Implementing Substructural Logical Frameworks

Anders Schack-Nielsen

PhD Dissertation
Programming, Logic, and Semantics group
IT University of Copenhagen

January, 2011

Abstract

A key component in proof assistant software is the meta-language used to encode the objects that are being reasoned about. Such a meta-language is called a logical framework. Several different logical frameworks exist; some only provide the most basic encoding of abstract syntax data, while others support powerful representation methodologies and concepts such as judgments-as-types and higher-order abstract syntax, e.g. the logical framework LF.

The direct support for high-level concepts in the logical framework allows for rapid prototyping of new logics, type systems, and semantics. It also eases the development of theorems when the key concepts are directly supported. A concept, which is becoming increasingly important, is resources, but so far resources have not been supported very well by existing logical frameworks.

In this thesis I develop the theoretical infrastructure required to implement — and give an implementation of — a new logical framework that extends LF with the concepts of both linear resources, which must be used *exactly once*, and affine resources, which can be used *at most once*.

I take a slightly revised version of CLF [CPWW02a] as the starting point for my logical framework. I develop an explicit substitution calculus with linear and affine types, which I use as the foundation in my implementation of CLF called Celf. In this setting I prove strong normalization, type preservation, and confluence for a context-split-oblivious reduction semantics. I define the pattern fragment for the higher-order unification problem in linear and affine type theory and give a deterministic algorithm that computes most general unifiers on this fragment. I also extend this algorithm to the linear-changing pattern fragment and use this to implement the unification algorithm in Celf. Finally, I describe the Celf implementation.

Contents

1 Introduction **5**

1.1 Background 5

1.2 Implementing a substructural logical framework 6

1.3 CLF 7

1.4 Contributions 10

1.5 Digital content 11

1.6 About this thesis 11

2 The $\lambda\sigma$-Calculus and Strong Normalization **12**

2.1 Summary 12

2.2 Introduction 12

2.3 The $\lambda\sigma$-calculus 14

2.3.1 Variations of $\lambda\sigma$ 15

2.4 A non-terminating example 16

2.5 Revisiting the congruence rules 17

2.6 Strong normalization of $\lambda\sigma$ 18

2.6.1 Strong normalization of the σ-fragment 18

2.6.2 Confluence of the σ-fragment 21

2.6.3 Preservation of strong normalization 21

2.6.4 Strong normalization of simply typed $\lambda\sigma$ 21

2.7 Extending the strong normalization proof 22

2.7.1 The extended reduction relation 23

2.7.2 Strong normalization of the extended reduction relation . 23

3 Curry-Style Explicit Substitutions for the Linear and Affine Lambda Calculus **27**

3.1 Summary 27

3.2 Introduction 27

3.3 Explicit substitutions 28

3.4 The type system 30

3.4.1 Contexts 30

3.4.2 Types 31

3.5 Reduction semantics 33

3.5.1 Type preservation 34

3.5.2 σ-reduction 35

3.5.3 Confluence and termination 37

3.6 Dependent types 38

4 Pattern Unification for the Lambda Calculus with Linear and Affine Types 39
4.1 Summary . . . 39
4.2 Introduction . . . 39
4.3 The explicit substitution calculus . . . 40
4.3.1 Canonical forms . . . 40
4.3.2 Spine notation . . . 42
4.3.3 Instantiation of logic variables . . . 43
4.4 Patterns . . . 44
4.4.1 The pattern fragment . . . 44
4.4.2 Stability of the pattern fragment . . . 45
4.4.3 Inversion of substitutions . . . 46
4.5 Pattern unification . . . 49
4.5.1 Unification problem . . . 49
4.5.2 Unification algorithm . . . 49
4.5.3 Correctness . . . 52
4.6 Linearity pruning . . . 53
4.6.1 Revisiting variable occurrence . . . 53
4.6.2 Linear-changing pattern substitutions . . . 54
4.6.3 Linearity pruning . . . 56
4.6.4 Correctness . . . 60
4.6.5 Algorithmic linearity pruning . . . 60

5 The CLF Type Theory 65
5.1 Summary . . . 65
5.2 The original CLF type theory . . . 65
5.3 Redesigning the CLF type theory . . . 66
5.3.1 Affine types and $\top$ . . . 66
5.3.2 Linear and dependent implication from a focused perspective . . . 67
5.3.3 Dependent pairs and tensor . . . 68
5.4 CLF version 2 . . . 69
5.4.1 Syntax . . . 69
5.4.2 Judgments . . . 69
5.4.3 Hereditary substitution . . . 72
5.5 CLF with logic variables and redices . . . 74
5.5.1 Redices . . . 74
5.5.2 Logic variables . . . 76

6 Celf — an Implementation of CLF 79
6.1 Introduction . . . 79
6.2 Core functionality . . . 79
6.3 Using Celf . . . 80
6.3.1 Commandline . . . 80
6.3.2 Grammar . . . 80
6.3.3 Abbreviations and redices . . . 80
6.3.4 Implicit parameters and type inference . . . 81
6.3.5 Modality inference . . . 81
6.3.6 Queries . . . 82
6.3.7 Double checking . . . 82

6.4 Implementation 83
6.4.1 Resource management 83
6.4.2 Explicit substitutions through recursion schemes 83
6.4.3 Contexts 86
6.4.4 Type inference 86
6.4.5 Proof search 86
6.4.6 Unification 86

7 Conclusion and Future Work **94**

Acknowledgements

> *You know this metal rectangle full of little lights?*
> *Yeah.*
> *I spend most of my life pressing buttons to make the pattern of lights change however I want.*
> *Sounds good.*
> *But today, the pattern of lights is* all wrong*!*
> *Oh God! Try pressing more buttons!*
> *It's not helping!*
>
> — *xkcd (Computer Problems)*

A few words of thanks are in order. First of all to my advisor Carsten Schürmann for being an all-around great guy and fun to work with. Also thanks to all the cool people at Carnegie Mellon University (and beyond) for making my stay in Pittsburgh so enjoyable. And to all my friends and colleagues at the IT University, of course!

And finally, to those people who actually read this thesis in its entirety, thanks, and I hope you enjoy the style — I have tried to be as brief and concise as possible.

Anders Schack-Nielsen

Chapter 1

Introduction

1.1 Background

Proof assistant software or interactive theorem provers are gaining increased popularity and use. Several different systems exist, such as Twelf [PS99], Abella [Gac08, Abea], Coq [Coq], Agda [BDN09], Isabelle [Isa], and many others. A common structure to such systems is to first encode the object of interest, say a programming language, a type system, a logic, or an operational semantics, and then prove the relevant theorems about the encoding. An important foundation of proof assistants is thus the meta-language used for encodings. Such a meta-language is called a *logical framework*.

In general a logical framework is a language, e.g. a type theory, coupled with an encoding methodology in order to make the language useful as a meta-language.

In Twelf the logical framework is LF [HHP93], which is a dependently typed lambda calculus. This framework comes with the encoding methodologies *judgments-as-types* and *higher-order abstract syntax*, which have proven to be immensely useful for encoding many different logics and semantics, in particular those which make use of bound variables as higher-order abstract syntax provides α-conversion and substitution for free.

Bound variables continue to be an ubiquitous concept, but recent trends in logics for computer science have seen an increasing focus on the concept of resources. Examples include separation logic [Rey02] and session types [HVK98, CP10]. It is therefore important to develop logical frameworks that directly support the representation of resources.

A natural setting for the resource concept is linear and affine logic [Gir87, Bie94], in which variables represent resources. A linear variable/resource must be used *exactly once*, whereas an affine variable/resource can be used *at most once*. Linear and affine logics are *substructural*, since they are characterized by their restriction of the structural rules; weakening is disallowed for linear assumptions and contraction is disallowed for both linear and affine assumptions.

Starting from LF the linear logical framework LLF [CP96] and subsequently the concurrent logical framework CLF [CPWW02a] was suggested as a way to incorporate first resources and then also concurrency into a logical framework.

LLF is a conservative extension of LF with the connectives $\multimap$, $\&$, and $\top$

from intuitionistic linear logic, and CLF is a conservative extension of LLF with the linear logic connectives $\otimes$, 1, $!$, and $\exists$. CLF also includes a monad corresponding to the lax modality in order to encapsulate the positive connectives and thereby retain canonical forms. The equational theory of CLF includes a permutative conversion of monadic bindings that facilitates a natural encoding of concurrent computation traces. A number of examples demonstrating the applicability of CLF as a logical framework was presented in the companion tech report [CPWW02b], and the encoding methodology for the representation of concurrent computations in CLF was further elaborated in [WCPW08].

Explicit substitutions [ACCL91, DHKP98, NW98, Kes07, BR95] are central for modern implementations of systems that provide mechanisms for variable binding, such as logical frameworks [PS99], theorem provers [Gac08, BGM$^+$07], proof assistants [Bar00], and programming language implementations [SPM03, NM99, SLM98, PS08] and analysis [CNR08]. In particular, the explicit substitution calculus $\lambda\sigma$ [ACCL91] forms the basis of the Twelf implementation and it is directly used as a means to specify and implement unification [DHKP98].

1.2 Implementing a substructural logical framework

The long-term goal of this work is to develop a full-fledged proof assistant that supports resources as naturally and easily as e.g. Twelf supports higher-order abstract syntax (see future work in chapter 7). Before this becomes possible we need to be able to implement the underlying logical framework.

CLF has already been suggested as a suitable logical framework that facilitates the encoding of both resources and concurrency.

So far CLF has only been defined in terms of a canonical forms presentation. But in order to implement a substructural logical framework such as CLF we need a lot more. One of the most important things is unification, as this is a crucial part of both type inference and proof search. This means a higher-order unification algorithm that handles resources correctly. Additionally, we need explicit substitutions both as a means to implement β-equality and in order to support logic variables for unification.

Foreshadowing the development in chapter 5 it will become clear that the additive unit, $\top$, is problematic and needs to be replaced by affine types. In particular, in a calculus with linear types, logic variables, and explicit substitutions, the addition of $\top$ breaks decomposition of equality and makes invertibility of substitutions ill-defined. I therefore revise CLF slightly in chapter 5 from its original definition in [CPWW02a].

In the chapters ahead I build up to the logical framework implementation, Celf, of the revised CLF type theory. So even though the various calculi presented throughout this thesis might look slightly different, they are all constructed as sub-calculi of the final CLF by only projecting away those details that are irrelevant to the problem at hand.

1.3 CLF

As an example of the use of the CLF type theory as a logical framework, I demonstrate an encoding of π-calculus with session types. I follow the presentation in [HVK98] although slightly simplified for presentation purposes.

The encoding given below can be downloaded from `http://www.twelf.org/~celf/download/session-types.clf` and can be run with Celf (see chapter 6).

The base language expressions, `exp`, are restricted to natural numbers, which evaluate to themselves, and names, which will be introduced by binders when needed.

```
% Base Language
exp : type.
z : exp.
s : exp -> exp.

% Base Language: Operational Semantics
eval : exp -> exp -> type.
evz : eval z z.
evs : eval (s E) (s V)
   <- eval E V.
```

The encoding of types consists of base types, `tp`, session types, `stp`, and two predicates encoding type validity by making sure that types match with their co-type.

```
% Base Types
tp : type.
nat : tp.

% Session Types
stp : type.
down : tp -> stp -> stp.
up : tp -> stp -> stp.
end : stp.
st : stp -> stp -> tp. % Injection into Base Types

% Duality and validity of types
dual : stp -> stp -> type.
validtype : tp -> type.
d1 : dual (down T S) (up T S')
   <- validtype T
   <- dual S S'.
d2 : dual (up T S) (down T S')
   <- validtype T
   <- dual S S'.
d3 : dual end end.
vnat : validtype nat.
vst : validtype (st T T')
   <- dual T T'.
```

Process expressions are encoded by the type `pe`. The process `print` e is a stuck dummy process, which is used to record output in the example query below.

```
% Process Primitives
channel : type.
pe : type.
request : exp -> (channel -> pe) -> pe.
accept : exp -> (channel -> pe) -> pe.
send : channel -> exp -> pe -> pe.
receive : channel -> (exp -> pe) -> pe.
| : pe -> pe -> pe.
inact : pe.
print : exp -> pe.
newS : tp -> (exp -> pe) -> pe.
```

The operational semantics works by representing each process P that is waiting to execute as a linear assumption `proc` P in the context. The query at the end demonstrates this by doing a proof search for $\texttt{proc}\ p \multimap \{\texttt{proc}\ (\texttt{print}\ X)\}$, where p is the program

$$\begin{aligned}&(\nu a)(\texttt{accept}\ a(k)\ \texttt{in}\ k![1]; k?(x)\ \texttt{in}\ \texttt{print}[x]\\ &\quad \mid \texttt{request}\ a(k)\ \texttt{in}\ k?(x)\ \texttt{in}\ k![x+1]; \texttt{inact})\end{aligned}$$

This effectively runs p with the expected result of `print` X for some X. When run, Celf will perform the proof search and print the trace (proof term) together with the solution $X = 2$.

```
% Operational Semantics
proc : pe -> type.

link : proc (accept A P1) -o proc (request A P2)
   -o {Exists k. proc (P1 !k) * proc (P2 !k)}.

com : proc (send K E P1) -o proc (receive K P2)
   -o eval E V
   -> {proc P1 * proc (P2 !V)}.

par : proc (| P1 P2) -o {proc P1 * proc P2}.

clean : proc (inact) -o {1}.

introS : proc (newS T P)
   -o {Exists a. !eval a a * proc (P !a)}.

#query * 1 * 1
   proc (newS T (\!a.
    | (accept a (\!k. send k (s z) (receive k \!x. print x)))
      (request a (\!k. receive k (\!x. send k (s x) inact)))))
   -o {proc (print X)}.
```

The typing judgment $\Theta; \Gamma \vdash P \triangleright \Delta$ is encoded by the type `valid` P. Assumptions $a : S$ in Γ are represented by intuitionistic assumptions `of` a S, and assumptions $k : \alpha$ in Δ are represented as linear assumptions `c_of` k α. The context Θ is not used in this example.

```
% Base Language: Static Semantics/Type System
of : exp -> tp -> type.
ofz : of z nat.
ofs : of (s N) nat
   <- of N nat.

% Sessions: Static Semantics/Type System
valid : pe -> type.
c_of : channel -> stp -> type.
valid' : channel -> pe -> type.

v_ : valid' K P
   o- valid P
   o- ((c_of K end -o {1}) -@ {1}).

v_acc: valid (accept A P)
   <- of A (st S S')
   o- (Pi k. c_of k S -o valid' k (P !k)).

v_req: valid (request A P)
   <- of A (st S S')
   o- (Pi k. c_of k S' -o valid' k (P !k)).

v_send: valid (send K E P)
   o- c_of K (up T S)
   <- of E T
   o- (c_of K S -o valid' K P).

v_receive: valid (receive K P)
   o- c_of K (down T S)
   o- (Pi x. of x T -> c_of K S -o valid' K (P !x)).

v_inact: valid inact.

v_print : valid (print E)
   <- of E T.

v_newS : valid (newS T P)
   o- (Pi a. of a T -> valid (P !a))
   <- validtype T.

v_par: valid (| P1 P2)
   o- valid P1
   o- valid P2.
```

The type `valid'` k P is equivalent to `valid` P except that it, for a specific channel k, specifies that linear assumptions of the form `c_of` k `end` are to be ignored. The reason for structuring it this way is to avoid premature unification of an unknown session type with `end` during proof search. Proof search can then be used as a session type inference algorithm by using logic variables as type annotations. In particular we can run the following query to correctly infer the session type of a in the program p from above.

```
#query * 1 * 1
   valid (newS T (\!a.
   | (accept a (\!k. send k (s z) (receive k \!x. print x)))
     (request a (\!k. receive k (\!x. send k (s x) inact))))).
```

The output of the query looks like this:

```
Solution: v_newS !(vst !(d2 !(d1 !d3 !vnat) !vnat))
   (\!a. \!X1. v_par
      (v_req (\!k. \X2. v_ (\@X3. {1}) (v_receive
         (\!x. \!X3. \X4. v_ (\@X5. {1}) (v_send
            (\X5. v_ (\@X6. {let {1} = X6 X5 in 1}) v_inact)
            !(ofs !X3) X4)) X2)) !X1)
      (v_acc (\!k. \X2. v_ (\@X3. {1}) (v_send
         (\X3. v_ (\@X4. {1}) (v_receive (\!x. \!X4. \X5. v_
            (\@X6. {let {1} = X6 X5 in 1})
            (v_print !X4)) X3)) !(ofs !ofz) X2)) !X1))
 #T = st !(up !nat !(down !nat !end))
        !(down !nat !(up !nat !end))
```

1.4 Contributions

My contributions and thesis are divided into two parts. A theoretical part consisting of chapters 2 through 4 and partly chapters 5 through 6, and an implementation part consisting of the Celf system and its documentation in chapters 5 through 6.

In order to give a solid foundation on which to implement substructural logical frameworks in general and my logical framework Celf in particular, I give several theoretical contributions concerning explicit substitution calculi and unification in substructural type theories.

In chapter 2 I prove strong normalization for the explicit substitution calculus $\lambda\sigma$ with a slight restriction in the congruence rules. $\lambda\sigma$ is the basis of the internal data structures in both Twelf and Celf and it is not strongly normalizing in its general form.

In chapter 3 I define a linear and affine type system for $\lambda\sigma$ with term metavariables along with a context-split-oblivious small-step reduction semantics for which I prove type preservation and confluence.

All the theorems in these two chapters have been formalized and mechanically checked in the proof assistants Abella and Twelf, respectively. The work in the remainder of my thesis has only been checked by hand, as some theorems are harder to formalize than others.

In chapter 4 I design the unification algorithm that is the core of Celf. Based on the explicit substitution calculus from chapter 3 I define the pattern fragment for higher-order unification problems in linear and affine type theory and give a deterministic unification algorithm that computes most general unifiers. I then extend the algorithm by a procedure called linearity pruning to bridge the gap to the intuitionistic pattern fragment.

This lays the theoretical foundation for my logical framework Celf.

In chapter 5 I take the logical framework CLF and revise it by adding affine types, removing $\top$, and generalizing several of the constructs. In particular, I show how to form the conservative generalization of the intuitionistic dependent function-type Π and the linear function-type $\multimap$.

Celf implements this revised version of CLF as a logical framework complete with unification, type inference, implicit parameter inference and reconstruction, and proof search. I describe the Celf implementation in chapter 6. Many of the interface design choices have been inspired by Twelf.

Besides what has been presented in chapters 2 through 5 an implementation with this level of complexity of course draws on several additional theoretical contributions from both myself and others, most of which are outlined in chapter 6. Most notably is the extension of the unification algorithm to a full-fledged unification constraint simplification algorithm for the full CLF type theory; in particular, I show how to reify non-deterministic context splits as unification constraints to be solved by linearity pruning.

1.5 Digital content

Chapter 2 is formalized in Abella. The proof source files can be downloaded from `http://www.itu.dk/people/anderssn/exsub-sn.tgz`.

Chapter 3 is formalized in Twelf. The proof source files can be downloaded from `http://www.itu.dk/people/anderssn/ex-sub-aff.tgz`.

The current Celf implementation is at the time of writing version 2.6. It can be downloaded from `http://www.twelf.org/~celf`.

1.6 About this thesis

This thesis has been written such that the chapters present a progression of theoretical work with a common thread culminating in the Celf implementation, but also in such a way that the individual chapters could be read independently.

Theorems that have been formalized in either Abella or Twelf are annotated with references to their proofs in the corresponding source files.

A note about “I” vs. “we”: In our field of study it is common to write “we” instead of “I”, either to refer to author and reader collectively, because of co-authorship, or simply because of tradition. Apart from the front matter, the conclusion, and this first chapter, I have thus written the entire thesis in the plural style.

Chapter 2

The $\lambda\sigma$-Calculus and Strong Normalization

2.1 Summary

In this chapter we show how a small restriction in the congruence rules of $\lambda\sigma$ gives a strongly normalizing calculus, which is otherwise not strongly normalizing. In addition to general insight into the normalization properties of explicit substitution calculi, this result also provides a very flexible foundation for the design of normalization procedures in any λ-calculus-based implementation, such as logical frameworks and proof assistants, and in particular our logical framework Celf.

The development in this chapter is formalized in the proof assistant Abella. Each theorem is annotated with the corresponding Abella file and theorem name. The Abella source files can be downloaded at `http://www.itu.dk/people/anderssn/exsub-sn.tgz`. The proofs have also been made part of the Abella example collection where they can be easily browsed: `http://abella.cs.umn.edu/examples/lambda-calculus/exsub-sn/`.

This chapter has been separately published as a technical report [SN10].

2.2 Introduction

Consider the λ-calculus and its reduction semantics $\rightarrow_\beta$:

$$t ::= x \mid t_1\ t_2 \mid \lambda x.\, t$$

$$(\lambda x.\, t_1)\ t_2 \rightarrow_\beta t_1\{t_2/x\}$$

As usual, terms are considered up to α-equivalence, and $t_1\{t_2/x\}$ denotes the capture-avoiding substitution of t_2 for x in t_1.

However, this form of substitution does not extend to meta-variables, since we cannot resolve $X\{t/x\}$ before we know the instantiation of X. Furthermore, a decomposition of the substitution application into more atomic steps allows for a more flexible and potentially more efficient implementation.

The simplest approach to an internalization of substitution is to change $t_1\{t_2/x\}$ into an explicit substitution $t_1[t_2/x]$:

$$t ::= \cdots \mid t_1[t_2/x]$$

and turn the definition of $t_1\{t_2/x\}$ into reduction rules:

$$\begin{array}{lll} x[t/x] \to t & & (t_1\ t_2)[t_3/x] \to t_1[t_3/x]\ t_2[t_3/x] \\ y[t/x] \to y & \text{if } x \neq y & (\lambda y.\, t_1)[t_2/x] \to \lambda y.\, t_1[t_2/x] \quad \text{if } x \neq y \text{ and } y \notin \mathrm{fv}(t_2) \end{array}$$

along with the β-rule $(\lambda x.\, t_1)\ t_2 \to t_1[t_2/x]$. This explicit substitution calculus is known as $\lambda\mathrm{x}$ [Kes07, BR95].

However, without any way to compose substitutions, this calculus is not confluent in the presence of meta-variables, and a naive addition of composition rules breaks normalization.

Many different explicit substitution calculi have been proposed trying to capture as many good properties as possible. Kesner gives a nice overview of related work in [Kes07] and highlights six important and desirable properties of explicit substitution calculi:

Confluence (C) Confluence of the reduction relation.

Meta-Confluence (MC) Confluence in the presence of meta-variables. This can either refer to both term and substitution meta-variables or term meta-variables alone.

Preservation of Strong Normalization (PSN) Any pure term that is strongly normalizing with respect to $\to_\beta$ is also strongly normalizing with respect to $\to$, where a pure term denotes an ordinary λ-term without any explicit substitutions.

Strong Normalization (SN) Strong normalization of well-typed terms.

Simulation (SIM) The reduction relation $\to$ can simulate β-reduction, i.e. $t_1 \to_\beta t_2$ implies $t_1 \to^* t_2$.

