf{Bool}
\end{aligned}
\tag{6}
$$

Listing 4.24: LDGV: CCLDGV example "dependent function cast"

```
val f = fn(x: Bool) fn(y: *) case x {
             'F: (y: * => (a:Bool) -> Bool) x,
             'T: (y: * => (a:Bool) -> (b:Bool) -> Bool) x x}
```

By applying label ℓ_F (representing boolean *False*) and boolean function not $= \lambda(x : \textbf{ Bool })\ \textbf{case}\ x\{\ell_T : \ell_F, \ell_F : \ell_T\}$ we evaluate the whole expression to value ℓ_T. This is presented in a step-by-step manner, starting with an application and an explicit dynamic cast of function not. For brevity, we omit the environment ρ inside equations.

$$f\ \ell_F\ \ (\text{ not } : (a : \textbf{ Bool }) \to \textbf{ Bool } \Rightarrow \star) \tag{7}$$

$$(f\ \ell_F)\ (\text{ not } : (a : \textbf{ Bool }) \to \textbf{ Bool } \Rightarrow \star) \tag{8}$$

First, we evaluate the left part of the application, which is an application itself: $f\ \ell_F$. This part does not involve the *Cast Calculus*, simply binding $x = \ell_F$ in the function's environment ρ, returning $f' := \lambda(y : \star)\ \textbf{case}\ x\{\dots\}$. Next, we evaluate the right side's cast expression using rule Factor-Left (13) which enfolds Cast-Is-Value (11). This results in a value $v' := (V : G \Rightarrow \star)$ where V is a function cast and G is function ground type $\Pi(a : \star)\star$.

$$\begin{array}{rr}
\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \rhd \underline{\Pi(a:\star)\star} & (9)\\ \hline
\text{not}\ :\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \Rightarrow \underline{\Pi(a:\star)\star} & (10)\\
\Downarrow (\ \text{not}\ :\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \Rightarrow \Pi(a:\star)\star) & (11)\\ \hline
\text{not}\ :\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \Rightarrow \star & (12)\\
\Downarrow ((\ \text{not}\ :\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \Rightarrow \Pi(a:\star)\star) : \underline{\Pi(a:\star)\star} \Rightarrow \star) & (13)\\
=(\ V \qquad\qquad\qquad\qquad\qquad\qquad\qquad : G \qquad\quad \Rightarrow \star) =: v' & (14)
\end{array}$$

Now, we evaluate new-formed expression $f'\ v'$, again an application. Evaluating the body of f', the **case** term matches to label ℓ_F. Substituting $f'[v'/y]$ results in cast expression ($v' : \star \Rightarrow \Pi(a:$ **Bool**) **Bool**). Since the right part of the cast is not a ground type we need to apply Factor-Right (21), including Cast-Collapse (17) and Cast-Is-Value (19) for intermediate values. Invoking this cast reduction results in a function cast value $v'' := (V : \Pi(a:\star)\star \Rightarrow \Pi(a:$ **Bool**) **Bool**) with V being a function cast, too (the same as above).

$$\begin{array}{rr}
\Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ \rhd \underline{\Pi(a:\star)\star} & (15)\\ \hline
v' : \star \Rightarrow \underline{\Pi(a:\star)\star} & (16)\\
=(V : \underline{\Pi(a:\star)\star} \Rightarrow \star) : \star \Rightarrow \underline{\Pi(a:\star)\star} \Downarrow V & (17)\\ \hline
V : \underline{\Pi(a:\star)\star} \Rightarrow \Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool} & (18)\\
\Downarrow (V : \Pi(a:\star)\star \Rightarrow \Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ ) & (19)\\ \hline
v' : \star \Rightarrow \Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool} & (20)\\
\Downarrow (V : \Pi(a:\star)\star \Rightarrow \Pi(a:\ \mathbf{Bool}\ )\ \mathbf{Bool}\ ) =: v'' & (21)
\end{array}$$

Finally, we evaluate the full expression in the example function's false branch (ℓ_F). We substitue the cast of variable y with v'' and variable x with

ℓ_F. What remains is application $v'' \ \ell_F$. Since v'' is a function cast value we use rule Cast-Function (29). This includes reduction of application $V\ (\ell_F : \mathbf{Bool} \Rightarrow \star)$ which we need to solve first, using Cast-Function, too (26), as well as Cast-Collapse (22) (30).