Full Composition (FC) For any t_1 and t_2 it holds that $t_1[t_2/x] \to^* t_1\{t_2/x\}$ for an appropriate extension of ordinary capture-avoiding substitution to terms with explicit substitutions.

At least one very important property, which is crucial for implementation, is missing from this list, namely:

Locality (L) The applicability of the reduction rules can be determined solely from local information in the term.

As an example, all the rules of $\lambda\mathrm{x}$ above are local, but the rule **gc** below, which is known from several explicit substitution calculi [Kes07, BR95], is not, since the side-condition cannot be achieved by α-conversion but must be checked by traversing t_1.

$$(\mathbf{gc}) \qquad t_1[t_2/x] \to t_1 \quad \text{if } x \notin \mathrm{fv}(t_1)$$

To our knowledge no explicit substitution calculus is known to have all of these properties.

The $\lambda\sigma$-calculus is a simple minimalistic explicit substitution calculus, which has most of the properties we expect from an explicit substitution calculus (**C**, **SIM**, **FC**, **L**), and it implements both β- and α-equality in a single uniform way through the use of de Bruijn indices. It has already shown itself to be useful for implementation and specification of higher-order unification [DHKP98] and serves as the underlying calculus in Twelf.

However, $\lambda\sigma$ does not preserve strong normalization, i.e. a strongly normalizing λ-term might not be strongly normalizing as a $\lambda\sigma$-term [Mel95]. Furthermore, confluence of the system is fairly brittle as the introduction of meta-variables can introduce non-confluence [CHL96]. We will address strong normalization in this chapter and return to confluence in the presence of meta-variables in chapter 3.

2.3 The $\lambda\sigma$-calculus

The syntax of the $\lambda\sigma$-calculus consists of terms and substitutions written in de Bruijn notation:

Terms:	$M, N ::= 1 \mid M[s] \mid \lambda M \mid M\ N$
Substitutions:	$s, t ::= \mathsf{id} \mid \uparrow \mid M \,.\, s \mid s \circ t$

The variable 1 refers to the innermost λ-binder. Other variables are represented with a closure and a sequence of shifts, e.g. $3 = 1[\uparrow \circ \uparrow]$.

The intuition behind each of the substitution constructs is the following: A term under an identity (id) is supposed to reduce to itself. A shift ($\uparrow$) applied to a term increments all freely occurring variables by one. An extension $M \,.\, s$ will substitute M for the variable 1, decrement all other freely occurring variables, and then apply s to them. Finally, a composition of two substitutions $s \circ t$ represents the substitution that first applies s and then t, i.e. $M[s \circ t]$ is supposed to reduce to the same term as reducing each closure individually in $M[s][t]$.

We will use $\uparrow^n$ where $n \geq 0$ as a short-hand for n compositions of shift, i.e. $\uparrow \circ (\uparrow \circ (\ldots \circ (\uparrow \circ \uparrow) \ldots))$, where $\uparrow^0$ means id. Additionally, de Bruijn indices n with $n > 1$ are short-hand for $1[\uparrow^{n-1}]$.

beta	$(\lambda M)\ N$	$\rightarrow$	$M[N \,.\, \mathsf{id}]$
clos-var	$1[M \,.\, s]$	$\rightarrow$	M
clos-clos	$M[s][t]$	$\rightarrow$	$M[s \circ t]$
clos-lam	$(\lambda M)[s]$	$\rightarrow$	$\lambda(M[1 \,.\, (s \circ \uparrow)])$
clos-app	$(M\ N)[s]$	$\rightarrow$	$M[s]\ N[s]$
clos-var-id	$1[\mathsf{id}]$	$\rightarrow$	1
comp-id-L	$\mathsf{id} \circ s$	$\rightarrow$	s
comp-shift-id	$\uparrow \circ\, \mathsf{id}$	$\rightarrow$	$\uparrow$
comp-shift	$\uparrow \circ\, (M \,.\, s)$	$\rightarrow$	s
comp-cons	$(M \,.\, s) \circ t$	$\rightarrow$	$M[t] \,.\, (s \circ t)$
comp-comp	$(s_1 \circ s_2) \circ s_3$	$\rightarrow$	$s_1 \circ (s_2 \circ s_3)$

Figure 2.1: Basic reduction rules

The reduction rules are shown in Figure 2.1. In addition to these rules we are also going to include every possible congruence rule, i.e. we allow rewrites to occur anywhere inside a term or substitution.

The rule **beta** corresponds to the ordinary β-step in the λ-calculus and introduces an explicit substitution inside a closure. The rest of the rules are called σ-rules and details how to evaluate closures and substitution compositions. The fragment of the rules that excludes the **beta** rule is called the σ-fragment and the corresponding relation is written $\rightarrow_\sigma$.

2.3.1 Variations of $\lambda\sigma$

Several variations of $\lambda\sigma$ have been treated in the literature. The most important distinction is whether or not we include term and substitution meta-variables. In the presence of meta-variables the system presented above is not even locally confluent, since the critical pair $((\lambda M)N)[s]$ can reduce to both $M[N[s] \,.\, s]$ and $M[N[s] \,.\, (s \circ \mathsf{id})]$. Closing the critical pairs leads to the following four additional rules:

comp-id-R	$s \circ \mathsf{id}$	$\rightarrow$	s
clos-id	$M[\mathsf{id}]$	$\rightarrow$	M
var-shift	$1 \,.\, \uparrow$	$\rightarrow$	id
s-cons	$1[s] \,.\, (\uparrow \circ s)$	$\rightarrow$	s

The first two rules are the general right-identity rules and thus replace **clos-var-id** and **comp-shift-id**. The **var-shift** and **s-cons** rules can be seen as η-contraction rules on substitutions. The resulting system is easily seen to be locally confluent by checking all the critical pairs.

Constants can be added to the calculus with just a single additional rule:

clos-const	$c[s]$	$\rightarrow$	c

We can also represent de Bruijn indices and sequences of shifts directly in the syntax with the following five rules:

clos-var-dot2	$(n+1)[M \,.\, s]$	$\rightarrow$	$n[s]$
clos-var-shift	$n[\uparrow^m]$	$\rightarrow$	$n+m$
comp-shift-dot	$\uparrow^{n+1} \circ (M \,.\, s)$	$\rightarrow$	$\uparrow^n \circ s$
comp-shift-shift	$\uparrow^n \circ \uparrow^m$	$\rightarrow$	$\uparrow^{n+m}$
eta-subst	$(n+1) \,.\, \uparrow^{n+1}$	$\rightarrow$	$\uparrow^n$

These rules are essentially shortcuts in the sense that the first four can be obtained by a sequence of the reduction steps shown in Figure 2.1 when n and $\uparrow^n$ are syntactic short-hands. It is also easy to check that the addition of the first four rules suffices to evaluate all compositions and closures. These rules obsolete **clos-var-id**, **comp-shift-id**, and **comp-shift**. The last rule generalizes **var-shift** and corresponds to **s-cons** in the case when $s = \uparrow^n$.

For now we will postpone further discussion of meta-variables. For the remainder of this chapter we will therefore define $\lambda\sigma$ to be:

Terms:	$M, N ::= n \mid c \mid M[s] \mid \lambda M \mid M\ N$
Substitutions:	$s, t ::= \uparrow^n \mid M \,.\, s \mid s \circ t$

where de Bruijn indices n have $n \geq 1$ and shifts $\uparrow^n$ have $n \geq 0$. The reduction rules are given in Figure 2.2. We have excluded the two general right-identity rules, as we can easily prove them admissible for the transitive closure:

beta	$(\lambda M)\ N$	$\to$	$M[N \,.\, \mathsf{id}]$
clos-const	$c[s]$	$\to$	c
clos-var-dot1	$1[M \,.\, s]$	$\to$	M
clos-var-dot2	$(n+1)[M \,.\, s]$	$\to$	$n[s]$
clos-var-shift	$n[\uparrow^m]$	$\to$	$n+m$
clos-clos	$M[s][t]$	$\to$	$M[s \circ t]$
clos-lam	$(\lambda M)[s]$	$\to$	$\lambda(M[1 \,.\, (s \circ \uparrow^1)])$
clos-app	$(M\ N)[s]$	$\to$	$M[s]\ N[s]$
comp-id-L	$\uparrow^0 \circ s$	$\to$	s
comp-cons	$(M \,.\, s) \circ t$	$\to$	$M[t] \,.\, (s \circ t)$
comp-shift-dot	$\uparrow^{n+1} \circ (M \,.\, s)$	$\to$	$\uparrow^n \circ s$
comp-shift-shift	$\uparrow^n \circ \uparrow^m$	$\to$	$\uparrow^{n+m}$
comp-comp	$(s_1 \circ s_2) \circ s_3$	$\to$	$s_1 \circ (s_2 \circ s_3)$
eta-subst	$(n+1) \,.\, \uparrow^{n+1}$	$\to$	$\uparrow^n$

Figure 2.2: Reduction rules

Theorem 2.3.1 (`beta.thm:clos_id_ext`). *For all M and s we have $M[\uparrow^0] \to^* M$ and $s \circ \uparrow^0 \to^* s$.*

Proof. The proof is an easy mutual induction over M and s. □

We have also omitted the general **s-cons** rule and only included **eta-subst** in its place. The general **s-cons** rule is problematic in the sense that it is not left-linear, so it is worth analyzing what it actually adds. It is easy to see that any composition can reduce to either $M \,.\, s$ or $\uparrow^n$ (a concrete reduction sequence is given by Theorem 2.7.3 below). In these two cases **s-cons** gives us the two reductions $1[M \,.\, s] \,.\, (\uparrow^1 \circ (M \,.\, s)) \to M \,.\, s$ and $1[\uparrow^n] \,.\, (\uparrow^1 \circ \uparrow^n) \to \uparrow^n$. The former can be achieved in two steps without **s-cons**, and the latter corresponds to **eta-subst**. This shows that we do not lose any reduction sequences by excluding the problematic **s-cons** in favor of **eta-subst**, and thus it is the better choice.

We are not going to say much about the congruence rules yet, so for now we will just assume that we can perform any reduction step anywhere within a term or substitution. We will return to this matter below in section 2.5.

2.4 A non-terminating example

Mellies showed in [Mel95] that the simply typed term

$$\lambda v.(\lambda x.(\lambda y.y)((\lambda z.z)x))((\lambda w.w)v)$$

has an infinite reduction sequence in $\lambda\sigma$. This is very counter-intuitive since the ordinary simply typed λ-calculus is strongly normalizing and the σ-fragment of $\lambda\sigma$ is also strongly normalizing [CHR92] (even on untyped terms).

We will sketch the idea of the counter-example here. Consider a β-redex under a closure:

$$((\lambda M)\ N)[s]$$

If we evaluate the redex first and then the substitution composition we arrive at $M[N[s].s]$. If we instead begin by pushing the substitution through the application we can get the following reduction sequence:

$$\begin{aligned}((\lambda M)\ N)[s] &\to (\lambda M)[s]\ N[s] \\ &\to (\lambda M[1\ .\ s \circ {\uparrow}^1])\ N[s] \\ &\to M[1\ .\ s \circ {\uparrow}^1][N[s]\ .\ {\uparrow}^0] \\ &\to^* M[N[s]\ .\ s \circ ({\uparrow}^1 \circ (N[s]\ .\ {\uparrow}^0))]\end{aligned}$$

Here we see a substitution $s' = {\uparrow}^1 \circ (N[s]\ .\ {\uparrow}^0)$, which contains s, being applied to s itself. Of course s' can reduce in one step to ${\uparrow}^0$, but if we carefully avoid that specific reduction step and if s contains a β-redex then we can push s' into s and through the redex in s and thus replicate the situation above with s' instead of s. Now since s' contains s and therefore also a β-redex we can keep on doing this (see [Mel95] for all the details).

The term that we end up creating in this way consists of a sequence of closures nested arbitrarily deep:

$$M[\dots M[\dots M[\dots M[\dots]\dots]\dots]\dots]$$

And consequently the reduction steps that we perform are similarly nested deeper and deeper through an arbitrary number of closures and substitutions.

This also highlights the brittle nature of the counter-example. If we at any time were to reduce any of the arising s's to ${\uparrow}^0$ the entire thing would normalize. Thus we have an indication that a suitable minor tweak to the reduction strategy will give us strong normalization (as opposed to the current non-deterministic, everything-is-allowed reduction strategy).

2.5 Revisiting the congruence rules

Let us present the congruence rules that up until now have remained implicit:

$$\frac{M \to M'}{M\ .\ s \to M'\ .\ s} \qquad \frac{s \to s'}{M\ .\ s \to M\ .\ s'} \qquad \frac{s \to s'}{s \circ t \to s' \circ t} \qquad \frac{t \to t'}{s \circ t \to s \circ t'}$$

$$\frac{M \to M'}{\lambda M \to \lambda M'} \qquad \frac{M \to M'}{M\ N \to M'\ N} \qquad \frac{N \to N'}{M\ N \to M\ N'}$$

$$\frac{M \to M'}{M[s] \to M'[s]} \qquad \frac{s \to s'}{M[s] \to M[s']}\,(*)$$

If we get rid of the second congruence rule for closures $(*)$ then certainly we will have a strongly normalizing calculus, but this is overly restrictive. If we instead replace it by a version that only allows σ-steps then since the σ-fragment by itself is strongly normalizing, we can hope for strong normalization. We are therefore going to use the following congruence rule in place of $(*)$:

$$\frac{s \to_\sigma s'}{M[s] \to_\sigma M[s']}$$

The resulting rewrite system is indeed strongly normalizing for simply typed terms as we will see below.

$$
\begin{array}{lcllcl}
c[e] & \rightarrow & c & 1[e_1 \,.\, e_2] & \rightarrow & e_1 \\
(n+1)[e_1 \,.\, e_2] & \rightarrow & n[e_2] & n[\uparrow^m] & \rightarrow & n+m \\
e_1[e_2][e_3] & \rightarrow & e_1[e_2[e_3]] & (\lambda e_1)[e_2] & \rightarrow & \lambda e_1[1 \,.\, e_2[\uparrow^1]] \\
(e_1 \,.\, e_2)[e_3] & \rightarrow & e_1[e_3] \,.\, e_2[e_3] & \uparrow^0[e] & \rightarrow & e \\
\uparrow^{n+1}[e_1 \,.\, e_2] & \rightarrow & \uparrow^n[e_2] & \uparrow^n[\uparrow^m] & \rightarrow & \uparrow^{n+m} \\
(n+1) \,.\, \uparrow^{n+1} & \rightarrow & \uparrow^n & & &
\end{array}
$$

$$
\frac{e_1 \rightarrow e_1'}{e_1[e_2] \rightarrow e_1'[e_2]} \qquad \frac{e_2 \rightarrow e_2'}{e_1[e_2] \rightarrow e_1[e_2']} \qquad \frac{e \rightarrow e'}{\lambda e \rightarrow \lambda e'}
$$

$$
\frac{e_1 \rightarrow e_1'}{e_1 \,.\, e_2 \rightarrow e_1' \,.\, e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1 \,.\, e_2 \rightarrow e_1 \,.\, e_2'}
$$

Figure 2.3: Expression reduction rules

2.6 Strong normalization of $\lambda\sigma$

The proof of strong normalization can be summarized to the following: Take any reduction sequence and divide it into groups of σ-steps and applications of **beta**. Since the σ-fragment is strongly normalizing we can focus on the **beta** steps. If we relate each term to its σ-normal form then we can show that every **beta** step corresponds to a regular β-reduction step in the λ-calculus, and we therefore get preservation of strong normalization.

The complete proof is formalized in the proof assistant Abella. Below are the central theorems along with references to the Abella source files.

2.6.1 Strong normalization of the σ-fragment

In [CHR92] the σ-fragment of $\lambda\sigma$ was proved strongly normalizing. The central part of the proof consists of showing that if e is strongly normalizing then $e[\uparrow^1]$ is also strongly normalizing, where e is an expression (defined below). The proof we give in this section share the same overall structure, but our proof of the strong normalization of $e[\uparrow^1]$ given strong normalization of e is greatly simplified (Lemma 2.6.5 and Lemma 2.6.6 below).

In order to give a simpler proof of strong normalization of the σ-fragment, we collapse the syntactic classes of terms and substitutions into a single class called expressions:

$$\textbf{Expressions:} \qquad e ::= n \mid c \mid \uparrow^n \mid e_1[e_2] \mid \lambda e \mid e_1 \,.\, e_2$$

The set of reduction rules for expressions is given in Figure 2.3. We use the following translation from terms and substitutions into expressions:

$$\mathcal{E}(n) = n \qquad \mathcal{E}(c) = c \qquad \mathcal{E}(M[s]) = \mathcal{E}(M)[\mathcal{E}(s)]$$

$$\mathcal{E}(\lambda M) = \lambda \mathcal{E}(M) \qquad \mathcal{E}(M\ N) = \mathcal{E}(M) \,.\, \mathcal{E}(N) \qquad \mathcal{E}(\uparrow^n) = \uparrow^n$$

$$\mathcal{E}(M \,.\, s) = \mathcal{E}(M) \,.\, \mathcal{E}(s) \qquad \mathcal{E}(s \circ t) = \mathcal{E}(s)[\mathcal{E}(t)]$$

It is easy to see that σ-reductions are preserved by the translation, and therefore strong normalization of expressions implies strong normalization of the σ-fragment.

We shall write $\mathcal{SN}$ for the set of strongly normalizing expressions, and then aim to prove $e \in \mathcal{SN}$ for all expressions e.

Strong normalization of the cases λe and $e_1 \,.\, e_2$ are fairly easy.

Lemma 2.6.1 (`sigma-strong.thm:esn_lam`). *If $e \in \mathcal{SN}$ then $\lambda e \in \mathcal{SN}$.*

Lemma 2.6.2 (`sigma-strong.thm:esn_dot,esn_dot_inv`). *$e_1 . e_2 \in \mathcal{SN}$ if and only if $e_1 \in \mathcal{SN}$ and $e_2 \in \mathcal{SN}$.*

Closures are a lot more difficult due to the rewrite $(\lambda e_1)[e_2] \to \lambda e_1[1 \,.\, e_2[\uparrow^1]]$. We will therefore split the proof depending on whether the given expressions contain λs. We write the exclusion of λs from an expression e as $\lambda \notin e$.[1]

Lemma 2.6.3 (`sigma-strong.thm:esn_clos_nolam`). *If $e_1 \in \mathcal{SN}$, $e_2 \in \mathcal{SN}$, and $\lambda \notin e_1$ then $e_1[e_2] \in \mathcal{SN}$.*

These lemmas can be put together to prove strong normalization in the absence of λs:

Theorem 2.6.4 (`sigma-strong.thm:nolam_esn`). *If $\lambda \notin e$ then $e \in \mathcal{SN}$.*

In order to prove the general statement we are going to need a version of Lemma 2.6.3 without the restriction on e_1. The tricky part is proving that $e_2 \in \mathcal{SN}$ implies $e_2[\uparrow^1] \in \mathcal{SN}$ (and thus $1 \,.\, e_2[\uparrow^1] \in \mathcal{SN}$).

If we do an intuitive comparison between reduction sequences for e and $e[\uparrow^1]$ it seems that any reduction in $e[\uparrow^1]$ either can be mimicked by a reduction in e or consists of pushing the $\uparrow^1$ through e. We will formalize this intuitive idea by building a relation $e \overset{\lhd}{\sim} e'$, such that $e \overset{\lhd}{\sim} e[\uparrow^1]$ for any e, and for any $e_1' \to e_2'$ with $e_1 \overset{\lhd}{\sim} e_1'$ then either $e_1 \overset{\lhd}{\sim} e_2'$ or $e_1 \to e_2$ with $e_2 \overset{\lhd}{\sim} e_2'$. In the former case the reduction $e_1' \to e_2'$ is intended to be one of the steps associated with pushing the $\uparrow^1$ through the structure of the expression and thus the number of such steps should be bounded by the structure of the expression. Given such a relation we would get the desired lemma as a corollary to the fact that $e \in \mathcal{SN}$ and $e \overset{\lhd}{\sim} e'$ implies $e' \in \mathcal{SN}$.

In order to build the relation such that it is closed under the conditions stated above, we need it to contain $e \overset{\lhd}{\sim} e[e']$ where e' can be $\uparrow^1$ and is closed under at least $1 \,.\, \cdot \circ \uparrow^1$ and reduction. Instead of characterizing this class of expressions we can simply take expressions without λ as we already know this class to be in $\mathcal{SN}$.

The relation is defined as follows:

$$\frac{}{c \overset{\lhd}{\sim} c} \qquad \frac{\lambda \notin e \quad \lambda \notin e'}{e \overset{\lhd}{\sim} e'} \qquad \frac{e_2 \overset{\lhd}{\sim} e_2'}{e_1[e_2] \overset{\lhd}{\sim} e_1[e_2']} \qquad \frac{e_1 \overset{\lhd}{\sim} e_1' \quad e_2 \overset{\lhd}{\sim} e_2'}{e_1 \,.\, e_2 \overset{\lhd}{\sim} e_1' \,.\, e_2'}$$

$$\frac{e \overset{\lhd}{\sim} e'}{\lambda e \overset{\lhd}{\sim} \lambda e'} \qquad \frac{\lambda \notin e'}{e \overset{\lhd}{\sim} e[e']} \qquad \frac{e_1 \overset{\lhd}{\sim} e_1' \quad \lambda \notin e_2}{e_1[e_2] \overset{\lhd}{\sim} e_1'[e_2]}$$

[1] The formalized proof represents the exclusion of λ by a predicate `nolam`. This predicate also excludes constants to reduce the number of cases needed in the proofs, but it could just as well have included them, and the general theorem is proved later anyway. Thus, whenever we write $\lambda \notin e$ we will also exclude occurrences of constants in e.

The following theorem states the desired simulation properties.

Lemma 2.6.5 (`sigma-strong.thm:is_esn_rel_under_shift`). *If $e_0 \overset{\lhd}{\sim} e'_0$ then there exists a k such that for all reductions $e'_0 \to e'_1 \to \cdots \to e'_n$ in n steps there exists a sequence $e_0, e_1, \ldots, e_n$ such that*

1. *either $e_i \to e_{i+1}$ or $e_i = e_{i+1}$ for $0 \leq i < n$,*
2. *$e_i \overset{\lhd}{\sim} e'_i$ for $0 \leq i \leq n$, and*
3. *$n \geq k$ implies $e_i \to e_{i+1}$ for some i.*

It is noteworthy to consider how Lemma 2.6.5 is encoded in Abella. The statement of the lemma is similar to strong normalization in the sense that some limit k exists after which any reduction sequence will bottom out in some way; in this case, have a corresponding reduction in the $\overset{\lhd}{\sim}$-related expression. The Abella-encoding of $\mathcal{SN}$ is the following inductive definition:

$$e \in \mathcal{SN} \quad := \quad \forall e'.\, e \to e' \supset e' \in \mathcal{SN}$$

We can then give a similar, but slightly more complicated, inductive definition of a relation $\mathcal{SN}[e]$ (denoted `esn_rel_under_shift` in the formalization).

$$\begin{aligned} e'_1 \in \mathcal{SN}[e_1] \quad := \quad & e_1 \overset{\lhd}{\sim} e'_1 \wedge \forall e'_2.\, e'_1 \to e'_2 \supset \\ & (\exists e_2.\, e_1 \to e_2 \wedge e_2 \overset{\lhd}{\sim} e'_2) \vee e'_2 \in \mathcal{SN}[e_1] \end{aligned}$$

Lemma 2.6.5 can now be represented as the statement $e \overset{\lhd}{\sim} e' \supset e' \in \mathcal{SN}[e]$.

By a nested induction on the strong normalization of e and the k from Lemma 2.6.5 we get the following lemma:

Lemma 2.6.6 (`sigma-strong.thm:exp_under_shift_esn`). *If $e \in \mathcal{SN}$ and $e \overset{\lhd}{\sim} e'$ then $e' \in \mathcal{SN}$.*

As an immediate corollary we get that $e \in \mathcal{SN}$ implies $e[\uparrow^1] \in \mathcal{SN}$. This gives us the desired version of Lemma 2.6.3 without the restriction on e_1.

Lemma 2.6.7 (`sigma-strong.thm:esn_clos`). *If $e_1 \in \mathcal{SN}$ and $e_2 \in \mathcal{SN}$ then $e_1[e_2] \in \mathcal{SN}$.*

Together Lemma 2.6.1, Lemma 2.6.2, and Lemma 2.6.7 give the strong normalization theorem:

Theorem 2.6.8 (`sigma-strong.thm:exp_esn`). *Every expression is strongly normalizing: $e \in \mathcal{SN}$ for all e.*

And thus, we have strong normalization of the σ-fragment of $\lambda\sigma$.

Theorem 2.6.9 (`lambda-sigma.thm:tm_sn_su,sub_sns_su`). *The reduction relation $\to_\sigma$ is strongly normalizing for terms and substitutions.*

2.6.2 Confluence of the σ-fragment

With strong normalization we can easily prove confluence of the σ-fragment by first proving local confluence.

Theorem 2.6.10 (`conf.thm:local_conf`)**.** *The σ-fragment is locally confluent.*

1. *If $M_1 \ {}_{\sigma}\!\!\leftarrow M \rightarrow_{\sigma} M_2$ then there exists an M' such that $M_1 \rightarrow^*_{\sigma} M' \ {}^*_{\sigma}\!\!\leftarrow M_2$.*

2. *If $s_1 \ {}_{\sigma}\!\!\leftarrow s \rightarrow_{\sigma} s_2$ then there exists an s' such that $s_1 \rightarrow^*_{\sigma} s' \ {}^*_{\sigma}\!\!\leftarrow s_2$.*

Theorem 2.6.11 (`conf.thm:conf_tm,conf_sub`)**.** *The σ-fragment is confluent.*

1. *If $M_1 \ {}^*_{\sigma}\!\!\leftarrow M \rightarrow^*_{\sigma} M_2$ then there exists an M' such that $M_1 \rightarrow^*_{\sigma} M' \ {}^*_{\sigma}\!\!\leftarrow M_2$.*

2. *If $s_1 \ {}^*_{\sigma}\!\!\leftarrow s \rightarrow^*_{\sigma} s_2$ then there exists an s' such that $s_1 \rightarrow^*_{\sigma} s' \ {}^*_{\sigma}\!\!\leftarrow s_2$.*

Together confluence and strong normalization give us the existence of unique σ-normal forms. We shall denote the σ-normal form of a term M or substitution s as $\sigma(M)$ and $\sigma(s)$, respectively.

2.6.3 Preservation of strong normalization

With the σ-fragment covered, we turn our attention to the **beta** steps. Let $\rightarrow_{\beta}$ denote a **beta** step. Then the reduction relation $\rightarrow$ is the disjoint union of $\rightarrow_{\sigma}$ and $\rightarrow_{\beta}$. The σ-normal forms correspond to the ordinary lambda calculus, so for such terms we can define ordinary β-reduction in the following way:

$$M \Rightarrow_{\beta} M' \quad := \quad M \rightarrow_{\beta} M'' \wedge M' = \sigma(M'')$$

Preservation of strong normalization (PSN) is the property that strong normalization of $\sigma(M)$ with respect to $\Rightarrow_{\beta}$ implies strong normalization of M with respect to $\rightarrow$.

Our restriction of the congruence rules allows us to prove that **beta** steps correspond to β-steps in the ordinary lambda calculus:

Theorem 2.6.12 (`beta.thm:project_beta`)**.** *If $M \rightarrow_{\beta} M'$ then $\sigma(M) \Rightarrow_{\beta} \sigma(M')$.*

Together with confluence and strong normalization of the σ-fragment, we immediately get PSN:

Theorem 2.6.13 (`strong-norm.thm:psn`)**.** *If $\sigma(M)$ is strongly normalizing with respect to $\Rightarrow_{\beta}$ then M is strongly normalizing with respect to $\rightarrow$.*

2.6.4 Strong normalization of simply typed $\lambda\sigma$

So far, we have only considered untyped terms and substitutions. So before we can talk about strong normalization of well-typed terms and substitutions, we need to introduce the type system. The typing rules are standard for $\lambda\sigma$ and given in Figure 2.4.

$$\frac{}{\Gamma, A \vdash 1 : A} \qquad \frac{\Gamma \vdash n : A}{\Gamma, B \vdash n + 1 : A} \qquad \frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash M : A}{\Gamma \vdash M[s] : A}$$

$$\frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad \frac{\Gamma, A \vdash M : B}{\Gamma \vdash \lambda M : A \to B}$$

$$\frac{}{\Gamma \vdash \uparrow^0 : \Gamma} \qquad \frac{\Gamma \vdash \uparrow^n : \Gamma'}{\Gamma, A \vdash \uparrow^{n+1} : \Gamma'}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M \,.\, s : \Gamma', A} \qquad \frac{\Gamma \vdash t : \Gamma'' \quad \Gamma'' \vdash s : \Gamma'}{\Gamma \vdash s \circ t : \Gamma'}$$

Figure 2.4: Types for $\lambda\sigma$

Theorem 2.6.14 (`typing.thm:of_step_ext`)**.** *Subject reduction.*

1. *If* $\Gamma \vdash M : A$ *and* $M \to M'$ *then* $\Gamma \vdash M' : A$.
2. *If* $\Gamma \vdash s : \Gamma'$ *and* $s \to s'$ *then* $\Gamma \vdash s' : \Gamma'$.

Since well-typed σ-normal forms are exactly the simply typed lambda calculus, we have the usual proof of strong normalization using a logical relation. The Abella proof is adapted from Girard's proof of strong normalization in the example suite of Abella [Abeb].

Theorem 2.6.15 (`beta-sn.thm:strong_beta`)**.** *Let* M *be a* σ*-normal form with* $\Gamma \vdash M : A$*. Then* M *is strongly normalizing with respect to* $\Rightarrow_\beta$*.*

Now PSN (Theorem 2.6.13) gives us strong normalization of simply typed terms:

Theorem 2.6.16 (`strong-norm.thm:strong_tm`)**.** *If* $\Gamma \vdash M : A$ *then* M *is strongly normalizing with respect to* $\to$*.*

We can adapt the proof of Lemma 2.6.3 to prove strong normalization of compositions. We will not have to consider λs in this case, since for terms we can simply appeal to Theorem 2.6.16.

Lemma 2.6.17 (`strong-norm.thm:sns_clos`)**.** *If* $\Gamma \vdash t : \Gamma''$, $\Gamma'' \vdash s : \Gamma'$, *and* s *and* t *are strongly normalizing then* $s \circ t$ *is strongly normalizing.*

As an immediate consequence we get strong normalization of substitutions:

Theorem 2.6.18 (`strong-norm.thm:strong_sub`)**.** *If* $\Gamma \vdash s : \Gamma'$ *then* s *is strongly normalizing.*

2.7 Extending the strong normalization proof

So far we have achieved strong normalization with the restricted congruence rule:

$$\frac{s \to_\sigma s'}{M[s] \to_\sigma M[s']}$$

With Theorem 2.6.18 we showed that substitutions are strongly normalizing, and it is therefore plausible that we could allow a single application of the unrestricted congruence rule in each step without breaking strong normalization. The intuition is that whenever we take a step inside a substitution we maintain the overall structure of the term disregarding the contents of substitutions. Thus, a nested induction on the strong normalization of the term and the strong normalization of all substitutions occurring within the term could presumably extend the strong normalization to this slightly more general reduction relation.

Going further, we can consider an extension of the reduction relation that allows the unrestricted congruence rule at most k times in each step for some fixed number k. As we saw in the counter-example, the non-terminating reduction sequence involved deeper and deeper nesting of **beta** steps using the unrestricted closure congruence rule. This means that having such a fixed limit k is still going to rule out this particular counter-example.

In this section we will formalize these ideas and prove the extended reduction relation strongly normalizing on well-typed terms and substitutions, thereby pushing the boundary of strong normalization right up to the limit set by the counter-example.

We will in the following assume that every term and substitution is well-typed.

2.7.1 The extended reduction relation

We will define a reduction relation $\xrightarrow{k}$ for each natural number $k \geq 0$ that allows k applications of the unrestricted congruence rule.

The reduction relation $\xrightarrow{k}$ is shown in its entirety in Figure 2.5. Notice that, for $k = 0$ the relation coincides with our regular reduction relation $\xrightarrow{0} = \rightarrow$, and of course a larger value of k allows more reductions $\xrightarrow{k} \subset \xrightarrow{k+1}$.

We will denote the subrelation of $\xrightarrow{k}$ that uses one of the rules **clos1** or **clos2** at least once as $\xrightarrow{k}_{[]}$[2]. The subrelation that uses neither **clos1** nor **clos2** is denoted $\xrightarrow{k}_{\cancel{[]}}$[3], such that $\xrightarrow{k} = \xrightarrow{k}_{[]} \cup \xrightarrow{k}_{\cancel{[]}}$. Since $\xrightarrow{k}_{\cancel{[]}}$ is independent of the value of k we will also write this relation as $\rightarrow_{\cancel{[]}}$.

2.7.2 Strong normalization of the extended reduction relation

We will prove $\xrightarrow{k}$ strongly normalizing for terms and substitutions by induction on k. For $k = 0$ the relation is equal to $\rightarrow$ and therefore strongly normalizing by Theorem 2.6.16 and Theorem 2.6.18.

In the following we will prove the induction step.

If we first consider $\xrightarrow{k+1}_{[]}$ then every step uses **clos1**. And this relation is therefore strongly normalizing by the simultaneous induction on the strong normalization of every substitution occurring inside the term, since these substitutions are in turn strongly normalizing by the induction on k.

[2] Read $\xrightarrow{k}_{[]}$ as "k-step-closure", the subscript closure brackets $[]$ should remind you that the reduction must happen inside a closure.

[3] Read $\xrightarrow{k}_{\cancel{[]}}$ as "k-step-no-closure", the subscript crossed-over closure brackets $\cancel{[]}$ should remind you that the reduction must *not* happen inside a closure.

beta	$(\lambda M)\ N$	$\xrightarrow{k}$	$M[N \mathbin{.} \mathsf{id}]$
clos-const	$c[s]$	$\xrightarrow{k}$	c
clos-var-dot1	$1[M \mathbin{.} s]$	$\xrightarrow{k}$	M
clos-var-dot2	$(n+1)[M \mathbin{.} s]$	$\xrightarrow{k}$	$n[s]$
clos-var-shift	$n[\uparrow^m]$	$\xrightarrow{k}$	$n+m$
clos-clos	$M[s][t]$	$\xrightarrow{k}$	$M[s \circ t]$
clos-lam	$(\lambda M)[s]$	$\xrightarrow{k}$	$\lambda(M[1 \mathbin{.} (s \circ \uparrow^1)])$
clos-app	$(M\ N)[s]$	$\xrightarrow{k}$	$M[s]\ N[s]$
comp-id-L	$\uparrow^0 \circ s$	$\xrightarrow{k}$	s
comp-cons	$(M \mathbin{.} s) \circ t$	$\xrightarrow{k}$	$M[t] \mathbin{.} (s \circ t)$
comp-shift-dot	$\uparrow^{n+1} \circ (M \mathbin{.} s)$	$\xrightarrow{k}$	$\uparrow^n \circ s$
comp-shift-shift	$\uparrow^n \circ \uparrow^m$	$\xrightarrow{k}$	$\uparrow^{n+m}$
comp-comp	$(s_1 \circ s_2) \circ s_3$	$\xrightarrow{k}$	$s_1 \circ (s_2 \circ s_3)$
eta-subst	$(n+1) \mathbin{.} \uparrow^{n+1}$	$\xrightarrow{k}$	$\uparrow^n$

$$\frac{M \xrightarrow{k} M'}{M \mathbin{.} s \xrightarrow{k} M' \mathbin{.} s} \qquad \frac{s \xrightarrow{k} s'}{M \mathbin{.} s \xrightarrow{k} M \mathbin{.} s'} \qquad \frac{s \xrightarrow{k} s'}{s \circ t \xrightarrow{k} s' \circ t} \qquad \frac{t \xrightarrow{k} t'}{s \circ t \xrightarrow{k} s \circ t'}$$

$$\frac{M \xrightarrow{k} M'}{\lambda M \xrightarrow{k} \lambda M'} \qquad \frac{M \xrightarrow{k} M'}{M\ N \xrightarrow{k} M'\ N} \qquad \frac{N \xrightarrow{k} N'}{M\ N \xrightarrow{k} M\ N'}$$

$$\frac{M \xrightarrow{k} M'}{M[s] \xrightarrow{k} M'[s]} \qquad \frac{s \xrightarrow{k} s'}{M[s] \xrightarrow{k+1} M[s']}\ \textbf{clos1} \qquad \frac{s \rightarrow_\sigma s'}{M[s] \xrightarrow{0} M[s']}\ \textbf{clos2}$$

Figure 2.5: Reduction rules

Theorem 2.7.1 (`strong-ext.thm:is_sn1_clo,is_sn1s_clo`). *Assuming that $\xrightarrow{k}$ is strongly normalizing on substitutions, the relation $\xrightarrow{k+1}_{[]}$ is strongly normalizing on terms and substitutions.*

Assume we have $M_1 \xrightarrow{k+1}{}^*_{[]} M_2$ and we wish to prove M_2 strongly normalizing with respect to $\xrightarrow{k+1}$. Any $\xrightarrow{k+1}$ step taken by M_2 is either a $\xrightarrow{k+1}_{[]}$ step or a $\xrightarrow{k+1}_{\not[]}$ step. In the first case we again have $M_1 \xrightarrow{k+1}{}^*_{[]} M_2'$ with the same M_1, and this case cannot happen indefinitely since $\xrightarrow{k+1}_{[]}$ is strongly normalizing. In the second case we have $M_1 \xrightarrow{k+1}{}^*_{[]} M_2 \rightarrow_{\not[]} M_2'$. Since the steps from M_1 to M_2 only occur inside substitutions and the step from M_2 to M_2' only occurs outside substitutions, the idea is to prove that these two parts commute in some sense. That is, if we could prove $M_1 \rightarrow M_1' \xrightarrow{k+1}{}^*_{[]} M_2'$ then we could appeal to induction on the strong normalization of M_1 with respect to $\rightarrow$. Unfortunately this is not always the case, but this idea will lead us to something similar, which in the end will get us there.

One of the hard cases turns out to be $1[s_1] \xrightarrow{k+1}{}^*_{[]} 1[M_2' \mathbin{.} s_2] \rightarrow_{\not[]} M_2'$ since this only gives us $s_1 \xrightarrow{k}{}^* M_2' \mathbin{.} s_2$ to work with. To deal with this case (and a few similar cases) we will introduce a weak head normalization function for substitution compositions.

The functions $\mathsf{wcomp}(s)$ and $\mathsf{wcomp}(n; s)$ compute a weak head normal form of s and $\uparrow^n \circ s$, respectively. They are defined as follows and easily seen to be total.

$$
\begin{array}{lcl}
\mathsf{wcomp}(s) & = & \mathsf{wcomp}(0; s) \\
\mathsf{wcomp}(n_1; \uparrow^{n_2}) & = & \uparrow^{n_1+n_2} \\
\mathsf{wcomp}(0; M \,.\, s) & = & M \,.\, s \\
\mathsf{wcomp}(n+1; M \,.\, s) & = & \mathsf{wcomp}(n; s) \\
\mathsf{wcomp}(n; s_1 \circ s_2) & = & \textbf{case}\ \mathsf{wcomp}(n; s_1)\ \textbf{of} \\
 & & \quad \uparrow^{n'} \Rightarrow \mathsf{wcomp}(n'; s_2) \\
 & & \quad M \,.\, s \Rightarrow M[s_2] \,.\, (s \circ s_2)
\end{array}
$$

Since wcomp plays a bit with the associativity of composition, the following theorems are not entirely trivial, but nevertheless true.

Theorem 2.7.2 (`strong-ext.thm:wcomp_to_msteps_su0`). *If* $\mathsf{wcomp}(n; s) = s'$ *then* $\uparrow^n \circ s \rightarrow_\sigma^* s'$.

Theorem 2.7.3 (`strong-ext.thm:wcomp0_to_msteps_su0`). *If* $\mathsf{wcomp}(s) = s'$ *then* $s \rightarrow_\sigma^* s'$.

The definition of wcomp is designed to do as little as possible. In particular, for $s_1 \circ s_2$ it avoids any reduction in s_2 whenever possible. This means that wcomp computes the least possibly reduced weak head normal form in a sense made precise by the following theorem. In particular, any reduction to a substitution s_2 with $\mathsf{wcomp}(s_2) = s_2$ factors through wcomp.

Theorem 2.7.4 (`strong-ext.thm:commute_wcomp_mstep1`). *If* $s_1 \xrightarrow{m}{}^* s_2$ *then* $\mathsf{wcomp}(n; s_1) \xrightarrow{m+1}{}^* \mathsf{wcomp}(n; s_2)$.

Returning our attention to the case $1[s_1] \xrightarrow{m}{}^*_{\Box} 1[M \,.\, s_2] \rightarrow_{\not\Box} M$, we can apply Theorem 2.7.4 to $s_1 \xrightarrow{m}{}^* M \,.\, s_2$ and thereby get a → reduction step of $1[s_1]$ to some M' with $M' \xrightarrow{m+1}{}^* M$. This deals with most of the otherwise problematic cases and allows us to prove the following theorem.

Theorem 2.7.5 (`strong-ext.thm:commute_clo_noc`). *If* $M_1 \xrightarrow{m}{}^*_{\Box} M_2 \rightarrow_{\not\Box} M_3$ *then there exists an* M *such that* $M_1 \rightarrow^+ M \xrightarrow{m+1}{}^* M_3$.

Theorem 2.7.5 presents the following diagram:

$$\begin{tikzcd}
M_1 \arrow[r, "m"{above}, "*"{near end, above right}] \arrow[d, "+"] & {}_{\Box}\, M_2 \arrow[d, "\not\Box"] \\
M \arrow[r, "m+1", "*"{near end, above right}] & M_3
\end{tikzcd}$$

Unfortunately, the bottom arrow is a general reduction sequence $M \xrightarrow{m+1}{}^* M_3$ and not a reduction sequence inside closures $M \xrightarrow{m+1}{}^*_{\Box} M_3$. The reduction sequence from M to M_3 can be divided into $\xrightarrow{m+1}_{\Box}$ steps and $\rightarrow_{\not\Box}$ steps, so if it is not entirely consisting of $\xrightarrow{m+1}_{\Box}$ steps, we can split it as $M \xrightarrow{m+1}{}^*_{\Box} M_4 \rightarrow_{\not\Box} M_5 \xrightarrow{m+1}{}^* M_3$ and apply Theorem 2.7.5 to the left half:

$$\begin{tikzcd}
M \arrow[r, "m+1", "{}^{*}_{\square}"{pos=1}] \arrow[d, "+"{pos=1}] & M_4 \arrow[d, "\not\square"{pos=1}] & \\
M' \arrow[r, "m+2", "{}^{*}"{pos=1}] & M_5 \arrow[r, "m+1", "{}^{*}"{pos=1}] & M_3
\end{tikzcd}$$

We can repeat this construction on $M' \xrightarrow{m+2}{}^{*} M_3$ and since M is strongly normalizing with respect to → we will eventually reach $M \to^{+} M'' \xrightarrow{m'}{}^{*}_{\square} M_3$ for some m'. Thus, we have strengthened Theorem 2.7.5 into:

Theorem 2.7.6 (`strong-ext.thm:commute_clo_noc2`). *If $M_1 \xrightarrow{m}{}^{*}_{\square} M_2 \to_{\not\square} M_3$ then there exists m' and M such that $M_1 \to^{+} M \xrightarrow{m'}{}^{*}_{\square} M_3$.*

Now we can prove $\xrightarrow{k+1}$ strongly normalizing for some given term M_2 by a nested induction on the strong normalization of M_1 with respect to → and the strong normalization of M_2 with respect to $\xrightarrow{k+1}_{\square}$ and the invariant $M_1 \xrightarrow{m}{}^{*}_{\square} M_2$ for some m.

Theorem 2.7.7 (`strong-ext.thm:is_sn1`). *If $\xrightarrow{k+1}_{\square}$ is strongly normalizing then $\xrightarrow{k+1}$ is strongly normalizing for terms.*

Is it easy to refit the proofs of Lemma 2.6.17 and Theorem 2.6.18 to yield strong normalization for substitutions with respect to $\xrightarrow{k+1}$ given strong normalization for terms.

With Theorem 2.7.1 and Theorem 2.7.7 we now have all the pieces to finish the induction on k and prove $\xrightarrow{k}$ strongly normalizing for all k.

Theorem 2.7.8 (`strong-ext.thm:strong_N`). *The reduction relation $\xrightarrow{k}$ is strongly normalizing for well-typed terms and substitutions for all $k \geq 0$.*

Chapter 3

Curry-Style Explicit Substitutions for the Linear and Affine Lambda Calculus

3.1 Summary

In this chapter we introduce a calculus of explicit substitutions for the λ-calculus with linear, affine, and intuitionistic variables and meta-variables. Using a Curry-style formulation, we redesign and extend previously suggested type systems for linear explicit substitutions. This way, we obtain a fine-grained context-split-oblivious small-step reduction semantics suitable for efficient implementation. In particular, we avoid syntactic splitting of substitutions. We prove that subject reduction, confluence, and termination holds. All theorems have been formally verified in the Twelf proof assistant.

All theorems proven in this chapter have been formalized and mechanically checked in the proof assistant Twelf. The formalized theorems are annotated with the corresponding source file in which the proof can be found. The Twelf source files can be downloaded at `http://www.itu.dk/people/anderssn/ex-sub-aff.tgz`.

This chapter has been separately published at IJCAR 2010 [SNS10a].