$$\begin{array}{lr}
\quad (\ell_F : \mathbf{Bool} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool} \Downarrow \ell_F & (22) \\ \hline
\quad V \qquad\qquad\qquad\qquad\qquad\qquad (\ell_F : \mathbf{Bool} \Rightarrow \star) & (23) \\
= (\ \text{not}\ : \Pi(a : \mathbf{Bool})\ \mathbf{Bool} \Rightarrow \Pi(a : \star)\star)\ (\ell_F : \mathbf{Bool} \Rightarrow \star) & (24) \\
\Downarrow (\ \text{not}\ \ell_F) : \mathbf{Bool} \Rightarrow \star & (25) \\
\downarrow \ \ \ell_T : \mathbf{Bool} \Rightarrow \star & (26) \\
\Downarrow (\ \ell_T : \mathbf{Bool} \Rightarrow \star) & (27) \\ \hline
\quad (\ V\ (\ell_F : \mathbf{Bool} \Rightarrow \star)) : \star \Rightarrow \mathbf{Bool} & (28) \\
\downarrow (\ \ell_T : \mathbf{Bool} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool} & (29) \\
\Downarrow \ \ \ell_T & (30)
\end{array}$$

In conclusion, even the reductions for a relatively simple function cast are quite involved, requiring many rules and operations. Note that when full *gradual typing* is implemented, the explicit casts are not written out by the programmer.

4.5 Recursor Expression

The *Cast Calculus* revolves around type casts and the dynamic type. Until now, we did provide an exhaustive description of these newly introduced constructs, its rules and an implementation. In the same breath we will extend the natural recursor and prepare its integration into the *Cast Calculus*.

The current implementation of $\mathbf{rec}\ V\ M\ x.f.N$ is mostly as its description in section 2.1.2. Its syntax is `rec f (x.es) ez` with variable names `f,z` and

Listing 4.25: Haskell: Introducing value representation of **rec** (`ProcessEnvironment.hs`)

```
data Value = -- ...
     | VRec PEnv String String Exp Exp
```

Listing 4.26: Haskell: Evaluation of **rec** (`Interpreter.hs`)

```
eval Rec f x es ez = ask >>= \env ->
  return $ VRec env f x es ez

interpretApp rec@(VRec env f x es ez) (VInt n)
  | n == 0 = interpret' ez
  | n  > 0 = do
    let env' = extendEnv x (VInt (n-1))
               (extendEnv f rec env)
    R.local (const env') (interpret' es)
```

expressions `es,ez`. When recursively evaluating the expression, a natural number is necessary. This is implemented by application. When applying a number `n` to a correct **rec** expression r, evaluation yields a value: `r n` implements **rec** $V\ M\ x.f.N$ for $V = n$.

Additionally, we prepare evaluation of type **rec** $V\ A\ t.B$ to *head normal form*. Parsing term `natrec e { tz, tid.ts }` maps to a corresponding type representation `TNatRec` as defined in `Syntax.hs`. Whereas `e` is an expression, it has to evaluate to a number value. Next, evaluation proceeds similar to its expression counterpart seen above with base type `tz` and recursive evaluation for `es`. The code is given in listing 4.27.

In listing 4.28 we give examples for the usage of **rec** in LDGV. The first example, `notrec`, constructs successive applications of the not function, starting with value `'F` (representing False). The second example `sumrec` constructs a function taking $n + 1$ integer arguments, returning their sum. Both functions have in common that they expect a natural number for their

Listing 4.27: Haskell: Evaluation of **rec** type to HNF(`Interpreter.hs`)

```
evalType TNatRec e tz tid ts = do
  v <- interpret' e
  case v of
    VInt 0 -> evalType tz
    VInt n ->
      let lower = TNatRec (Lit $ LNat (n-1)) tz tid ts
      in R.local (extendEnv tid (VType lower)) (evalType ts)
```

Listing 4.28: LDGV: Example usage of **rec** expression and type

```
val notrec : (n: Nat) -> natrec n {Bool, A.A}
val notrec = rec f (x.nof(f x)) 'F

val sumrec : (n: Nat) -> natrec n {Int, A.Int -> A}
val sumrec = rec f
  (n1 . (fn (acc : Int) fn (x : Int) f n1 (acc + x)))
  (fn (acc: Int) acc)
```

first argument. This is sufficient to recursively evaluate the type using the `natrec` construct.

Currently, recursor expressions and types do not fully integrate into the *Cast Calculus*. Casting from and to recursor types remains complicated to write out. Furthermore, type evaluation does require knowledge of number value n which must be available at runtime. For more precise typing, introducing an additional type for restricted natural numbers $n \leq x$ would prove advantageous, too. In conclusion, while casts and elimination (via **case**) of labels provide elegant functionality, their number-dependent counterpart—**rec** types and expressions—still needs some work.