3.2 Introduction

When denoting a λ-term using de Bruijn indices, we usually use one of two commonly accepted notation styles. Following Church, we encode the simply-typed term $\lambda x : \mathsf{a}.\, x$ (of type $\mathsf{a} \to \mathsf{a}$) as $\lambda \cdot : \mathsf{a}.\, 1$ where 1 refers to the innermost binder occurrence. Following Curry, we simplify this encoding, omit the type label, and write $\lambda 1$. Church style encodings are a bit more verbose, but the real disadvantage lies in the additional code that has to be written to keep type labels up to date, for example in substitution application for dependently typed λ-terms. Therefore, for a real implementation, the brevity of Curry-style notation renders it preferable. However, in reality, it is not as easily adopted, in part because one needs to be sure a priori that type labels are indeed irrelevant or at the very least efficiently inferable whenever necessary using techniques such as, for example, bidirectional type checking.

Further differences between the two notation systems become apparent when we consider linear and affine λ-calculi. A linear λ-term binds a variable that must be used in the body of the term *exactly once*, whereas an affine λ-term binds a variable that can be used *at most once*.

Consider, for example, the term $M \,\widehat{}\, N$ (pronounced M linearly applied to N) of type B in some context Γ. Given the standard formulation of the $\multimap$ elimination rule

$$\frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M \,\widehat{}\, N : B} \multimap \mathsf{E}$$

it seems impossible to derive how to split Γ into Γ_1 and Γ_2 without examining the structure of M or N. Even worse, deriving the context split might be further complicated if M or N contain meta-variables.

When working with explicit substitutions this poses a real problem, since substitutions must correspond to the context they are substituting for. In particular, this means that the reduction $(M \,\widehat{}\, N)[s] \to M[s_1] \,\widehat{}\, N[s_2]$ depends on the context split.

The lack of information on how to split the context in the Curry-style encoding suggests that the alternative Church-style version $M \;\widehat{\Gamma_1 \bowtie \Gamma_2}\; N$ is to be preferred. This version, however, renders terms, types, and contexts mutually dependent, which puts a significant additional burden on the implementation effort. In fact, we suspect that the prospect of the unwieldy complexity is at least in part responsible for that there exist only so few implementations of substructural logical frameworks, theorem provers, and other linear logic based systems.

The problem is not new but has been observed in the past. Cervesato et al. [CdPR99] give a good overview and discuss various suggestions and their shortcomings. But none of the suggestions are satisfactory as they are either too cumbersome to be usable in practice or lack basic properties such as subject reduction. Additionally, none of the suggestions scale to meta-variables.

In this chapter we demonstrate that Curry-style explicit substitutions can be made to work. We define a type system for a linear and affine λ-calculus with explicit substitutions and meta-variables together with a Curry-style reduction semantics. We prove subject reduction, confluence, and termination for our calculus. This means that we can guarantee type preservation of the reduction $(M \,\widehat{}\, N)[s] \to M[s] \,\widehat{}\, N[s]$ without splitting the substitution, and that our calculus therefore permits much more concise data structures in implementations than previously suggested type systems for linear λ-calculi.

The main contributions in this chapter are the subject reduction and confluence theorems. Subject reduction is achieved by a novel type system allowing controlled occurrence of garbage terms (see the rules **typ-cons-ua** and **typ-cons-ul** in Figure 3.3). Confluence on terms with meta-variables is achieved by a limited η-expansion of substitutions.

3.3 Explicit substitutions

We define a calculus of explicit substitutions for the linear and affine λ-calculus by extending $\lambda\sigma$, which was defined in chapter 2. In order to simplify the presentation we restrict attention to the simply typed fragment. A generalization to dependent types is direct, but omitted for presentation purposes. We will sketch the extension to dependent types in section 3.6. Our calculus denotes variables in de Bruijn notation, since it is closer to a real implementation, avoids naming problems due to α-conversion, and highlights the nature of the explicit substitutions.

$$
\begin{array}{lrl}
\textbf{Types:} & A, B ::= & a \mid A \mathbin{\&} B \mid A \multimap B \mid A \mathbin{-\!\!@} B \mid A \rightarrow B \\
\textbf{Terms:} & M, N ::= & 1^f \mid M[s] \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid X[s] \\
 & & \mid \widehat{\lambda} M \mid \mathring{\lambda} M \mid \lambda M \mid M \mathbin{^\frown} N \mid M \mathbin{@} N \mid M\ N \\
\textbf{Substitutions:} & s, t ::= & \mathsf{id} \mid \uparrow \mid M^f \mathbin{.} s \mid s \circ t \\
\textbf{Linearity flags:} & f ::= & \mathbf{I} \mid \mathbf{A} \mid \mathbf{L}
\end{array}
$$

Types consist of base types (a), additive pairs ($A \mathbin{\&} B$), linear functions ($A \multimap B$), affine functions ($A \mathbin{-\!\!@} B$), and intuitionistic functions ($A \rightarrow B$).

Terms consist of the various introduction and elimination forms for each of the type constructs along with variable indices (1^f), meta-variables (X), and closures ($M[s]$). Variables n^f with $n > 1$ do not need to be included explicitly in the syntax, since they can be represented by a closure as described below. We require each meta-variable to be under a closure as it gives rise to more uniform normal forms; an alternative would have been to add the "reduction" rule $X \rightarrow X[\mathsf{id}]$. Meta-variables are also called logic variables. Variables are marked with a flag that indicates whether they reference an intuitionistic, affine, or linear assumption.

Substitutions are composed from identity (id), shifts ($\uparrow$), intuitionistic ($M^{\mathbf{I}}$), affine ($M^{\mathbf{A}}$), and linear ($M^{\mathbf{L}}$) extensions, and an explicit substitution composition. In the interest of readability we introduce a syntactic category f for linearity flags.

Example 3.1. Consider the term $(\widehat{\lambda} 1^{\mathbf{L}} \mathbin{^\frown} 2^{\mathbf{A}}) \mathbin{^\frown} 2^{\mathbf{I}}$ where we write 2^f as a shorthand for $1^f[\uparrow]$. In a named representation, this term is written as $(\widehat{\lambda} x.\, x \mathbin{^\frown} y) \mathbin{^\frown} z$. The $1^{\mathbf{L}}$ refers to the variable bound by the $\widehat{\lambda}$, the $2^{\mathbf{A}}$ to the first free variable (because it is in the scope of the $\widehat{\lambda}$), and the $2^{\mathbf{I}}$ to the second free variable. Notice how the linear flag $\mathbf{L}$ on the variable $1^{\mathbf{L}}$ matches up with the linear binder $\widehat{\lambda}$. This kind of well-formedness will be enforced by the typing rules. Furthermore, the de Bruijn representation specifies that the first free variable is affine and the second is intuitionistic. Explicit substitutions are used to represent an intermediate stage during ordinary β-reduction. Our term reduces to $(1^{\mathbf{L}} \mathbin{^\frown} 2^{\mathbf{A}})[2^{\mathbf{IL}} \mathbin{.} \mathsf{id}]$ where the $\mathbf{L}$ flag in the substitution signifies that we are substituting the term $2^{\mathbf{I}}$ for a linear variable.

The intuition behind each of the substitution constructs is the following: A term under an identity is supposed to reduce to itself. A shift applied to a term increments all freely occurring variables by one. An extension $M^f \mathbin{.} s$ will substitute M for the variable $1^{f'}$ (the typing rules will ensure that f and f' are equal), decrement all other freely occurring variables by one, and then apply s to them. Finally a composition of two substitutions $s \circ t$ represents the substitution that first applies s and then t, i.e. $M[s \circ t]$ is supposed to reduce to the same term as reducing each closure individually in $M[s][t]$.

We will use $\uparrow^n$ where $n \geq 0$ as a short-hand for n compositions of shift, i.e. $\uparrow \circ (\uparrow \circ (\ldots \circ (\uparrow \circ \uparrow) \ldots))$, where $\uparrow^0$ means id. Additionally, de Bruijn indices n^f with $n > 1$ are short-hand for $1^f[\uparrow^{n-1}]$.

3.4 The type system

Before we get to the actual typing judgments we introduce contexts.

3.4.1 Contexts

Since we are using de Bruijn indices, contexts are simply ordered lists of types without any names. Looking up a variable n in a context Γ amounts to selecting the nth element of Γ. This means that the usual context splitting from named versions of linear λ-calculus has to be redefined, as this would otherwise ruin the meaning of de Bruijn variables. Essentially we have to introduce dummy elements in the context whenever we split, in order to maintain the correct position of every type in the context. These dummy elements are in some presentations [CdPR99] written as $\Gamma \widehat{,} \_$. Alternatively, one may view this as never actually splitting the context, but instead maintaining a bit-vector with the same length as the context, which indicates whether each particular variable is available.

For a concise representation we will superimpose the bit-vector signifying availability on the context along with the information about whether the declaration is linear, affine, or intuitionistic. This information is called a context linearity flag.

$$\begin{array}{ll} \textbf{Contexts:} & \Gamma ::= \cdot \mid \Gamma, A^l \\ \textbf{Context linearity flags:} & l ::= f \mid \mathbf{U_L} \mid \mathbf{U_A} \end{array}$$

The usual linear, affine, and intuitionistic assumptions are written as $\Gamma, A^{\mathbf{L}}$, $\Gamma, A^{\mathbf{A}}$, and $\Gamma, A^{\mathbf{I}}$, respectively. We flag a declaration with $\mathbf{U_A}$ to denote that an affine assumption is not available, and similarly flag unavailable linear assumptions with $\mathbf{U_L}$.

The standard definition of the context splitting judgment $\Gamma = \Gamma_1 \bowtie \Gamma_2$ (or context joining judgment depending on the direction it is being read) is shown in Figure 3.1.

We will introduce a couple of auxiliary definitions to do with context splitting and context linearity flag management. Any context may be trivially split into $\Gamma = \Gamma \bowtie \Gamma'$ by putting all affine and linear assumptions to the left. This means that Γ' will consist of only the intuitionistic parts of Γ. We will denote this by $\overline{\Gamma}$ and make it into a separate definition.

$$\overline{\cdot} = \cdot \qquad \overline{\Gamma, A^{\mathbf{I}}} = \overline{\Gamma}, A^{\mathbf{I}} \qquad \overline{\Gamma, A^{\mathbf{L}}} = \overline{\Gamma}, A^{\mathbf{U_L}} \qquad \overline{\Gamma, A^{\mathbf{A}}} = \overline{\Gamma}, A^{\mathbf{U_A}}$$

$$\overline{\Gamma, A^{\mathbf{U_L}}} = \overline{\Gamma}, A^{\mathbf{U_L}} \qquad \overline{\Gamma, A^{\mathbf{U_A}}} = \overline{\Gamma}, A^{\mathbf{U_A}}$$

We will need to make reference to the largest context that can split to a given context. This is denoted $\underline{\Gamma}$ and defined easily by changing every $\mathbf{U_L}$ to $\mathbf{L}$ and $\mathbf{U_A}$ to $\mathbf{A}$.

$$\underline{\cdot} = \cdot \qquad \underline{\Gamma, A^{\mathbf{I}}} = \underline{\Gamma}, A^{\mathbf{I}} \qquad \underline{\Gamma, A^{\mathbf{L}}} = \underline{\Gamma}, A^{\mathbf{L}} \qquad \underline{\Gamma, A^{\mathbf{A}}} = \underline{\Gamma}, A^{\mathbf{A}}$$

$$\frac{}{\cdot = \cdot \bowtie \cdot}\;\textbf{join-nil} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{I}} = \Gamma_1, A^{\mathbf{I}} \bowtie \Gamma_2, A^{\mathbf{I}}}\;\textbf{join-i}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{U_L}} = \Gamma_1, A^{\mathbf{U_L}} \bowtie \Gamma_2, A^{\mathbf{U_L}}}\;\textbf{join-l-used}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{L}} = \Gamma_1, A^{\mathbf{L}} \bowtie \Gamma_2, A^{\mathbf{U_L}}}\;\textbf{join-l-L} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{L}} = \Gamma_1, A^{\mathbf{U_L}} \bowtie \Gamma_2, A^{\mathbf{L}}}\;\textbf{join-l-R}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{U_A}} = \Gamma_1, A^{\mathbf{U_A}} \bowtie \Gamma_2, A^{\mathbf{U_A}}}\;\textbf{join-a-used}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{A}} = \Gamma_1, A^{\mathbf{A}} \bowtie \Gamma_2, A^{\mathbf{U_A}}}\;\textbf{join-a-L} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{A}} = \Gamma_1, A^{\mathbf{U_A}} \bowtie \Gamma_2, A^{\mathbf{A}}}\;\textbf{join-a-R}$$

Figure 3.1: Context splitting

$$\underline{\Gamma, A^{\mathbf{U_L}}} = \underline{\Gamma}, A^{\mathbf{L}} \qquad \underline{\Gamma, A^{\mathbf{U_A}}} = \underline{\Gamma}, A^{\mathbf{A}}$$

Additionally, we will need a predicate on contexts specifying that there are no linear assumptions. We write this predicate as $\mathsf{nolin}(\Gamma)$ and it is defined to be true iff no context linearity flag in Γ is $\mathbf{L}$.

Notice that $\Gamma = \overline{\Gamma}$ implies $\mathsf{nolin}(\Gamma)$ whereas the opposite does not hold, since $\mathsf{nolin}(\Gamma)$ does not preclude occurrences of $\mathbf{A}$ in Γ.

Finally, we need an affine weakening relation $\Gamma \succ_{\text{aff}} \Gamma'$, which is defined as

$$\Gamma \succ_{\text{aff}} \Gamma' \;\equiv\; \exists \Gamma''.\; \Gamma = \Gamma'' \bowtie \Gamma' \;\wedge\; \mathsf{nolin}(\Gamma'')$$

Notice that affine weakening is reflexive and transitive, as it merely amounts to changing some number of $\mathbf{A}$s into $\mathbf{U_A}$s.

3.4.2 Types

The typing judgments for terms and substitutions are denoted $\Gamma \vdash M : A$ and $\Gamma \vdash s : \Gamma'$, respectively. The typing rules are shown in Figures 3.2 and 3.3. In both cases the Γ describes the types and availability of the free variables, and in the case of substitution typing, Γ' describes the context that s substitutes for.

Each meta-variable X carries its own context Γ_X and type A_X as referenced in the typing rule **typ-metavar**. This is equivalent to introducing a new contextual modal context [NPP08] of the form $\Psi ::= \cdot \mid \Psi, (X :: A \;\mathsf{in}\; \Gamma)$. In order to avoid clutter, we omit this additional context from the judgments, since it would simply remain constant and be copied everywhere. Instead we keep the lookup implicit by writing Γ_X and A_X to mean that $X :: A_X \;\mathsf{in}\; \Gamma_X$ is in Ψ.

The **typ-cons-i**, **typ-cons-a**, and **typ-cons-l** are the natural typing rules for substitution extensions. A Church-style explicit substitution calculus would then add two new syntactic substitution constructs, $\bot^{\mathbf{U_L}}.s$ and $\bot^{\mathbf{U_A}}.s$, to correspond to the contexts $\Gamma', A^{\mathbf{U_L}}$ and $\Gamma', A^{\mathbf{U_A}}$ with the following typing rules.

$$\frac{\Gamma \vdash s : \Gamma'}{\Gamma \vdash \bot^{\mathbf{U_L}} \,.\, s : \Gamma', A^{\mathbf{U_L}}}\;\textbf{typ-cons-ul'} \qquad \frac{\Gamma \vdash s : \Gamma'}{\Gamma \vdash \bot^{\mathbf{U_A}} \,.\, s : \Gamma', A^{\mathbf{U_A}}}\;\textbf{typ-cons-ua'}$$

$$\frac{\mathsf{nolin}(\Gamma)}{\Gamma, A^f \vdash 1^f : A}\,\textbf{typ-var} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \mathbin{\&} B}\,\textbf{typ-pair}$$

$$\frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathsf{fst}\ M : A}\,\textbf{typ-fst} \qquad \frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathsf{snd}\ M : B}\,\textbf{typ-snd}$$

$$\frac{\Gamma, A^{\mathbf{L}} \vdash M : B}{\Gamma \vdash \widehat{\lambda} M : A \multimap B}\,\textbf{typ-lam-l} \qquad \frac{\Gamma, A^{\mathbf{A}} \vdash M : B}{\Gamma \vdash \mathring{\lambda} M : A \mathbin{-@} B}\,\textbf{typ-lam-a}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma \vdash M \mathbin{\hat{}} N : B}\,\textbf{typ-app-l}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_2) \quad \Gamma_1 \vdash M : A \mathbin{-@} B \quad \Gamma_2 \vdash N : A}{\Gamma \vdash M \mathbin{@} N : B}\,\textbf{typ-app-a}$$

$$\frac{\Gamma, A^{\mathbf{I}} \vdash M : B}{\Gamma \vdash \lambda M : A \to B}\,\textbf{typ-lam-i} \qquad \frac{\Gamma \vdash M : A \to B \quad \overline{\Gamma} \vdash N : A}{\Gamma \vdash M\ N : B}\,\textbf{typ-app-i}$$

$$\frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash M : A}{\Gamma \vdash M[s] : A}\,\textbf{typ-clos} \qquad \frac{\Gamma \vdash s : \Gamma_X}{\Gamma \vdash X[s] : A_X}\,\textbf{typ-metavar}$$

Figure 3.2: Typing of terms

The application of these substitution constructs read as follows: do not substitute anything for the variable 1 (since it does not occur), but decrement all free variables by one and apply s. This approach has been tried before [GdPR00] but yields the problem described in the introduction (section 3.2), i.e. the reduction $(M \mathbin{\hat{}} N)[s] \to M[s_1] \mathbin{\hat{}} N[s_2]$ needs to split s according to the context split and thus cannot be performed without this additional information.

Our solution to this problem is to reuse the syntax $M^{\mathbf{L}}.s$ and $M^{\mathbf{A}}.s$ where we would expect $\bot^{\mathbf{U_L}} . s$ and $\bot^{\mathbf{U_A}} . s$. The idea is then to perform the substitution splitting on the typing judgments instead of the syntax — this is the crucial switch from the Church-style approach to our Curry-style formulation. However, we cannot reuse the typing rules **typ-cons-ul'** and **typ-cons-ua'**, since this would leave M untyped and prevent us from proving termination. And we cannot just require M to be typed in some context, since M can potentially violate linearity constraints. We therefore introduce a relaxed typing judgment $\Gamma \vdash_i M : A$ and get the typing rules **typ-cons-ul** and **typ-cons-ua**. This relaxed judgment is similar to $\Gamma \vdash M : A$ except that it makes all variables available everywhere disregarding linearity and affineness, i.e. it can be obtained by removing all nolin constraints and by replacing all the relations "$\Gamma = \Gamma_1 \bowtie \Gamma_2$", "$\Gamma \succ_{\text{aff}} \Gamma'$", and "$\Gamma' = \overline{\Gamma}$" by identity relations.

Example 3.2. Consider again the term $(1^{\mathbf{L}} \mathbin{\hat{}} 2^{\mathbf{A}})[2^{\mathbf{IL}}.\mathsf{id}]$ from Example 3.1. If we follow the Church-style system sketched above, this term reduces to $1^{\mathbf{L}}[2^{\mathbf{IL}}.\mathsf{id}] \mathbin{\hat{}} 2^{\mathbf{A}}[\bot^{\mathbf{U_L}}.\mathsf{id}]$, which in a few more steps reduces to the normal form $2^{\mathbf{I}} \mathbin{\hat{}} 1^{\mathbf{A}}$. Note that the two explicit substitutions are syntactically different. In our system, we can however leave the substitution unchanged $1^{\mathbf{L}}[2^{\mathbf{IL}}.\mathsf{id}] \mathbin{\hat{}} 2^{\mathbf{A}}[2^{\mathbf{IL}}.\mathsf{id}]$ and perform splitting only on the type level using the rule **typ-cons-ul** defined in Figure 3.3.

$$\frac{\Gamma \succ_{\mathrm{aff}} \Gamma'}{\Gamma \vdash \mathsf{id} : \Gamma'}\,\textbf{typ-id} \qquad \frac{\Gamma \succ_{\mathrm{aff}} \Gamma' \quad l \in \{\mathbf{I}, \mathbf{A}, \mathbf{U_L}, \mathbf{U_A}\}}{\Gamma, A^{l} \vdash \uparrow : \Gamma'}\,\textbf{typ-shift}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}} \,.\, s : \Gamma', A^{\mathbf{L}}}\,\textbf{typ-cons-l}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_1) \quad \Gamma_1 \vdash M : A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}} \,.\, s : \Gamma', A^{\mathbf{A}}}\,\textbf{typ-cons-a}$$

$$\frac{\underline{\Gamma} \vdash_i M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}} \,.\, s : \Gamma', A^{\mathbf{U_L}}}\,\textbf{typ-cons-ul} \qquad \frac{\underline{\Gamma} \vdash_i M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}} \,.\, s : \Gamma', A^{\mathbf{U_A}}}\,\textbf{typ-cons-ua}$$

$$\frac{\overline{\Gamma} \vdash M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{I}} \,.\, s : \Gamma', A^{\mathbf{I}}}\,\textbf{typ-cons-i} \qquad \frac{\Gamma \vdash s_2 : \Gamma'' \quad \Gamma'' \vdash s_1 : \Gamma'}{\Gamma \vdash s_1 \circ s_2 : \Gamma'}\,\textbf{typ-comp}$$

Figure 3.3: Typing of substitutions

The resulting normal form is of course the same.

The fact that this solves the problem and allows us to split type derivations of substitutions without changing the actual syntactic substitution is shown in Lemma 3.5.1.

Lemma 3.4.1 (`int-typing-lemmas.elf:erase_of,erase_ofs`).

1. *If* $\Gamma \vdash M : A$ *then* $\underline{\Gamma} \vdash_i M : A$.

2. *If* $\Gamma \vdash s : \Gamma'$ *then* $\underline{\Gamma} \vdash_i s : \underline{\Gamma'}$.

Since affine assumptions can be weakened away at all leaves of a typing derivation, we get the following weakening lemma:

Lemma 3.4.2 (`weakening.elf`).

1. *If* $\Gamma_1 \vdash M : A$ *and* $\Gamma_2 \succ_{\mathrm{aff}} \Gamma_1$ *then* $\Gamma_2 \vdash M : A$.

2. *If* $\Gamma_1 \vdash s : \Gamma'$ *and* $\Gamma_2 \succ_{\mathrm{aff}} \Gamma_1$ *then* $\Gamma_2 \vdash s : \Gamma'$.

3. *If* $\Gamma \vdash s : \Gamma'_1$ *and* $\Gamma'_1 \succ_{\mathrm{aff}} \Gamma'_2$ *then* $\Gamma \vdash s : \Gamma'_2$.

3.5 Reduction semantics

The reduction rules defining the semantics are given in Figure 3.4. In order to give a concise presentation of the congruence rules, we use $Y(\!|Z|\!)$ to denote an expression Y with a hole in it, where the hole has been replaced by Z. Note that this is completely syntactical and has no α-equivalence problems as we are using de Bruijn indices.

Most reduction systems with explicit substitutions also include reductions such as $1 \,.\! \uparrow \to \mathsf{id}$ and $1[s] \,.\, (\uparrow \circ s) \to s$, and indeed without them (or something similar) the system is not confluent. However, since these reductions essentially are η-reductions on substitutions, it seems that they should really turn the

beta-l	$(\widehat{\lambda} M) \frown N$	$\rightarrow$	$M[N^{\mathbf{L}} \,.\, \mathsf{id}]$
beta-a	$(\mathring{\lambda} M) \mathbin{@} N$	$\rightarrow$	$M[N^{\mathbf{A}} \,.\, \mathsf{id}]$
beta-i	$(\lambda M)\ N$	$\rightarrow$	$M[N^{\mathbf{I}} \,.\, \mathsf{id}]$
beta-fst	$\mathsf{fst}\langle M, N\rangle$	$\rightarrow$	M
beta-snd	$\mathsf{snd}\langle M, N\rangle$	$\rightarrow$	N
clos-var	$1^f[M^{f'} \,.\, s]$	$\rightarrow$	M
clos-clos	$M[s][t]$	$\rightarrow$	$M[s \circ t]$
clos-metavar	$X[s][t]$	$\rightarrow$	$X[s \circ t]$
clos-pair	$\langle M, N\rangle[s]$	$\rightarrow$	$\langle M[s], N[s]\rangle$
clos-fst	$(\mathsf{fst}\ M)[s]$	$\rightarrow$	$\mathsf{fst}\ (M[s])$
clos-snd	$(\mathsf{snd}\ M)[s]$	$\rightarrow$	$\mathsf{snd}\ (M[s])$
clos-lam-l	$(\widehat{\lambda} M)[s]$	$\rightarrow$	$\widehat{\lambda}(M[1^{\mathbf{LL}} \,.\, (s \circ \uparrow)])$
clos-lam-a	$(\mathring{\lambda} M)[s]$	$\rightarrow$	$\mathring{\lambda}(M[1^{\mathbf{AA}} \,.\, (s \circ \uparrow)])$
clos-lam-i	$(\lambda M)[s]$	$\rightarrow$	$\lambda(M[1^{\mathbf{II}} \,.\, (s \circ \uparrow)])$
clos-app-l	$(M \frown N)[s]$	$\rightarrow$	$M[s] \frown N[s]$
clos-app-a	$(M \mathbin{@} N)[s]$	$\rightarrow$	$M[s] \mathbin{@} N[s]$
clos-app-i	$(M\ N)[s]$	$\rightarrow$	$M[s]\ N[s]$
clos-id	$M[\mathsf{id}]$	$\rightarrow$	M
comp-id-L	$\mathsf{id} \circ s$	$\rightarrow$	s
comp-id-R	$s \circ \mathsf{id}$	$\rightarrow$	s
comp-shift	$\uparrow \circ\, (M^f \,.\, s)$	$\rightarrow$	s
comp-cons	$(M^f \,.\, s) \circ t$	$\rightarrow$	$M[t]^f \,.\, (s \circ t)$
comp-comp	$(s_1 \circ s_2) \circ s_3$	$\rightarrow$	$s_1 \circ (s_2 \circ s_3)$

$$\frac{N \rightarrow N'}{M\llparenthesis N\rrparenthesis \rightarrow M\llparenthesis N'\rrparenthesis}\,\textbf{cong-tm-tm} \qquad \frac{s \rightarrow s'}{M\llparenthesis s\rrparenthesis \rightarrow M\llparenthesis s'\rrparenthesis}\,\textbf{cong-tm-sub}$$

$$\frac{M \rightarrow M'}{s\llparenthesis M\rrparenthesis \rightarrow s\llparenthesis M'\rrparenthesis}\,\textbf{cong-sub-tm} \qquad \frac{t \rightarrow t'}{s\llparenthesis t\rrparenthesis \rightarrow s\llparenthesis t'\rrparenthesis}\,\textbf{cong-sub-sub}$$

Figure 3.4: Reduction rules

other way, enabling us to η-expand substitutions. We will regain confluence by allowing substitution expansion at meta-variables. This allows us to bound the expansion by the context carried by the meta-variable, since this must match the type of the substitution. The substitution expansion rules are given in Figure 3.5 with the **eta-sub** rule extending the rules in Figure 3.4. Notice that the **eta-x-shifts-*** rules are exactly $s \rightarrow 1[s] \,.\, (\uparrow \circ\, s)$ in the case where $s = \uparrow^n$.

3.5.1 Type preservation

Lemma 3.5.1 (`subst-lemmas.elf`)**.** *Substitutions preserve context splits.*

1. *If* $\Gamma \vdash s : \Gamma'$ *and* $\Gamma' = \Gamma'_1 \bowtie \Gamma'_2$ *then there exists* Γ_1 *and* Γ_2 *such that* $\Gamma_1 \vdash s : \Gamma'_1$*,* $\Gamma_2 \vdash s : \Gamma'_2$*, and* $\Gamma = \Gamma_1 \bowtie \Gamma_2$*.*

2. *If* $\Gamma \vdash s : \Gamma'$ *and* $\Gamma' = \Gamma' \bowtie \Gamma'_2$ *then there exists* Γ_2 *such that* $\Gamma_2 \vdash s : \Gamma'_2$ *and* $\Gamma = \Gamma \bowtie \Gamma_2$*.*

$$\begin{array}{llcl}
\textbf{eta-x-shifts-i} & \uparrow^n : \Gamma, A^{\mathbf{I}} & \to_\eta & (n+1)^{\mathbf{II}} \mathbin{.} \uparrow^{n+1} \\
\textbf{eta-x-shifts-a} & \uparrow^n : \Gamma, A^{\mathbf{A}} & \to_\eta & (n+1)^{\mathbf{AA}} \mathbin{.} \uparrow^{n+1} \\
\textbf{eta-x-shifts-ua} & \uparrow^n : \Gamma, A^{\mathbf{U}_{\mathbf{A}}} & \to_\eta & (n+1)^{\mathbf{AA}} \mathbin{.} \uparrow^{n+1} \\
\textbf{eta-x-shifts-l} & \uparrow^n : \Gamma, A^{\mathbf{L}} & \to_\eta & (n+1)^{\mathbf{LL}} \mathbin{.} \uparrow^{n+1} \\
\textbf{eta-x-shifts-ul} & \uparrow^n : \Gamma, A^{\mathbf{U}_{\mathbf{L}}} & \to_\eta & (n+1)^{\mathbf{LL}} \mathbin{.} \uparrow^{n+1}
\end{array}$$

$$\frac{s : \Gamma_X \to_\eta s'}{X[s] \to X[s']}\,\textbf{eta-sub} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{I}} \mathbin{.} s : \Gamma, A^{\mathbf{I}} \to_\eta M^{\mathbf{I}} \mathbin{.} s'}\,\textbf{eta-x-i}$$

$$\frac{s : \Gamma \to_\eta s'}{M^{\mathbf{A}} \mathbin{.} s : \Gamma, A^{\mathbf{A}} \to_\eta M^{\mathbf{A}} \mathbin{.} s'}\,\textbf{eta-x-a} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{A}} \mathbin{.} s : \Gamma, A^{\mathbf{U}_{\mathbf{A}}} \to_\eta M^{\mathbf{A}} \mathbin{.} s'}\,\textbf{eta-x-ua}$$

$$\frac{s : \Gamma \to_\eta s'}{M^{\mathbf{L}} \mathbin{.} s : \Gamma, A^{\mathbf{L}} \to_\eta M^{\mathbf{L}} \mathbin{.} s'}\,\textbf{eta-x-l} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{L}} \mathbin{.} s : \Gamma, A^{\mathbf{U}_{\mathbf{L}}} \to_\eta M^{\mathbf{L}} \mathbin{.} s'}\,\textbf{eta-x-ul}$$

Figure 3.5: Substitution expansion

3. *If* $\Gamma \vdash s : \Gamma'$ *and* $\mathsf{nolin}(\Gamma')$ *then* $\mathsf{nolin}(\Gamma)$.

Notice that the second part of Lemma 3.5.1 is equivalent to the statement that $\Gamma \vdash s : \Gamma'$ implies $\overline{\Gamma} \vdash s : \overline{\Gamma'}$. Also it might be tempting to try and prove the second part from the first, but this does not work since $\Gamma_2 \vdash s : \overline{\Gamma'}$ does not imply $\Gamma_2 = \overline{\Gamma_2}$ due to the possible existence of affine assumptions in Γ_2.

With this lemma we can now prove type-preservation:

Theorem 3.5.2 (`preservation-thm.elf:pres,press,pres-expand_sub`). *The reduction relation* $\to$ *is type-preserving.*

1. *If* $\Gamma \vdash M : A$ *and* $M \to M'$ *then* $\Gamma \vdash M' : A$.
2. *If* $\Gamma \vdash s : \Gamma'$ *and* $s \to s'$ *then* $\Gamma \vdash s' : \Gamma'$.
3. *If* $\Gamma \vdash s : \Gamma'$ *and* $s : \Gamma' \to_\eta s'$ *then* $\Gamma \vdash s' : \Gamma'$.

3.5.2 σ-reduction

Before we prove confluence and termination of the entire calculus, we deal with substitutions. This will allow us to reduce the confluence and termination proofs to the case of the ordinary λ-calculus.

Consider the reduction relation without the **beta-*** rules. Just as we did in chapter 2, we will call this sub-relation σ-reduction and denote it $\to_\sigma$. A term or substitution that cannot σ-reduce is said to be in σ-normal form. We write this as the postfix predicate $\not\to_\sigma$.

Theorem 3.5.3 (`signf-exists.elf:snf1-exists,ssnf1-exists`). *σ-reduction is terminating.*

1. *For all terms* M *there exists a term* M' *such that* $M \to_\sigma^* M'$ *and* $M' \not\to_\sigma$.
2. *For all substitutions* s *there exists a substitution* s' *such that* $s \to_\sigma^* s'$ *and* $s' \not\to_\sigma$.

$$\frac{}{\uparrow^n \sim_\eta \uparrow^n} \qquad \frac{s \sim_\eta s' \quad M \not\to_\sigma}{M^f \,.\, s \sim_\eta M^f \,.\, s'}$$

$$\frac{s \sim_\eta s'}{s' \sim_\eta s} \qquad \frac{\uparrow^{n+1} \sim_\eta s}{\uparrow^n \sim_\eta (n+1)^{ff} \,.\, s}$$

Figure 3.6: η-equivalence of substitutions

Furthermore typed σ-reduction is confluent, which is equivalent to having unique normal forms, since it is terminating.[1]

For typed terms M and substitutions s we will denote their unique σ-normal forms $\sigma(M)$ and $\sigma(s)$, respectively.

Theorem 3.5.4 (`signf-uniq.elf:sigma-conf,sigma-confs`). *Typed σ-reduction is confluent.*

1. *If $\Gamma \vdash M : A$, $M \to_\sigma^* M_1$, and $M \to_\sigma^* M_2$ then there exists a term M' such that $M_1 \to_\sigma^* M'$ and $M_2 \to_\sigma^* M'$.*

2. *If $\Gamma \vdash s : \Gamma'$, $s \to_\sigma^* s_1$, and $s \to_\sigma^* s_2$ then there exists a substitution s' such that $s_1 \to_\sigma^* s'$ and $s_2 \to_\sigma^* s'$.*

Having confluence of σ-reduction gives us a lot of nice algebraic properties. One of the most important properties for the formalizations of the proofs is $\sigma(\sigma(M[s])[t]) = \sigma(M[s \circ t])$, which shows that substitution composition indeed behaves as expected. But since we have restricted η-conversions on substitutions we get no immediate corollaries from σ-confluence concerning η-equivalences. To remedy this we will first define η-equivalence on substitutions in σ-normal form and then show that η-equivalent substitutions indeed behave identically. The definition of η-equivalence is given in Figure 3.6. The following lemma shows that $\sim_\eta$ is an equivalence relation on typed substitutions since symmetry is given by definition.

Lemma 3.5.5 (`signf-equiv.elf:equiv_ssnf_refl,equiv_ssnf_trans`). *Reflexivity and transitivity for η-equivalence of substitutions.*

1. *If $s \not\to_\sigma$ then $s \sim_\eta s$.*

2. *If $\Gamma \vdash s_1 : \Gamma'$, $\Gamma \vdash s_2 : \Gamma'$, $\Gamma \vdash s_3 : \Gamma'$, $s_1 \sim_\eta s_2$, and $s_2 \sim_\eta s_3$ then $s_1 \sim_\eta s_3$.*

Lemma 3.5.6 (`signf-equiv.elf:expand2equiv`). *η-equivalence of substitutions contains η-expansion.*

If $s : \Gamma \to_\eta s'$ and $s \not\to_\sigma$ then $s \sim_\eta s'$.

Theorem 3.5.7 (`signf-equiv.elf:signf-equiv,ssignf-equiv`). *Substitutions that are η-equivalent behave the same way with respect to σ-reduction.*

If $\Gamma' \vdash M : A$, $\Gamma' \vdash t : \Gamma''$, $\Gamma \vdash s : \Gamma'$, $\Gamma \vdash s' : \Gamma'$, and $s \sim_\eta s'$ then:

1. *$\sigma(M[s]) = \sigma(M[s'])$*

2. *$\sigma(t \circ s) \sim_\eta \sigma(t \circ s')$*

[1] The reason why we need types is because confluence relies on η-expansion of substitutions.

$$\begin{tikzcd}
& & M \arrow[dll, "R_1^*"'] \arrow[d, "R_1^*"] \arrow[drr, "R_1^*"] & & \\
M_1 \arrow[d, "R_1^*"'] & & f(M) \arrow[dll, "R_2^*"'] \arrow[drr, "R_2^*"] & & M_2 \arrow[d, "R_1^*"] \\
f(M_1) \arrow[drr, "R_2^*"] & & & & f(M_2) \arrow[dll, "R_2^*"'] \\
& & M' & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & M \arrow[dll] \arrow[d] \arrow[drr] & & \\
M_1 \arrow[d] & & \sigma(M) \arrow[dll, "\beta^*"'] \arrow[drr, "\beta^*"] & & M_2 \arrow[d] \\
\sigma(M_1) \arrow[drr, "\beta^*"] & & & & \sigma(M_2) \arrow[dll, "\beta^*"'] \\
& & M' & &
\end{tikzcd}$$

Figure 3.7: Confluence by the interpretation method

3.5.3 Confluence and termination

Now we have the tools necessary to prove confluence of the entire reduction relation.

Lemma 3.5.8 (Generalized Interpretation Method)**.** *Given two sets $B \subseteq A$, reduction relations R_1 and R_2 on A and B, respectively, and a function $f : A \to B$ such that:*

1. $R_2 \subseteq R_1^*$.
2. $\forall M \in A :\ M \to_{R_1}^* f(M)$.
3. $\forall M, M' \in A :\ M \to_{R_1}^* M' \Rightarrow f(M) \to_{R_2}^* f(M')$.

Then confluence of R_2 implies confluence of R_1.

Proof. The proof is by simple diagram chasing as shown in Figure 3.7 on the left. □

To prove confluence of the entire reduction relation we use Lemma 3.5.8 with σ-normalization as the interpreting function f. As R_2 we take ordinary β-reduction on σ-normal forms, which in our case can be defined as one of the **beta-*** rules followed by σ-normalization. The situation is illustrated in Figure 3.7 on the right. First everything is σ-normalized, then σ-normalization is shown to preserve β-reduction, and finally confluence of the usual λ-calculus is used to conclude confluence of our calculus.

Theorem 3.5.9 (`confluence.elf`)**.** *The reduction relation $\to$ is confluent on typed terms and substitutions.*

1. *If $\Gamma \vdash M : A$, $M \to^* M_1$, and $M \to^* M_2$ then there exists a term M' such that $M_1 \to^* M'$ and $M_2 \to^* M'$.*
2. *If $\Gamma \vdash s : \Gamma'$, $s \to^* s_1$, and $s \to^* s_2$ then there exists a substitution s' such that $s_1 \to^* s'$ and $s_2 \to^* s'$.*

Similarly we can also prove termination of $\rightarrow$ from termination of the usual λ-calculus by σ-normalization.

Theorem 3.5.10 (`nf-exists.elf:nf-exists,ns-exists`)**.** *The reduction relation $\rightarrow$ is terminating on typed terms and substitutions.*

1. *If $\Gamma \vdash M : A$ then there exists a term M' such that $M \twoheadrightarrow M'$ and $M' \not\rightarrow$.*

2. *If $\Gamma \vdash s : \Gamma'$ then there exists a substitution s' such that $s \twoheadrightarrow s'$ and $s' \not\rightarrow$.*

The normal forms for the reduction relation $\rightarrow$ are called β-normal forms and because of Theorems 3.5.9 and 3.5.10 we know that they exist and are unique.

3.6 Dependent types

The presented calculus can easily be extended to dependent types [Muñ98], since this extension is orthogonal to linear and affine types. We will sketch the extension here.

The base types a and intuitionistic function types $A \rightarrow B$ are replaced by type families $a\ M_1 \ldots M_n$ and dependent functions $\Pi A.\, B$. The type system is extended with a judgment $\Gamma \vdash A : \mathsf{type}$ specifying that the type A is well-formed in the context Γ and a well-formedness requirement on contexts stating that whenever we form the context Γ, A^l we must have $\overline{\Gamma} \vdash A : \mathsf{type}$. The system is then tied together by the central invariant stating that whenever $\Gamma \vdash M : A$ then $\overline{\Gamma} \vdash A : \mathsf{type}$.

The typing rules of the system become slightly more complicated to account for the objects occurring in the types. As an example of one of the updated rules, consider **typ-cons-l** which becomes:

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A[s] \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}} \mathbin{.} s : \Gamma', A^{\mathbf{L}}}\ \textbf{deptyp-cons-l}$$

Notice that $\overline{\Gamma_1} = \overline{\Gamma_2}$ and that Lemma 3.5.1 gives us $\overline{\Gamma_2} \vdash s : \overline{\Gamma'}$ to ensure that $A[s]$ is well-formed in the right context.

Theorems 3.5.9 and 3.5.10 extend to the dependently typed setting as well, since the proofs can be reused after type erasure.

Chapter 4

Pattern Unification for the Lambda Calculus with Linear and Affine Types

4.1 Summary

We define the pattern fragment for higher-order unification problems in linear and affine type theory and give a deterministic unification algorithm that computes most general unifiers.

We then extend the algorithm to the linear-changing pattern fragment by a procedure called linearity pruning to bridge the gap to the intuitionistic pattern fragment.

A shortened version of this chapter has been separately published at LFMTP 2010 [SNS10b].

4.2 Introduction

Logic programming languages, type inference algorithms, and automated theorem provers are all examples of systems that rely on unification. If the unification problem has to deal with logic variables at higher type (functional type), we speak of higher-order unification [Hue75]. Higher-order unification is in general undecidable, but it can be turned decidable, if appropriately restricted to a fragment. For example, Miller's pattern fragment characterizes a first-order fragment, for which unification is decidable [Mil91].

As substructural type theories are becoming more prevalent, for example, in systems that need to represent consumable resources, higher-order unification algorithms need to deal with logic variables at linear or affine type. Linear and affine type theories refine intuitionistic type theory in the following way: Besides intuitionistic assumptions, which can be referred to an arbitrary number of times, linear and affine assumptions are treated as resources that must be referred to *exactly once* and *at most once*, respectively.

As substructural type theories are mere refinements, one might erroneously suspect that the standard intuitionistic pattern unification algorithm can be applied to this setting directly. This, unfortunately, is not the case. Consider the following two linear unification problems, where we write, as usual, $\hat{}$ for linear application and juxtaposition for intuitionistic application.

$$F \mathbin{\hat{}} x \doteq c \mathbin{\hat{}} (H_1\, x) \mathbin{\hat{}} (H_2\, x) \tag{4.1}$$

$$F \mathbin{\hat{}} x \doteq c \mathbin{\hat{}} (H\, x) \tag{4.2}$$

These examples take place in a context in which x is an intuitionistic variable. However, the linear application on the left-hand side implies that the variable must occur *exactly* once in any valid instantiation of F, but in (4.1) we cannot know whether x should occur in H_1 or H_2. This additional problem over normal intuitionistic higher-order unification is caused exactly by the interaction of linear and intuitionistic variables. We solve this issue by imposing a separation of linear, affine, and intuitionistic variables.

In this chapter, we refine the intuitionistic pattern fragment into a pattern fragment for linear and affine type theory. We describe a unification algorithm for this fragment and prove it correct. Furthermore, we show that in this fragment most general unifiers exist. Finally, we extend the algorithm with a procedure we call *linearity pruning.* This procedure goes beyond the pattern fragment and treats equations such as (4.1) and (4.2) where variables may have to change their status, for example from being affine to linear. Unification problems in this *linear-changing pattern fragment* continue to be decidable. For example, for (4.2) the algorithm finds the most general unifier, which is $F = \widehat{\lambda} x.\, c \mathbin{\hat{}} (G \mathbin{\hat{}} x)$ and $H = \lambda x.\, G \mathbin{\hat{}} x$. Our focus in this chapter is finding unique most general unifiers, and since (4.1) has a set of most general unifiers of size two, we are not going to try to solve it. However, one could easily extend linearity pruning to these cases by considering the finite number of context splits.

Previous approaches to higher-order linear unification have been restricted to highly non-deterministic algorithms, such as the pre-unification by Cervesato and Pfenning [CP97]. In contrast, our algorithm is completely deterministic, and very well suited for implementation. It is the core algorithm of the Celf system (chapter 6).