5 Conclusion

In this work we presented an implementation of *Cast Calculus Label Dependent Lambda Calculus*, an extension to regular *Label Dependent Lambda Calculus*. Mainly adding an expression for explicit type casts and a dynamic type. Its primary purpose is to shift type operations from the type checking stage to the later interpretation stage. Introducing type handling during runtime gives an option for dynamic typing. Explicit casts to and from the dynamic type are still unwieldy in this calculus. However, as introductory mentioned, this is but an intermediate representation when introducing full gradual typing.

The implementation was built upon LDGV, an existing interpreter and testbed for various language features. It already did support *Label Dependency* and recursion over natural numbers.[1] Furthermore, there is support for *Session Types*, a kind system, distinction between unrestricted and linear types, a compiler for the C programming language et cetera. Even though these features were not relevant for our extensions it is vital to consider them for compatibility with other calculi. On that node, we did extend the test suite to ease the implementation of additional features without introducing regression.

[1]Although partly in an unfinished state.

5.1 Open Problems

As given by the nature of programming language research as well as compiler development there are numerous ways to proceed. Possibilities for improvements of LDGV are twofold: Extending its “depth” by introducing novel language features and extending its “width” by integrating existing features.

On the one hand, the most obvious feature for adding “depth” is the implementation of full gradual typing via *Gradual Label Dependent Lambda Calculus* (GLDLC). There is an description in the same paper by Fu et al. [2021] that did introduce CCLDLC. The main concern is the translation between these calculi, deriving cast expressions for terms where merely the dynamic type is given. Instead of casting arguments to dynamic ($\ell_F : \textbf{Bool} \Rightarrow \star$) and later “undoing” this cast inside case terms ($\lambda(x : \star)$. **case** $(x : \star \Rightarrow \textbf{Bool})\{\dots\}$) it appears more natural from a programmer’s view to simply apply ℓ_F where type $\star$ is expected, letting the compiler figure out the intermediates. This seems vital for making dynamic types actually usable.

On the other hand, adding “width” is possible by better integration of aforementioned **rec** terms. Regardless of whether they are forming expressions or types, a value representing a number is necessary for resolving the recursion. Even though we had to introduce types to the interpreter the type checker still does not know about values (as preferable to avoid entanglement). Furthermore, resolving complex expressions inside case terms remains awkward, especially in CCLDLC. For example, passing a simple **rec** expression for variable $rc : \star$ warrants an application to a complicated cast inside a **case** expression.

$$\begin{aligned}
&\lambda(n : \textbf{ Nat }) \\
&\quad \lambda(rc : \star) \\
&\qquad \textbf{case } ((rc : \star \Rightarrow (n : Nat) \to \textbf{ natrec } n\{ \textbf{ Bool } , A.A\})\ n) \\
&\qquad\ \{\ell_T : n, \ell_F : rc\ n\}
\end{aligned} \tag{31}$$

Instead, one might use an expression combining **rec** and **natrec** terms. Such a composite could be `new_natrec f:n.tid.t {ez, x.es}` and is currently implemented similar to **rec**, evaluating to a value when applying a natural number. Whether this expression will prove useful depends on better integration into the *Cast Calculus*, too.

At last, LDGV supports a backend for the C programming language. We did not extend this part of the software, instead focusing on the interpreter. Thus, is still solely supports basic LDLC. Extending the C compiler requires the generation of C code to represent types (*type passing*) and type casts as well as implementing the cast reduction rules as in chapter 4. Even though it is possible to cross-check against the interpreter's results it will require precision and effort to fully adopt the *Cast Calculus*.

5.2 Outlook

Label Dependency is just one out of many concepts from programming language research. There are session types, linear types and others not mentioned here. Even though type systems may help to reduce programming errors of various kinds, they add an initial cognitive burden. They must be understood before applying them to practical problem. In contrast, dynamic languages make it is easy to start out even with limited understanding. Here, the burden will manifest later when maintaining sufficiently complex software. Postponing problems to the future should feel natural for most human beings.

Generally, gradual typing might help with the adoption of new concepts, thus making advanced type systems optional but accessible at the time. This hopefully opens up a path between research and programmers, with and without experience in dependent types.

Bibliography

David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273–309, 2001. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00175-4.

Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6): 567–600, 2002. doi: 10.1017/S0956796801004282.

Weili Fu, Fabian Krause, and Peter Thiemann. Label dependent lambda calculus and gradual typing. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485485.

Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. doi: 10.1017/S0956796809990268.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371135.