4.3 The explicit substitution calculus

4.3.1 Canonical forms

In chapter 3 we introduced a calculus of explicit substitutions for the λ-calculus with linear, affine, and intuitionistic variables and logic variables. Along with the calculus we introduced a type system and a reduction semantics, which was proven to be type-preserving, confluent, and terminating. And in chapter 2 we also proved it strongly normalizing when reduction was disallowed inside an arbitrary number of substitutions nested inside each other. We recall the

$$\frac{\mathsf{nolin}(\Gamma)}{\Gamma, A^f \vdash 1^f \Rightarrow A} \qquad \frac{\Gamma \vdash n^f \Rightarrow B \quad l \in \{\mathbf{I}, \mathbf{A}, \mathbf{U_L}, \mathbf{U_A}\}}{\Gamma, A^l \vdash (n+1)^f \Rightarrow B}$$

$$\frac{\Gamma \vdash M \Rightarrow a}{\Gamma \vdash M \Leftarrow a} \qquad \frac{\Gamma \vdash s : \Gamma_X}{\Gamma \vdash X[s] \Rightarrow A_X}$$

$$\frac{\Gamma \vdash M \Leftarrow A \quad \Gamma \vdash N \Leftarrow B}{\Gamma \vdash \langle M, N \rangle \Leftarrow A \,\&\, B} \qquad \frac{\Gamma \vdash M \Rightarrow A \,\&\, B}{\Gamma \vdash \mathsf{fst}\ M \Rightarrow A} \qquad \frac{\Gamma \vdash M \Rightarrow A \,\&\, B}{\Gamma \vdash \mathsf{snd}\ M \Rightarrow B}$$

$$\frac{\Gamma, A^{\mathbf{L}} \vdash M \Leftarrow B}{\Gamma \vdash \widehat{\lambda} M \Leftarrow A \multimap B} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M \Rightarrow A \multimap B \quad \Gamma_2 \vdash N \Leftarrow A}{\Gamma \vdash M \,\hat{}\, N \Rightarrow B}$$

$$\frac{\Gamma, A^{\mathbf{A}} \vdash M \Leftarrow B}{\Gamma \vdash \mathring{\lambda} M \Leftarrow A \mathbin{-\!@} B}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_2) \quad \Gamma_1 \vdash M \Rightarrow A \mathbin{-\!@} B \quad \Gamma_2 \vdash N \Leftarrow A}{\Gamma \vdash M \mathbin{@} N \Rightarrow B}$$

$$\frac{\Gamma, A^{\mathbf{I}} \vdash M \Leftarrow B}{\Gamma \vdash \lambda M \Leftarrow A \to B} \qquad \frac{\Gamma \vdash M \Rightarrow A \to B \quad \overline{\Gamma} \vdash N \Leftarrow A}{\Gamma \vdash M\ N \Rightarrow B}$$

Figure 4.1: Bidirectional typing of terms in canonical form

definition from chapter 3:

Types: $A, B ::= a \mid A \,\&\, B \mid A \multimap B \mid A \mathbin{-\!@} B \mid A \to B$

Terms: $M, N ::= 1^f \mid M[s] \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid X[s]$
$\mid \widehat{\lambda} M \mid \mathring{\lambda} M \mid \lambda M \mid M \,\hat{}\, N \mid M \mathbin{@} N \mid M\ N$

Substitutions: $s, t ::= \mathsf{id} \mid \uparrow \mid M^f \,.\, s \mid s \circ t$

Linearity flags: $f ::= \mathbf{I} \mid \mathbf{A} \mid \mathbf{L}$

Contexts: $\Gamma ::= \cdot \mid \Gamma, A^l$

Context linearity flags: $l ::= f \mid \mathbf{U_L} \mid \mathbf{U_A}$

Each variable 1^f is tagged with a flag signifying whether the variable is intuitionistic, affine, or linear. We use $\uparrow^n$ where $n \geq 0$ as a short-hand for n compositions of shift, i.e. $\uparrow \circ (\uparrow \circ (\ldots \circ (\uparrow \circ \uparrow) \ldots))$, where $\uparrow^0$ means id. Additionally, de Bruijn indices n^f with $n > 1$ are short-hand for $1^f[\uparrow^{n-1}]$. The context linearity flags and the corresponding assumptions in contexts are denoted *intuitionistic* ($\mathbf{I}$), *affine* ($\mathbf{A}$), *used affine* ($\mathbf{U_A}$), *linear* ($\mathbf{L}$), and *used linear* ($\mathbf{U_L}$).

In this chapter we will work exclusively with the corresponding calculus of canonical forms and hereditary substitutions. This can be obtained simply by viewing each term as a short-hand for its unique normal form and assuming that everything is fully η-expanded. The resulting type system is shown in Figure 4.1 and Figure 4.2. We write $\Gamma \vdash M : A$ as a shorthand for either $\Gamma \vdash M \Leftarrow A$ or $\Gamma \vdash M \Rightarrow A$.

The intuitionistic part of a context $\overline{\Gamma}$ is formed by rendering all linear and affine variables unavailable, which corresponds to changing the context linearity

$$\frac{}{\cdot \vdash {\uparrow}^0 : \cdot} \qquad \frac{\Gamma \vdash {\uparrow}^n : \Gamma' \quad l \in \{\mathbf{I}, \mathbf{A}, \mathbf{U_L}, \mathbf{U_A}\}}{\Gamma, A^l \vdash {\uparrow}^{n+1} : \Gamma'} \qquad \frac{\overline{\Gamma} \vdash M \Leftarrow A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{I}} \mathbin{.} s : \Gamma', A^{\mathbf{I}}}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M \Leftarrow A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}} \mathbin{.} s : \Gamma', A^{\mathbf{L}}} \qquad \frac{\underline{\Gamma} \vdash_i M \Leftarrow A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}} \mathbin{.} s : \Gamma', A^{\mathbf{U_L}}}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_1) \quad \Gamma_1 \vdash M \Leftarrow A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}} \mathbin{.} s : \Gamma', A^{\mathbf{A}}}$$

$$\frac{\underline{\Gamma} \vdash_i M \Leftarrow A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}} \mathbin{.} s : \Gamma', A^{\mathbf{U_A}}}$$

Figure 4.2: Typing of substitutions

flags from $\mathbf{L}$ to $\mathbf{U_L}$ and $\mathbf{A}$ to $\mathbf{U_A}$. Similarly, the largest context that can split to a given context is denoted $\underline{\Gamma}$ and constructed by changing every $\mathbf{U_L}$ to $\mathbf{L}$ and $\mathbf{U_A}$ to $\mathbf{A}$. The predicate $\mathsf{nolin}(\Gamma)$ specifies that no linear assumptions occur in Γ, i.e. no flag in Γ is equal to $\mathbf{L}$. The relaxed typing judgment $\Gamma \vdash_i M : A$ is similar to $\Gamma \vdash M : A$ except that it makes all variables available everywhere disregarding linearity and affineness.

Restricting ourselves to canonical forms while retaining the syntax of redices and closures as short-hands for their corresponding normal forms induces equalities corresponding to the rewrite rules of the original system. Similarly ${\uparrow}^n$ is retained as a short-hand for its corresponding η-expanded form inducing the equality ${\uparrow}^n = (n+1)^{ff} \mathbin{.} {\uparrow}^{n+1}$. The induced equalities are shown in Figure 4.3. Additionally, the two typing rules for $M[s]$ and $s_1 \circ s_2$ from Figure 3.2 and Figure 3.3, which are left out, are now simply admissible rules proving type preservation of hereditary substitution:

$$\frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash M : A}{\Gamma \vdash M[s] : A} \qquad \frac{\Gamma \vdash s_2 : \Gamma'' \quad \Gamma'' \vdash s_1 : \Gamma'}{\Gamma \vdash s_1 \circ s_2 : \Gamma'}$$

4.3.2 Spine notation

We use *spine notation* [CP03] as a convenient short-hand for series of applications and projections. Spines are defined by the following grammar:

$$S ::= () \mid M; S \mid M \mathbin{\overset{\circ}{;}} S \mid M \mathbin{\widehat{;}} S \mid \mathsf{fst}; S \mid \mathsf{snd}; S$$

$$
\begin{aligned}
1^f[M^f \,.\, s] &= M & (\widehat{\lambda} M) \mathbin{\hat{}} N &= M[N^{\mathbf{L}} \,.\, \mathsf{id}] \\
M[s][t] &= M[s \circ t] & (\mathring{\lambda} M) \mathbin{@} N &= M[N^{\mathbf{A}} \,.\, \mathsf{id}] \\
X[s][t] &= X[s \circ t] & (\lambda M)\ N &= M[N^{\mathbf{I}} \,.\, \mathsf{id}] \\
\langle M, N\rangle[s] &= \langle M[s], N[s]\rangle & \mathsf{fst}\langle M, N\rangle &= M \\
(\mathsf{fst}\ M)[s] &= \mathsf{fst}\ (M[s]) & \mathsf{snd}\langle M, N\rangle &= N \\
(\mathsf{snd}\ M)[s] &= \mathsf{snd}\ (M[s]) & & \\
(\widehat{\lambda} M)[s] &= \widehat{\lambda}(M[1^{\mathbf{LL}} \,.\, (s \circ {\uparrow})]) & \mathsf{id} \circ s &= s \\
(\mathring{\lambda} M)[s] &= \mathring{\lambda}(M[1^{\mathbf{AA}} \,.\, (s \circ {\uparrow})]) & s \circ \mathsf{id} &= s \\
(\lambda M)[s] &= \lambda(M[1^{\mathbf{II}} \,.\, (s \circ {\uparrow})]) & {\uparrow} \circ (M^f \,.\, s) &= s \\
(M \mathbin{\hat{}} N)[s] &= M[s] \mathbin{\hat{}} N[s] & (M^f \,.\, s) \circ t &= M[t]^f \,.\, (s \circ t) \\
(M \mathbin{@} N)[s] &= M[s] \mathbin{@} N[s] & (s_1 \circ s_2) \circ s_3 &= s_1 \circ (s_2 \circ s_3) \\
(M\ N)[s] &= M[s]\ N[s] & & \\
M[\mathsf{id}] &= M & {\uparrow}^n &= (n+1)^{ff} \,.\, {\uparrow}^{n+1}
\end{aligned}
$$

Figure 4.3: Equalities

The term $M \cdot S$ is short-hand for the term where all the terms and projections in S are applied to M as follows:

$$
\begin{aligned}
M \cdot () &= M \\
M \cdot (N; S) &= (M\ N) \cdot S \\
M \cdot (N \mathring{;} S) &= (M \mathbin{@} N) \cdot S \\
M \cdot (N \mathbin{\widehat{;}} S) &= (M \mathbin{\hat{}} N) \cdot S \\
M \cdot (\mathsf{fst}; S) &= (\mathsf{fst}\ M) \cdot S \\
M \cdot (\mathsf{snd}; S) &= (\mathsf{snd}\ M) \cdot S
\end{aligned}
$$

We define $S[s]$ to be the spine in which s is applied to each term in S. This yields the equality $(M \cdot S)[s] = M[s] \cdot S[s]$.

4.3.3 Instantiation of logic variables

We write $[X \leftarrow N]M$ for the *instantiation of the logic variable* X with term N in term M. This means that all occurrences of $X[s] \cdot S$ are replaced by $N[s] \cdot S$. Note that the latter is a short-hand for its normal form. We will assume that all instantiations are well-typed, i.e. we require that $\Gamma_X \vdash N : A_X$. This requirement ensures that the resulting term $[X \leftarrow N]M$ is well-typed, by induction on M and the subject reduction property of hereditary substitutions.

Theorem 4.3.1. *If* $\Gamma_X \vdash N : A_X$ *and* $\Gamma \vdash M : A$ *then* $\Gamma \vdash [X \leftarrow N]M : A$.

Theorem 4.3.1 is also called the contextual modal cut admissibility theorem for linear and affine contextual modal logic.

4.4 Patterns

The hallmark characteristic of the intuitionistic pattern fragment is the invertibility of substitutions [DHKP98]. Our pattern fragment for the linear and affine calculus that we are going to introduce next continues to guarantee this important property.

4.4.1 The pattern fragment

Consider a substitution $\Gamma \vdash a_1^{f_1} \ldots a_p^{f_p} \, . \uparrow^n \, : \Gamma'$. Assume that a_j is a variable $n_j^{f'_j}$. We say the substitution extension $n_j^{f'_j f_j}$ is *linear* if $f'_j f_j = \mathbf{LL}$, it is *affine* if $f'_j f_j = \mathbf{AA}$, it is *intuitionistic* if $f'_j f_j = \mathbf{II}$, and it is *linear-changing* if $f'_j f_j = \mathbf{IL}$, $f'_j f_j = \mathbf{IA}$, or $f'_j f_j = \mathbf{AL}$. Notice that the possibilities $\mathbf{LI}$, $\mathbf{AI}$, and $\mathbf{LA}$ cannot occur in well-typed substitutions since this would imply referencing a linear or affine assumption in an intuitionistic context or a linear assumption in an affine context. This property of a variable in a substitution is called its *mode*.

Definition 4.4.1. A substitution $\Gamma \vdash a_1^{f_1} \ldots a_p^{f_p} \, . \uparrow^n \, : \Gamma'$ is said to be a *pattern substitution* if all the terms a_j for $j \in \{1, \ldots, p\}$ are distinct de Bruijn indices and none of them are linear-changing extensions in the substitution. A pattern substitution is called a *weakening substitution* if the indices a_j form an increasing sequence.

Note that in a pattern substitution all de Bruijn indices are less than or equal to n since n is equal to the length of Γ.

This definition is motivated by the need for a pattern substitution to have a well-typed inverse (see section 4.4.3).

We define the *extension of pattern substitution s by spine S*, written as $S \, . \, s$:

$$\begin{aligned}
() \, . \, s &= s \\
(N; S) \, . \, s &= S \, . \, (N^{\mathbf{I}} \, . \, s) \\
(N \mathring{;} S) \, . \, s &= S \, . \, (N^{\mathbf{A}} \, . \, s) \\
(N \hat{;} S) \, . \, s &= S \, . \, (N^{\mathbf{L}} \, . \, s) \\
(\mathsf{fst}; S) \, . \, s &= S \, . \, s \\
(\mathsf{snd}; S) \, . \, s &= S \, . \, s
\end{aligned}$$

This is motivated by the fact that $S \, . \, s$ is the substitution that appears when the logic variable X in $X[s] \cdot S$ is lowered (lowering is explained below in 4.5.2).

Notice that we get the equality $(S \, . \, s) \circ t = S[t] \, . \, (s \circ t)$ by induction on S.

Definition 4.4.2. A term M is said to be a *pattern* or within the *pattern fragment* if all occurrences of logic variables $X[s] \cdot S$ satisfy the property that the substitution $S \, . \, s$ is a pattern substitution.

Recall example (4.1) from the introduction. In our system, the equation is written as $F[\uparrow^1] \cdot (1^{\mathbf{I}} \hat{;} ()) \doteq c \cdot (H_1[\uparrow^1] \cdot (1^{\mathbf{I}}; ()) \hat{;} H_2[\uparrow^1] \cdot (1^{\mathbf{I}}; ()) \hat{;} ())$. We observe that it is not a pattern since there is a linear-changing substitution extension on the left-hand side in $(1^{\mathbf{I}} \hat{;} ()) \, . \uparrow^1 = 1^{\mathbf{IL}} \, . \uparrow^1$.

4.4.2 Stability of the pattern fragment

Most of the theorems below pertaining to the stability of the pattern fragment are extensions of their intuitionistic counterparts, which can be found in [DHKP98].

Lemma 4.4.3. *If $\Gamma \vdash s : \Gamma'$ and $\Gamma' \vdash t : \Gamma''$ are pattern substitutions then the following four substitutions are also pattern substitutions:*

1. $\Gamma, A^{\mathbf{I}} \vdash 1^{\mathbf{II}} \,.\, (s \circ \uparrow) : \Gamma', A^{\mathbf{I}}$
2. $\Gamma, A^{\mathbf{A}} \vdash 1^{\mathbf{AA}} \,.\, (s \circ \uparrow) : \Gamma', A^{\mathbf{A}}$
3. $\Gamma, A^{\mathbf{L}} \vdash 1^{\mathbf{LL}} \,.\, (s \circ \uparrow) : \Gamma', A^{\mathbf{L}}$
4. $\Gamma \vdash t \circ s : \Gamma''$

Proof. Let $s = a_1^{f_1 f_1} \dots a_p^{f_p f_p} \,.\, \uparrow^n$ and $t = b_1^{g_1 g_1} \dots b_q^{g_q g_q} \,.\, \uparrow^m$. The substitutions in 1, 2, and 3 are clearly pattern substitutions since

$$1^{ff} \,.\, (s \circ \uparrow) = 1^{ff} \,.\, (a_1 + 1)^{f_1 f_1} \dots (a_p + 1)^{f_p f_p} \,.\, \uparrow^{n+1}.$$

For the substitution in 4 we first notice that $m = p$, since both m and p are equal to the length of Γ'.

$$t \circ s = b_1^{g_1}[s]^{g_1} \dots b_q^{g_q}[s]^{g_q} \,.\, \uparrow^n = a_{b_1}^{f_{b_1} g_1} \dots a_{b_q}^{f_{b_q} g_q} \,.\, \uparrow^n$$

Distinctness of the variables follows from distinctness in s and t, since two equal variables $a_{b_j} = a_{b_i}$ implies $b_j = b_i$, which in turn implies $j = i$.

The modes of the variables are also all either linear, affine, or intuitionistic since their types are represented in all of Γ, Γ', and Γ'' and there are no linear-changing variables in s or t. □

Lemma 4.4.4. *If t and $S \,.\, s$ are pattern substitutions then so is $S \,.\, (t \circ s)$.*

Proof. By induction on S and Lemma 4.4.3. We give only the case $S = N; S'$ as the rest are either similar or trivial. Now $1^{\mathbf{II}} \,.\, (t \circ \uparrow)$ and $S' \,.\, (N^{\mathbf{I}} \,.\, s)$ are pattern substitutions, and the induction hypothesis therefore gives us that $S' \,.\, ((1^{\mathbf{II}} \,.\, (t \circ \uparrow)) \circ (N^{\mathbf{I}} \,.\, s))$ is a pattern substitution, but now we are done since it is also equal to $S' \,.\, (N^{\mathbf{I}} \,.\, (t \circ s)) = (N; S') \,.\, (t \circ s)$. □

Lemma 4.4.5. *If M is a pattern and s is a pattern substitution then $M[s]$ is a pattern.*

Proof. By induction on M and Lemma 4.4.3. In the case $M = X[t] \cdot S$ we need to prove that the normal form of $S[s] \,.\, (t \circ s)$ is a pattern substitution. But this follows from Lemma 4.4.3 and the fact that $(S \,.\, t) \circ s = S[s] \,.\, (t \circ s)$. □

Lemma 4.4.6. *If M is a pattern and $S \,.\, s$ is a pattern substitution then $M[s] \cdot S$ is a pattern.*

Proof. The proof is by induction on S. If $M = n^f \cdot S'$ then $n^f[s] \cdot S'[s] \cdot S$ is a pattern by Lemma 4.4.5 on each term in $S'[s]$. The case $S = ()$ is also handled by Lemma 4.4.5.

If $M = X[t] \cdot S'$ then we need to show that $S \mathbin{.} S'[s] \mathbin{.} (t \circ s)$ is a pattern substitution. This follows from Lemma 4.4.4 and $S.((S'.t) \circ s) = S.S'[s].(t \circ s)$.

If $S = N; S'$ and $M = \lambda M'$ then

$$(\lambda M')[s] \cdot (N; S') = M'[1^{\mathbf{II}} \mathbin{.} (s \circ \uparrow)][N^{\mathbf{I}} \mathbin{.} \mathsf{id}] \cdot S' = M'[N^{\mathbf{I}} \mathbin{.} s] \cdot S'$$

which is a pattern by the induction hypothesis.

The two projection and pair cases are trivial and the two remaining cases are similar to the one above. □

Theorem 4.4.7. *The pattern fragment is stable under logic variable instantiation. I.e. for any patterns M and N, $[X \leftarrow N]M$ is a pattern.*

Proof. By induction on M and Lemma 4.4.6. □

Theorem 4.4.8. *The pattern fragment is stable under inversion of substitutions.*

1. *If s is a pattern substitution and $M[s]$ is a pattern then M is a pattern.*
2. *If s is a pattern substitution and $t \circ s$ is a pattern substitution then t is a pattern substitution.*

Proof. The proof is by mutual induction on M and t using Lemma 4.4.3. The interesting case is the substitution case. Let $t = b_1^{g_1} \dots b_q^{g_q} \mathbin{.} \uparrow^p$ and $s = a_1^{f_1} \dots a_p^{f_p} \mathbin{.} \uparrow^n$.

$$t \circ s = b_1[s]^{g_1} \dots b_q[s]^{g_q} \mathbin{.} \uparrow^n$$

The term $b_j[s]$ is a variable and b_j is therefore also a variable. Since $b_1[s], \dots, b_q[s]$ are distinct variables then $b_1, \dots, b_q$ are also distinct variables. Additionally, there can be no linear-changing variables in t, since $b_j = i^f$ implies $a_i^{f_i} = k^{ff}$ and thus $g_j \neq f$ implies a linear-changing variable in $t \circ s$ in the same position. □

4.4.3 Inversion of substitutions

Next, we define the inverse of a pattern substitution. The name is justified by Theorem 4.4.11 below.

Definition 4.4.9. Let $s = a_1^{f_1} \dots a_p^{f_p} \mathbin{.} \uparrow^n$ be a pattern substitution. We define its inverse to be $s^{-1} = e_1^{g_1} \dots e_n^{g_n} \mathbin{.} \uparrow^p$ where $e_j^{g_j} = i^{f_i f_i}$ when $a_i = j^{f_i}$ and e_j is undefined otherwise. The undefined extensions $e_j^{g_j}$ are flagged intuitionistic, affine, or linear in s^{-1} depending on the jth assumption in the codomain of s.

Notice first that the definition is well defined: the a_is are distinct and less than or equal to n. For the undefined e_j one can think of an arbitrary term of the right type, e.g. a freshly created logic variable.

Second, we see that inversion is stable under η, thereby allowing us to compute inverses based on η-short short-hands for the actual canonical forms. This is the case, since $s = a_1^{f_1} \dots a_p^{f_p} \mathbin{.} \uparrow^n$ and $s' = a_1^{f_1} \dots a_p^{f_p} \mathbin{.} (n+1)^{ff} \mathbin{.} \uparrow^{n+1}$ implies $s^{-1} = e_1^{g_1} \dots e_n^{g_n} \mathbin{.} \uparrow^p$ and $s'^{-1} = e_1^{g_1} \dots e_n^{g_n} \mathbin{.} (p+1)^{ff} \mathbin{.} \uparrow^{p+1}$, which are equal.

In the following we will refer to affine weakening on contexts $\Gamma \succ_{\mathsf{aff}} \Gamma'$, which was defined in chapter 3 as

$$\Gamma \succ_{\mathsf{aff}} \Gamma' \;\equiv\; \exists\Gamma''.\ \Gamma = \Gamma'' \bowtie \Gamma' \ \wedge\ \mathsf{nolin}(\Gamma'')$$

Recall that affine weakening is reflexive and transitive, as it merely amounts to changing some number of $\mathbf{A}$s into $\mathbf{U_A}$s.

Lemma 4.4.10. *For a pattern substitution $\Gamma_2 \vdash s : \Gamma'$ there exists a Γ_1 with $\Gamma_2 \succ_{\mathsf{aff}} \Gamma_1$ such that $\Gamma_1 \vdash s : \Gamma'$ and the inverse is well-typed with $\Gamma' \vdash s^{-1} : \Gamma_1$.*

Proof. Let $s = a_1^{f_1} \ldots a_p^{f_p} . {\uparrow}^n$. Then $\Gamma_2 = \cdot, B_n^{l_n^2}, \ldots, B_1^{l_1^2}$ and $\Gamma' = \cdot, A_p^{l'_p}, \ldots, A_1^{l'_1}$. Intuitively we are going to take Γ_1 to be the smallest possible such that s is still well-typed, i.e. we are going to make all the affine assumptions that are not used in s unavailable. More formally we are going to set $\Gamma_1 = \cdot, B_n^{l_n^1}, \ldots, B_1^{l_1^1}$ where $l_j^1 = l_j^2$ when $l_j^2 \in \{\mathbf{I}, \mathbf{L}, \mathbf{U_L}, \mathbf{U_A}\}$. When $l_j^2 = \mathbf{A}$ the l_j^1 will be defined below.

Consider each extension $a_i^{f_i} = j^{f_i f_i}$ in s. Note that we have $A_i = B_j$. If $l'_i = f$ where f is either $\mathbf{I}$ or $\mathbf{L}$ then we have $f_i = f$ and $l_j^2 = l_j^1 = f$. In the case where $l'_i = \mathbf{U_L}$ then $f_i = \mathbf{L}$ and $l_j^2 = l_j^1$ are either equal to $\mathbf{U_L}$ or $\mathbf{L}$, but since all the variables in s are distinct it has to be $\mathbf{U_L}$. If $l'_i = \mathbf{A}$ then $f_i = \mathbf{A}$ and $l_j^2 = \mathbf{A}$, and in this case we set $l_j^1 = \mathbf{A}$. Finally, if $l'_j = \mathbf{U_A}$ then $f_i = \mathbf{A}$ and l_j^2 is either $\mathbf{U_A}$ or $\mathbf{A}$. If $l_j^2 = \mathbf{U_A}$ then l_j^1 is also equal to $\mathbf{U_A}$, and if $l_j^2 = \mathbf{A}$ then we can set $l_j^1 = \mathbf{U_A}$ since j does not occur anywhere else in s. This means that for all defined extensions $e_j = i^{f_i}$ in s^{-1} we have $l'_i = l_j^1$.

The remaining $B_j^{l_j^2}$s for which there are no $a_i = j^{f_i}$ are all shifted away by the ${\uparrow}^n$ part of s. Therefore none of them can be linear, and if any of them are affine, i.e. have $l_j^2 = \mathbf{A}$, we set $l_j^1 = \mathbf{U_A}$. This means that all the undefined extensions in s^{-1} correspond to intuitionistic, used linear, or used affine assumptions in Γ_1, and we see that s^{-1} indeed is well-typed with $\Gamma' \vdash s^{-1} : \Gamma_1$. □

Theorem 4.4.11. *Given a pattern substitution $\Gamma \vdash s : \Gamma'$, we have $\Gamma' \vdash s \circ s^{-1} : \Gamma'$ and $s \circ s^{-1} = \mathsf{id}$.*

Proof. Let $s = a_1^{f_1} \ldots a_p^{f_p} . {\uparrow}^n$. Since $a_i = j^{f_i}$ then the jth extension in s^{-1} is equal to i^{f_i}, and thus $a_i[s^{-1}] = i^{f_i}$ for all i. □

We have the usual definition of occurrence, rigid occurrence, and flexible occurrence written as $\in$, $\in_{\mathsf{rig}}$, and $\in_{\mathsf{flex}}$ respectively. These relations are only defined for canonical forms in which all logic variables are of base type (lowering will achieve this). Occurrence is defined as $\in \;=\; \in_{\mathsf{rig}} \cup \in_{\mathsf{flex}}$. Rigid occurrence and flexible occurrence are defined as follows, where we write $\in_o$ for either rigid or flexible occurrence.

$$\frac{}{n \in_{\mathsf{rig}} n^f} \qquad \frac{n \in_{\mathsf{rig}} s}{n \in_{\mathsf{flex}} X[s]} \qquad \frac{a_i = n^f}{n \in_{\mathsf{rig}} a_1^{f_1} \ldots a_p^{f_p} . {\uparrow}^m}$$

$$\frac{n \in_o M_i}{n \in_o \langle M_1, M_2 \rangle} \qquad \frac{n+1 \in_o M}{n \in_o \widehat{\lambda} M} \qquad \frac{n+1 \in_o M}{n \in_o \mathring{\lambda} M} \qquad \frac{n+1 \in_o M}{n \in_o \lambda M}$$

$$\frac{n \in_o M}{n \in_o \mathsf{fst}\ M} \qquad \frac{n \in_o M}{n \in_o \mathsf{snd}\ M} \qquad \frac{n \in_o M_i}{n \in_o M_1 \mathbin{\hat{}} M_2} \qquad \frac{n \in_o M_i}{n \in_o M_1 \mathbin{@} M_2} \qquad \frac{n \in_o M_i}{n \in_o M_1\ M_2}$$

If $n \in_{\mathsf{flex}} M$ then the definition implies that there exists some logic variable $X[a_1^{f_1} \dots a_p^{f_p} . \uparrow^m]$ in M beneath k lambdas such that $(n+k)^{f_i} = a_i$. In this case we say that n occurs in the ith argument of X.

Lemma 4.4.12. *Linearity implies occurrence.*

1. *Let $\Gamma \vdash s : \Gamma'$ be a pattern substitution and the nth assumption in Γ be linear. Then n occurs in s.*
2. *Let $\Gamma \vdash M : A$ be a pattern and let the nth assumption in Γ be linear. Then n occurs in M.*

Proof. If $s = a_1^{f_1 f_1} \dots a_p^{f_p f_p} . \uparrow^m$ then we must have $n = a_i$ for some a_i since a linear assumption cannot be shifted away. The second case is by induction on M. □

Definition 4.4.13. Given the typing of a substitution $\Gamma \vdash s : \Gamma'$ we will call it *strong* if there exists no $\Gamma'' \neq \Gamma'$ such that $\Gamma'' \succ_{\mathsf{aff}} \Gamma'$ and $\Gamma \vdash s : \Gamma''$.

For a pattern substitution $\cdot, B_n^{l_n}, \dots, B_1^{l_1} \vdash a_1^{f_1} \dots a_p^{f_p} . \uparrow^n : \cdot, A_p^{l'_p}, \dots, A_1^{l'_1}$ we see that it is strong if and only if for each affine variable $a_i = j^{\mathbf{A}}$ we have $l'_i = \mathbf{U_A}$ implies $l_j = \mathbf{U_A}$.

Consider the split of a strong pattern substitution $\Gamma \vdash s : \Gamma'$ over a context split $\Gamma' = \Gamma'_1 \bowtie \Gamma'_2$ into $\Gamma_1 \vdash s : \Gamma'_1$ and $\Gamma_2 \vdash s : \Gamma'_2$ with $\Gamma = \Gamma_1 \bowtie \Gamma_2$. For any used affine assumption in Γ'_1 the assumption is either affine or used affine in Γ' and Γ'_2. If it is used affine then the corresponding assumption is also used affine in Γ and thereby Γ_1. If it is affine then the corresponding assumption has to be affine in Γ_2 and is thereby used affine in Γ_1. This means that $\Gamma_1 \vdash s : \Gamma'_1$ is strong and by symmetry so is $\Gamma_2 \vdash s : \Gamma'_2$.

Theorem 4.4.14. *Let $\Gamma \vdash s : \Gamma'$ be a pattern substitution and $\Gamma \vdash M : A$ be a term in which all logic variables are of base type.*

1. *If there exists a term $\Gamma' \vdash M' : A$ such that $M = M'[s]$ then every variable occurring in M also occurs in s.*
2. *If the typing $\Gamma \vdash s : \Gamma'$ is strong and every variable occurring in M also occurs in s then there exists a term $\Gamma' \vdash M' : A$ such that $M = M'[s]$.*

Proof. Part 1 is by induction on the normal form of M'.

Part 2 is by induction on M where we as remarked above note that context splits preserve a strong typing of s. It is also easy to see that a strong typing of s implies a strong typing of $1^{ff} . (s \circ \uparrow)$ when going beneath a lambda-binder.

For the base case $M = n^f$ we get that $n \in s$ implies that the nth assumption in Γ corresponds to an assumption, say the mth, in Γ'. Now, we can take $M' = m^f$, and since s is strong, availability of the nth assumption in Γ implies availability of the mth assumption in Γ' and thus that M' is well-typed. The base case $M = X[t]$ is similar, when noting that the shift at the end of s is equal to the shift at the end of t, since they are both equal to the length of Γ. □

Theorem 4.4.14 states that occurrence is a conservative approximation of the set of variables occurring in any instantiation of a term, i.e. if $n \in [X \leftarrow N]M$ then $n \in M$. The opposite is not necessarily true.

4.5 Pattern unification

4.5.1 Unification problem

A unification problem P is a conjunction of unification equations, and a solution to a unification problem is an instantiation of the logic variables such that all equations are satisfied. Such a collection of logic variable instantiations will be written as θ and we say that θ solves P. In this section we describe an algorithm that returns "no" if no such solution exists or a most general unifier otherwise, i.e. a solution that all other solutions are refinements of.

More formally, we write $\Gamma \vdash M_1 \doteq M_2 : A$ for a unification equation or simply $M_1 \doteq M_2$ with the implicit understanding that both terms have the same type in the same context. Unification equations are symmetric and we will implicitly switch from $M_1 \doteq M_2$ to $M_2 \doteq M_1$ when needed. Unification problems are given by the following grammar, where $\mathbb{T}$ is the solved unification problem and $\mathbb{F}$ is the unification problem with no solutions.

$$P ::= \mathbb{T} \mid \mathbb{F} \mid P \wedge (\Gamma \vdash M_1 \doteq M_2 : A)$$

For convenience we generalize unification equations to spines and write $S_1 \doteq S_2$ as a short-hand for the argumentwise conjunction of unification equations (see Decomposition in section 4.5.2).

4.5.2 Unification algorithm

The unification algorithm consists of a set of transformation rules of the form $P \mapsto P'$. We will see that the repeated application of these rules to any pattern unification problem will eventually terminate resulting in either $\mathbb{F}$, which indicates that the original problem has no solution, or $\mathbb{T}$, which indicates that all equations have been solved and that a most general unifier has been found. In this case the most general unifier is a mapping from logic variables to their instantiations as computed during the execution of the algorithm.[1]

The unification algorithm is given in Figure 4.4 and each rule is explained in detail below. For convenience we write the decomposition of a term M into one of its subterms N and the surrounding term with a hole $M'(\!|\cdot|\!)$ as $M = M'(\!|N|\!)$.

Decomposition

Consider a unification equation $\Gamma \vdash M_1 \doteq M_2 : A$ and assume that A is not a base type.

If $A = B \multimap C$ then we must have $M_1 = \widehat{\lambda} M_1'$ and $M_2 = \widehat{\lambda} M_2'$. In this case M_1 is equal to M_2 under some θ if and only if M_1' is equal to M_2' under θ and we therefore apply **dec-lam-l**. The other non-base type cases for A are similar and give rise to **dec-lam-a**, **dec-lam-i**, and **dec-pair**.

If A is a base type then $M_1 = H_1 \cdot S_1$ and $M_2 = H_2 \cdot S_2$ where H_1 and H_2 are either variables or logic variables. The case of logic variables is handled below. We therefore have $n^f \cdot S_1 \doteq m^{f'} \cdot S_2$. If $n \neq m$ then no θ can make the two

[1] The instantiated logic variables could be represented explicitly in the unification problem, but we have chosen to represent them implicitly in order to get a cleaner presentation of the algorithm.

dec-lam-l	$P \wedge \widehat{\lambda} M_1 \doteq \widehat{\lambda} M_2$	$\mapsto$	$P \wedge M_1 \doteq M_2$
dec-lam-a	$P \wedge \mathring{\lambda} M_1 \doteq \mathring{\lambda} M_2$	$\mapsto$	$P \wedge M_1 \doteq M_2$
dec-lam-i	$P \wedge \lambda M_1 \doteq \lambda M_2$	$\mapsto$	$P \wedge M_1 \doteq M_2$
dec-pair	$P \wedge \langle M_1, N_1 \rangle \doteq \langle M_2, N_2 \rangle$	$\mapsto$	$P \wedge M_1 \doteq M_2 \wedge N_1 \doteq N_2$
dec-atomic-eq	$P \wedge n^f \cdot S_1 \doteq n^f \cdot S_2$	$\mapsto$	$P \wedge S_1 \doteq S_2$
dec-atomic-neq	$P \wedge n^f \cdot S_1 \doteq m^{f'} \cdot S_2$	$\mapsto$	$\mathbb{F}$
	if $n \neq m$		
lower-lolli	P	$\mapsto$	$[X \leftarrow \widehat{\lambda} Y[\mathsf{id}]]P$
	if $A_X = A \multimap B$ and Y is a fresh logic variable with $A_Y = B$ and $\Gamma_Y = \Gamma_X, A^{\mathbf{L}}$		
lower-affarr	P	$\mapsto$	$[X \leftarrow \mathring{\lambda} Y[\mathsf{id}]]P$
	if $A_X = A \mathrel{-\!\!@} B$ and Y is a fresh logic variable with $A_Y = B$ and $\Gamma_Y = \Gamma_X, A^{\mathbf{A}}$		
lower-arr	P	$\mapsto$	$[X \leftarrow \lambda Y[\mathsf{id}]]P$
	if $A_X = A \rightarrow B$ and Y is a fresh logic variable with $A_Y = B$ and $\Gamma_Y = \Gamma_X, A^{\mathbf{I}}$		
lower-and	P	$\mapsto$	$[X \leftarrow \langle Y[\mathsf{id}], Z[\mathsf{id}] \rangle]P$
	if $A_X = A \mathbin{\&} B$ and Y and Z are fresh logic variables with $A_Y = A$, $A_Z = B$, $\Gamma_Y = \Gamma_X$, and $\Gamma_Z = \Gamma_X$		
occurs-check	$P \wedge X[s] \doteq n^f \cdot S(\!\vert X[t] \vert\!)$	$\mapsto$	$\mathbb{F}$
pruning-fail	$P \wedge X[s] \doteq M$	$\mapsto$	$\mathbb{F}$
	if $n \notin s$ and $n \in_{\mathsf{rig}} M$		
pruning	$P \wedge X[s] \doteq M$	$\mapsto$	$[Y \leftarrow Z[w]](P \wedge X[s] \doteq M)$
	if $n \notin s$, n occurs flexibly in M in the ith argument of the logic variable Y, $w = \mathsf{weaken}(\Gamma_Y; i)$, and Z is a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \Gamma_Y \div i$		
ctx-pruning	P	$\mapsto$	$[X \leftarrow Y[w]]P$
	if $\Gamma_X = \cdot, A_p^{l_p}, \ldots, A_1^{l_1}$ with $l_n \in \{\mathbf{U_A}, \mathbf{U_L}\}$, $w = \mathsf{weaken}(\Gamma_X; n)$, and Y is a fresh logic variable with $A_Y = A_X$ and $\Gamma_Y = \Gamma_X \div n$		
instantiation	$P \wedge X[s] \doteq M$	$\mapsto$	$[X \leftarrow M[s^{-1}]]P$
	if X does not occur in M, Γ_X contains no used affine assumptions, and $n \in M$ implies $n \in s$		
intersection-eq	$P \wedge X[s] \doteq X[s]$	$\mapsto$	P
intersection-fail	$P \wedge X[s] \doteq X[t]$	$\mapsto$	$\mathbb{F}$
	if $s \neq t$ and $s \cap t$ does not exist		
intersection	$P \wedge X[s] \doteq X[t]$	$\mapsto$	$[X \leftarrow Y[s \cap t]]P$
	if $s \neq t$, $s \cap t$ exists, and Y is a fresh logic variable with $A_Y = A_X$ and Γ_Y equal to the domain of the weakening substitution $s \cap t$		

Figure 4.4: Pattern unification rules

equal and we can therefore apply **dec-atomic-neq**. If $n = m$ then the spines must unify and we apply **dec-atomic-eq** where $P \wedge S_1 \doteq S_2$ is defined as:

$$\begin{array}{ll}
P \wedge () \doteq () & = P \\
P \wedge (M_1; S_1) \doteq (M_2; S_2) & = P \wedge M_1 \doteq M_2 \wedge S_1 \doteq S_2 \\
P \wedge (M_1 \mathring{;} S_1) \doteq (M_2 \mathring{;} S_2) & = P \wedge M_1 \doteq M_2 \wedge S_1 \doteq S_2 \\
P \wedge (M_1 \,\widehat{;}\, S_1) \doteq (M_2 \,\widehat{;}\, S_2) & = P \wedge M_1 \doteq M_2 \wedge S_1 \doteq S_2 \\
P \wedge (\mathsf{fst}; S_1) \doteq (\mathsf{fst}; S_2) & = P \wedge S_1 \doteq S_2 \\
P \wedge (\mathsf{snd}; S_1) \doteq (\mathsf{snd}; S_2) & = P \wedge S_1 \doteq S_2 \\
P \wedge (\mathsf{fst}; S_1) \doteq (\mathsf{snd}; S_2) & = \mathbb{F} \\
P \wedge (\mathsf{snd}; S_1) \doteq (\mathsf{fst}; S_2) & = \mathbb{F}
\end{array}$$

No other cases can occur because $n = m$ trivially imply that they have the same type.

Lowering

When a logic variable occurs in a unification problem in the form $X[s] \cdot S$ with a non-empty spine, we know that A_X cannot be a base type. And since canonical forms of non-base type have unique head constructors, we can safely instantiate X to that particular constructor. This is accomplished by the rules **lower-***. Therefore we can assume that all logic variables are of base type.

Occurs check

Consider a unification equation of the form $X[s] \doteq M$. If X also occurs in the right-hand side then either $M = n^f \cdot S(\!| X[t] |\!)$ or $M = X[t]$. The latter case is handled below in the Intersection section. In the former case we have the equation $X[s] \doteq n^f \cdot S(\!| X[t] |\!)$. Since a pattern substitution t applied to any term can never alter the shape of the term but only rename variables, this equation has no solutions, and we can apply **occurs-check**.

Pruning

When we have $X[s] \doteq M$ then Theorem 4.4.14 tells us that under some θ solving the equation, variables that do not occur in s cannot occur in M. Assume that $n \notin s$ and $n \in M$. If $n \in_{\mathsf{rig}} M$ then no instantiation of logic variables can get rid of the occurrence and we apply **pruning-fail**. If on the other hand $n \in_{\mathsf{flex}} M$ then the occurrence is in the ith argument of some logic variable Y. This means, however, that no instantiation of Y in a solution can contain i. By Lemma 4.4.12 we know that n cannot refer to a linear assumption in the context in which $X[s]$ and M are typed and therefore the ith assumption in Γ_Y cannot be linear.[2] Let w be the weakening substitution $\mathsf{weaken}(\Gamma_Y; i)$ where weaken is defined as:

$$\begin{array}{lcll}
\mathsf{weaken}(\Gamma, A^l; 1) & = & \uparrow & \text{if } l \neq \mathbf{L} \\
\mathsf{weaken}(\Gamma, A^{\mathbf{I}}; i+1) & = & 1^{\mathbf{II}} \,.\, \mathsf{weaken}(\Gamma; i) \circ \uparrow & \\
\mathsf{weaken}(\Gamma, A^l; i+1) & = & 1^{\mathbf{AA}} \,.\, \mathsf{weaken}(\Gamma; i) \circ \uparrow & \text{if } l \in \{\mathbf{A}, \mathbf{U_A}\} \\
\mathsf{weaken}(\Gamma, A^l; i+1) & = & 1^{\mathbf{LL}} \,.\, \mathsf{weaken}(\Gamma; i) \circ \uparrow & \text{if } l \in \{\mathbf{L}, \mathbf{U_L}\}
\end{array}$$

[2] Notice that this argument relies on the fact that Y is under a pattern substitution and thus has no linear-changing extensions.

Define $\Gamma \div i$ to be the context Γ with the ith assumption removed. We see that $\Gamma \vdash \mathsf{weaken}(\Gamma; i) : \Gamma \div i$. Furthermore, this is a strong typing. Since the ith assumption in Γ_Y is not linear then $w = \mathsf{weaken}(\Gamma_Y; i)$ does indeed exist.

Theorem 4.4.14 tells us that Y has to be instantiated to something on the form $M'[w]$ and we can therefore apply **pruning**.

Context pruning

If a logic variable X is declared in context $\Gamma_X = \cdot, A_p^{l_p}, \ldots, A_1^{l_1}$ with $l_n \in \{\mathbf{U_A}, \mathbf{U_L}\}$, we know that n cannot occur in a well-typed instantiation of X. Therefore, by Theorem 4.4.14, X has to be instantiated to something on the form $M[\mathsf{weaken}(\Gamma_X; n)]$ and we can therefore apply **ctx-pruning**.

Note that pruning the context of X in this way in the case of $X[s] \doteq M$ may allow further pruning in M. Additionally, repeated applications of this step will ensure that no used affine assumptions occur in the context of logic variables. Therefore all typings of the associated substitutions are strong.

Instantiation

Consider the unification equation $X[s] \doteq M$ where all used affine assumptions have been pruned from Γ_X and the typing of s therefore is strong. If all $n \in M$ also occur in s then Theorem 4.4.14 tells us that M is equal to $M'[s]$ for some M'. By Theorem 4.4.11 we know that M' is equal to $M[s^{-1}]$ and we can therefore instantiate X by the rule **instantiation** provided that X does not occur in M.

Intersection

The final case is when we have $X[s] \doteq X[t]$. If $s = t$ then the equation will be trivially satisfied no matter what term X might be instantiated to, so we can simply remove the equation by the rule **intersection-eq**.

Consider an instantiation of X to some M. If for all $n \in M$ we have $n[s] = n[t]$ then the equation is clearly satisfied. If on the other hand there is some $n \in M$ such that $n[s] \neq n[t]$ then the two sides of the equation will not be equal. Therefore any variable n for which $n[s] \neq n[t]$ cannot occur in an instantiation of X. If such an n is linear then Lemma 4.4.12 tells us that n has to occur in all instantiations and we can conclude that there is no solution and apply **intersection-fail**. Otherwise, any instantiation of X has to be on the form $M'[s \cap t]$ for some M' where $s \cap t$ is defined as the following weakening substitution:

$$\begin{array}{lcll} M^f \,.\, s \cap M^f \,.\, t & = & 1^{ff} \,.\, (s \cap t) \circ \uparrow & \\ n^{ff} \,.\, s \cap m^{ff} \,.\, t & = & (s \cap t) \circ \uparrow & \text{if } n \neq m \text{ and } f \in \{\mathbf{I}, \mathbf{A}\} \\ \uparrow^n \cap \uparrow^n & = & \mathsf{id} & \end{array}$$

Note that $s \cap t$ exists exactly when $n[s] = n[t]$ for all linear n. The domain of $s \cap t$ is seen to be Γ_X with those assumptions removed for which $n[s] \neq n[t]$.

This step is summarized by the rule **intersection**.

4.5.3 Correctness

Correctness of the unification algorithm has three parts: preservation, progress, and termination.

Theorem 4.5.1. *The unification algorithm solves all pattern unification problems correctly.*

1. *If $P \mapsto P'$ then the set of solutions to P is equal to the set of solutions to P'.*

2. *If P has unsolved equations (i.e. P is not equal to $\mathbb{F}$ or $\mathbb{T}$) then there exists a P' such that $P \mapsto P'$.*

3. *The unification algorithm terminates.*

Proof. The discussion above in section 4.5.2 proves preservation of solutions (1) and progress (2). For termination (3) we will consider the lexicographic ordering of

1. The total size of all types of all logic variables occurring in the unification problem.

2. The total size of all contexts of the logic variables occurring in the unification problem.

3. The total size of all terms in the unification problem.

We see that the decomposition rules **dec-*** decrease (3) while keeping (1) and (2) constant. The lowering rules **lower-*** and **instantiation** decrease (1). The **intersection-eq** rule decreases (3) while keeping (1) and (2) constant. The **pruning**, **ctx-pruning**, and **intersection** rules decrease (2) while keeping (1) constant. □

4.6 Linearity pruning

Within the pattern fragment we know that most general unifiers exist and we have a decidable algorithm for finding them. For practical applications, however, it is often necessary to relax the pattern restriction and accept that the algorithm sometimes returns left-over unification problems. Reed [Ree09], for example, describes the dynamic intuitionistic pattern fragment that postpones any unification equation as constraints that cannot be solved immediately.

In this section we will relax the restriction of pattern substitutions from Definition 4.4.1 to *linear-changing pattern substitutions* permitting linear-changing extensions, greatly expanding the applicability of our unification algorithm. If a unification equation involving linear-changing pattern substitutions cannot be resolved with a most general unifier, it is simply postponed as a constraint. Instead of just returning $\mathbb{T}$ or $\mathbb{F}$, the unification algorithm using linearity pruning may thus fail with leftover constraints.

4.6.1 Revisiting variable occurrence

In order to handle linear-changing extensions in substitutions we first need to revisit the notion of variable occurrence that was defined in section 4.4.3. So far, occurrences have been divided into two categories; rigid and flexible. We will need to make further distinctions into a total of 12 categories.

pruning-fail $P \wedge X[s] \doteq M \quad \mapsto \quad \mathbb{F}$
if $n \notin s$ and either $n \in_{\mathsf{rig}} M$ or $n \in_{\mathsf{flex},\mathbf{L}} M$

pruning $P \wedge X[s] \doteq M \quad \mapsto \quad [Y \leftarrow Z[w]](P \wedge X[s] \doteq M)$
if $n \notin s$, n occurs flexibly in M in the ith argument of the logic variable Y, $w = \mathsf{weaken}(\Gamma_Y; i)$ exists, and Z is a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \Gamma_Y \div i$

Figure 4.5: Modified pruning rules

We say that an occurrence is in an *intuitionistic position* in a term if the term can be written as $M\llparenthesis n^f \cdot S \cdot (N; S')\rrparenthesis$ such that the occurrence is within N. If an occurrence is not in an intuitionistic position and the term can be written as $M\llparenthesis n^f \cdot S \cdot (N \mathring{;} S')\rrparenthesis$ such that the occurrence is within N we say that it is in an *affine position.* If an occurrence is neither in an intuitionistic position nor in an affine position we say that it is in a *linear position.* This means that intuitionistic positions are precisely those positions in a term in which top-level affine and linear assumptions are not available. Similarly, affine positions are those in which top-level affine assumptions are available but the linear are not. Finally, linear positions are those where all top-level assumptions are available.

When an n occurs flexibly in a term M, i.e. it occurs in the ith argument of some logic variable X, there are five possibilities for the ith assumption in Γ_X; it can be intuitionistic, affine, used affine, linear, or used linear. We say that n occurs in an *intuitionistic argument* if the ith assumption in Γ_X is intuitionistic, we say that it occurs in an *affine argument* if the ith assumption in Γ_X is affine, and we say that it occurs in a *linear argument* if the ith assumption in Γ_X is linear. We will write this as $n \in_{\mathsf{flex},\mathbf{I}} M$, $n \in_{\mathsf{flex},\mathbf{A}} M$, and $n \in_{\mathsf{flex},\mathbf{L}} M$, respectively. Occurrences where the ith assumption in Γ_X is either used affine or used linear are not relevant, since context pruning will have removed them (see rule **ctx-pruning** in Figure 4.4).

This gives a total of 12 categories of occurrence, since any occurrence is either in an intuitionistic, affine, or linear position and it is either a rigid occurrence or a flexible occurrence in an intuitionistic, affine, or linear argument.

If we at any time are forced to prune a variable occurring in a linear argument we can simply fail, since the reason for pruning implies that the variable cannot occur in the given place but the linear typing tells us that it will. Consider the case $X[s] \doteq M$ with $n \notin s$ and $n \in M$. Since we have widened the fragment we are considering to include linear-changing pattern substitutions it is now possible that $n \in_{\mathsf{flex},\mathbf{L}} M$. This was previously impossible since if every substitution is a pattern then $n \in_{\mathsf{flex},\mathbf{L}} M$ implies that n is linear which in turn implies $n \in s$. The **pruning** and **pruning-fail** rules therefore has to be modified slightly in this case as shown in Figure 4.5.

4.6.2 Linear-changing pattern substitutions

Definition 4.6.1. A linear-changing pattern substitution s is called a *linear-changing identity* substitution if it is on the form:

$$1^{f_1 f_1'} . 2^{f_2 f_2'} \dots n^{f_n f_n'} . \uparrow^n$$

or equivalently that it is η-equivalent to id except for some number of linear-changing extensions.

Theorem 4.6.2. *Linear-changing identity substitutions are injective.*

Given M, M', and a linear-changing identity substitution s, then $M[s] = M'[s]$ implies $M = M'$.

Proof. The substitution s simply changes the linearity flags in M and M' from $\mathbf{L}$ to $\mathbf{A}$ or $\mathbf{I}$ or from $\mathbf{A}$ to $\mathbf{I}$ on those variables that are linear-changing in s and it is therefore trivially injective. □

Theorem 4.6.3. *A linear-changing pattern substitution can be decomposed into a pattern substitution and a linear-changing identity substitution.*

If s is a linear-changing pattern substitution then there exists a pattern substitution s' and a linear-changing identity substitution t such that $s = s' \circ t$.

Proof. Take s' to be s with all linear-changing extensions $\mathbf{AL}$ and $\mathbf{IL}$ changed to linear extensions and all linear-changing extensions $\mathbf{IA}$ changed to affine extensions and t to be a linear-changing identity substitution with the corresponding linear-changing extensions. □

Theorem 4.6.4. *Let s be a linear-changing identity substitution with exactly one linear-changing extension $n^{ff'}$ and M be some term.*

1. *If the linear-changing extension is $ff' = \mathbf{IL}$ then there exists an M' such that $M = M'[s]$ if and only if the following five properties hold:*

   (a) *n occurs in M.*

   (b) *There are no occurrences of n in intuitionistic or affine positions in M.*

   (c) *For all subterms $\langle M_1, M_2\rangle$ of M under k lambdas $n+k$ occurs in M_1 if and only if it occurs in M_2.*

   (d) *For all subterms $M_1 \,\hat{}\, M_2$ of M under k lambdas $n+k$ occurs in at most one of M_1 and M_2.*

   (e) *All flexible occurrences of n in M are in linear arguments.*

2. *If the linear-changing extension is $ff' = \mathbf{IA}$ then there exists an M' such that $M = M'[s]$ if and only if the following three properties hold:*

   (a) *There are no occurrences of n in intuitionistic positions in M.*

   (b) *For all subterms $M_1 \,\hat{}\, M_2$ and $M_1 \mathbin{@} M_2$ of M under k lambdas $n+k$ occurs in at most one of M_1 and M_2.*

   (c) *All flexible occurrences of n in M are in linear or affine arguments.*

3. *If the linear-changing extension is $ff' = \mathbf{AL}$ then there exists an M' such that $M = M'[s]$ if and only if the following four properties hold:*

   (a) *n occurs in M.*

   (b) *There are no occurrences of n in affine positions in M.*

   (c) *For all subterms $\langle M_1, M_2\rangle$ of M under k lambdas $n+k$ occurs in M_1 if and only if it occurs in M_2.*

(d) All flexible occurrences of n in M are in linear arguments.

Proof. By induction on M noting that each of the three sets of properties are precisely the occurrence requirements for, respectively, linear variables, affine variables, and linear variables known to adhere to the affine occurrence requirements. □

Theorem 4.6.4 tells us when there exists an M' such that $M = M'[s]$ for a linear-changing identity substitution s with a single linear-changing extension. As a corollary we get the conditions when s is a general linear-changing identity substitution. The existence of M' is equivalent to the conjunction of the requirements for each linear-changing extension, since we can decompose any linear-changing identity substitution s with k linear-changing extensions into $s = s_1 \circ s_2 \circ \cdots \circ s_k$ where each s_i is a linear-changing identity substitution with exactly one linear-changing extension.

4.6.3 Linearity pruning

Consider the following unification equation where s is a linear-changing pattern substitution:

$$\Gamma \vdash X[s] \doteq M : B$$

We cannot invert s directly but we can decompose it by Theorem 4.6.3 into a pattern substitution s' and a linear-changing identity substitution t changing the problem to:

$$\Gamma \vdash X[s'][t] \doteq M : B$$

In this case we perform a number of pruning steps on the right-hand side since in any solution the M must adhere to the requirements in Theorem 4.6.4. We will consider each linear-changing extension $n^{ff'}$ in t individually.

The entire algorithm is given in Figures 4.6 and 4.7 and each rule is explained below.

Since many of the rules rely on pruning, we extend our language of unification problems with the constraint $\mathsf{prune}(n; M)$ to simplify the presentation. This constraint states that n cannot occur in M in a solution. If this is already the case then the rule **prune-finish** removes it. If n occurs either rigidly or flexibly in a linear argument in M then no instantiation of logic variables can get rid of the occurrence, and therefore there are no solutions. The rule **prune-fail** covers this case. If there are flexible occurrences in either intuitionistic or affine arguments then we can safely prune them away with the rule **prune**.

Position-based pruning

The variable n cannot occur in any intuitionistic position. Furthermore, if $f' = \mathbf{L}$ then n also cannot occur in affine positions. These occurrences can therefore be pruned away with the rules **int-pos** and **aff-pos**.

Pruning at multiplicative context splits

We will now consider all linear applications $M_1 \,\hat{}\, M_2$ and all affine applications $M_1 \mathbin{@} M_2$ in the term M and compare occurrences in M_1 and M_2, as these positions are where the context is split multiplicatively.

int-pos $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s] \doteq M \wedge \mathsf{prune}(n+k; N)$
if $n^{ff'}$ is a linear-changing extension in s and n occurs in an intuitionistic position in M in the subterm N under k lambdas

aff-pos $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s] \doteq M \wedge \mathsf{prune}(n+k; N)$
if $n^{f\mathbf{L}}$ is a linear-changing extension in s and n occurs in an affine position in M in the subterm N under k lambdas

prune-fail $P \wedge \mathsf{prune}(n; M) \quad \mapsto \quad \mathbb{F}$
if $n \in_{\mathsf{rig}} M$ or $n \in_{\mathsf{flex},\mathbf{L}} M$

prune $P \wedge \mathsf{prune}(n; M) \quad \mapsto \quad [Y \leftarrow Z[w]](P \wedge \mathsf{prune}(n; M))$
if n occurs in the ith argument of the logic variable Y in M, the argument is either intuitionistic or affine, $w = \mathsf{weaken}(\Gamma_Y; i)$, and Z is a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \Gamma_Y \div i$

prune-finish $P \wedge \mathsf{prune}(n; M) \quad \mapsto \quad P$
if $n \notin M$

multiplicative $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s] \doteq M \wedge \mathsf{prune}(n+k; M_2)$
if $n^{\mathbf{I}f'}$ is a linear-changing extension in s, $n+k$ occurs either rigidly or flexibly in a linear argument in M_1, and $n+k$ occurs in M_2, where either $M_1 \mathbin{\hat{}} M_2$, $M_2 \mathbin{\hat{}} M_1$, $M_1 @ M_2$, or $M_2 @ M_1$ is a subterm of M beneath k lambdas

additive $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s] \doteq M \wedge \mathsf{prune}(n+k; M_2)$
if $n^{f\mathbf{L}}$ is a linear-changing extension in s, $n+k \notin M_1$, and $n+k \in M_2$, where $\langle M_1, M_2 \rangle$ or $\langle M_1, M_2 \rangle$ is a subterm of M beneath k lambdas

int-strengthen $P \wedge X[s] \doteq M \quad \mapsto \quad [Y \leftarrow Z[t]](P \wedge X[s] \doteq M)$
if $n^{\mathbf{I}f'}$ is a linear-changing extension in s, n occurs flexibly in M in the ith argument of the logic variable Y, the argument is intuitionistic, $t = \mathsf{linweaken}(i; \mathbf{IA})$, and Z is a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \mathsf{strengthen}(\Gamma_Y; i; \mathbf{IA})$

aff-strengthen $P \wedge X[s] \doteq M \quad \mapsto \quad [Y \leftarrow Z[t]](P \wedge X[s] \doteq M)$
if $n^{\mathbf{AL}}$ is a linear-changing extension in s, n occurs flexibly in M in the ith argument of the logic variable Y, the argument is affine, $t = \mathsf{linweaken}(i; \mathbf{AL})$, and Z is a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \mathsf{strengthen}(\Gamma_Y; i; \mathbf{AL})$

no-occur $P \wedge X[s] \doteq M \quad \mapsto \quad \mathbb{F}$
if $n^{f\mathbf{L}}$ is a linear-changing extension in s and $n \notin M$

Figure 4.6: Linearity pruning

int-aff-invert $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s'] \doteq M'$
if $n^{\mathbf{I}f'}$ is a linear-changing extension in s, there are no occurrences of n in intuitionistic positions in M, for all subterms $M_1 \mathbin{^\frown} M_2$ and $M_1 \mathbin{@} M_2$ of M under k lambdas $n+k$ occurs in at most one of M_1 and M_2, and all flexible occurrences of n in M are in linear or affine arguments; s' and M' are given by $s = s' \circ t$ and $M = M'[t]$ where $t = \mathsf{linweaken}(n; \mathbf{IA})$

aff-lin-invert $P \wedge X[s] \doteq M \quad \mapsto \quad P \wedge X[s'] \doteq M'$
if $n^{\mathbf{AL}}$ is a linear-changing extension in s, n occurs in M, there are no occurrences of n in affine positions in M, for all subterms $\langle M_1, M_2 \rangle$ of M under k lambdas $n+k$ occurs in M_1 if and only if it occurs in M_2, and all flexible occurrences of n in M are in linear arguments; s' and M' are given by $s = s' \circ t$ and $M = M'[t]$ where $t = \mathsf{linweaken}(n; \mathbf{AL})$

Figure 4.7: Linearity pruning

For any multiplicative context split the variable should only occur in one of the branches by Theorem 4.6.4. A multiplicative split with rigid or linear argument occurrences in one of the branches therefore allows us to prune any occurrences in the other branch with the rule **multiplicative**, and if this is impossible due to rigid or linear argument occurrences in both branches, we conclude that there is no solution by following up with **prune-fail**.

We can restrict the **multiplicative** rule to the case where $f = \mathbf{I}$, since $ff' = \mathbf{AL}$ implies that n already occurs in at most one of the branches at each multiplicative split.

Pruning at additive context splits

Similarly, we consider all pairs $\langle M_1, M_2 \rangle$ in the term M, i.e. the places where the context is split additively.

If $f' = \mathbf{L}$ then the variable n must occur in either both branches of the additive split or in none of them. An additive split without occurrences in one of the branches therefore allows us to prune any occurrences in the other branch, which is done in the **additive** rule.

Strengthening intuitionistic variables

Consider the case when $f = \mathbf{I}$, i.e. n is intuitionistic, and consider some flexible occurrence of n in an intuitionistic argument, say the ith, of some logic variable Y in M. If $f' = \mathbf{L}$ then we do not necessarily know whether this particular occurrence should be pruned away or strengthened to a linear occurrence, but in either case, and also if $f' = \mathbf{A}$, we can safely strengthen the ith assumption of Y from intuitionistic to affine. Let $t = \mathsf{linweaken}(\Gamma_Y; i; \mathbf{IA})$ and Z be a fresh logic variable with $A_Z = A_Y$ and $\Gamma_Z = \mathsf{strengthen}(\Gamma_Y; i; \mathbf{IA})$, where $\mathsf{linweaken}$

and $\mathsf{strengthen}$ are defined as follows:

$$\begin{array}{lcll}
\mathsf{linweaken}(\Gamma, A^f; 1; ff') & = & 1^{ff'} & \\
\mathsf{linweaken}(\Gamma, A^{\mathbf{I}}; i+1; ff') & = & 1^{\mathbf{II}} \,.\, \mathsf{linweaken}(\Gamma; i; ff') \circ \uparrow & \\
\mathsf{linweaken}(\Gamma, A^l; i+1; ff') & = & 1^{\mathbf{AA}} \,.\, \mathsf{linweaken}(\Gamma; i; ff') \circ \uparrow & \text{if } l \in \{\mathbf{A}, \mathbf{U_A}\} \\
\mathsf{linweaken}(\Gamma, A^l; i+1; ff') & = & 1^{\mathbf{LL}} \,.\, \mathsf{linweaken}(\Gamma; i; ff') \circ \uparrow & \text{if } l \in \{\mathbf{L}, \mathbf{U_L}\} \\
\\
\mathsf{strengthen}(\Gamma, A^f; 1; ff') & = & \Gamma, A^{f'} & \\
\mathsf{strengthen}(\Gamma, A^l; i+1; ff') & = & \mathsf{strengthen}(\Gamma; i; ff'), A^l &
\end{array}$$

Note that $\Gamma \vdash \mathsf{linweaken}(\Gamma; i; ff') : \mathsf{strengthen}(\Gamma; i; ff')$ when the ith assumption in Γ is A^f and ff' is either **IA**, **IL**, or **AL**. When referring to $\mathsf{linweaken}$ we will sometimes leave out the context and simply write $\mathsf{linweaken}(i; ff')$ as Γ can be inferred from the codomain of the substitution.

We can now instantiate Y to $Z[t]$ as shown in the **int-strengthen** rule.

When we cannot apply this rule anymore, and we furthermore cannot apply any of the pruning steps described above, then either M satisfies the three conditions of part 2 of Theorem 4.6.4 or else there is some subterm $M_1 \,\hat{}\, M_2$ or $M_1 \mathbin{@} M_2$ with flexible occurrences in both M_1 and M_2. In the latter case there is really not much we can do.[3] In the former case, we can write the equation $X[s] \doteq M$ as $X[s'][t] \doteq M'[t]$ where $t = \mathsf{linweaken}(n; \mathbf{IA})$. Since t is injective this equation simplifies to $X[s'] \doteq M'$, which corresponds to changing every occurrence of $n^{\mathbf{I}}$ to $n^{\mathbf{A}}$. This is summarized by the rule **int-aff-invert**.

Strengthening affine variables

Consider now the case when $ff' = \mathbf{AL}$, i.e. n is affine. Since we at this point know that n occurs affinely but should occur linearly, no more pruning will be necessary. This means that any flexible occurrence of n in an affine argument, say the ith, of some logic variable Y in M can be strengthened to a linear occurrence. Thus, as is summarized in the **aff-strengthen** rule we instantiate Y to $Z[t]$, where Z is a fresh logic variable with $A_Z = A_Y$, $\Gamma_Z = \mathsf{strengthen}(\Gamma_Y; i; \mathbf{AL})$, and $t = \mathsf{linweaken}(\Gamma_Y; i; \mathbf{AL})$.

Since we know that n is supposed to be linear then it should also occur. If it does not, we can fail with the rule **no-occur**.

If none of the rules **no-occur**, **aff-pos**, **additive**, or **aff-strengthen** apply then $n^{\mathbf{AL}}$ satisfies the four properties of part 3 of Theorem 4.6.4 and we can strengthen n from affine to linear using **aff-lin-invert**.

Example 4.1. As an example we sketch how the algorithm solves equation (4.2) supposing that it has already been lowered.

$$\begin{array}{rcl}
F_1[1^{\mathbf{IL}} . \Uparrow] \doteq c \,\hat{}\, H_1[1^{\mathbf{II}} . \Uparrow] & \mapsto & F_1[1^{\mathbf{IL}} . \Uparrow] \doteq c \,\hat{}\, H_2[1^{\mathbf{IA}} . \Uparrow] \\
& \mapsto & F_1[1^{\mathbf{AL}} . \Uparrow] \doteq c \,\hat{}\, H_2[1^{\mathbf{AA}} . \Uparrow] \\
& \mapsto & F_1[1^{\mathbf{AL}} . \Uparrow] \doteq c \,\hat{}\, H_3[1^{\mathbf{AL}} . \Uparrow] \\
& \mapsto & F_1[1^{\mathbf{LL}} . \Uparrow] \doteq c \,\hat{}\, H_3[1^{\mathbf{LL}} . \Uparrow]
\end{array}$$

The last equation is a pattern, which can be solved directly. Along the way we got the instantiations $H_1 \leftarrow H_2[1^{\mathbf{IA}} . \Uparrow]$ and $H_2 \leftarrow H_3[1^{\mathbf{AL}} . \Uparrow]$.

[3] If we instead of a most general unifier were looking for the set of most general unifiers then we could easily enumerate the different possible solutions by introducing a disjunction and then either prune the variable from M_1 or M_2.

4.6.4 Correctness

The discussion above relies heavily on Theorem 4.6.4 and proves that the algorithm preserves solutions. It is therefore easily possible to generalize part 1 of the Correctness Theorem 4.5.1 to the version of the unification algorithm including linearity pruning. Termination (part 3) also holds for the extended algorithm with a slight elaboration of the termination ordering. When calculating the size of a term we will order the linearity flags $\mathbf{I} > \mathbf{A} > \mathbf{L}$ because with this ordering, the strengthening rules **int-strengthen**, **aff-strengthen**, **int-aff-invert**, and **aff-lin-invert** decrease unification problems in size. Furthermore, we require that every introduction of the $\mathsf{prune}(\cdot;\cdot)$ constraint is followed by a sequence of **prune** steps followed by a **prune-fail** or **prune-finish** step. When the introduction and elimination of the $\mathsf{prune}(\cdot;\cdot)$ constraint are seen together as one step then the combined result always reduces the termination measure, since $\mathsf{prune}(n; M)$ only is introduced if n occurs in M.

However, since the extended algorithm can get stuck on certain equations with a "don't know", we have to accept that progress, as it is stated in part 2 of the theorem, no longer holds. In these cases we can simply report a set of leftover constraints, each of which require strengthening of some intuitionistic variable that occurs flexibly in multiple parts of the right-hand side.

4.6.5 Algorithmic linearity pruning

Now we demonstrate how linearity pruning can be implemented in a bottom-up manner by giving an algorithmic version of the abstract algorithm from above (Figures 4.6 and 4.7).

The algorithmic linearity pruning is trivially sound with respect to the abstract linearity pruning, as every step is derived directly from the non-deterministic linearity pruning rules. We conjecture that completeness also holds for all practical purposes. However, since we are considering the logic variables in a bottom-up manner instead of globally, it is possible to construct corner cases, in which the abstract algorithm gives an answer while the concrete implementation gives a "don't know". If this turns out to be an issue in practice, then one can simply rerun the algorithm as long as the answer is "don't know" and as long as progress is being made (i.e. variables are being pruned). This will achieve completeness with respect to the non-deterministic algorithm, since termination holds.

The implementation-ready algorithm is shown in Figures 4.8, 4.9, and 4.10. We use a call-by-value pseudo-code with pattern matching in the style of SML and keep it mostly pure except for four specific side-effects, which make the presentation a lot easier. The four side-effects are failure, instantiation of logic variables, creation of logic variables, and a single flag called postpone, which can be set and read.

Failure is treated as an exception, i.e. whenever we reach a `Fail` we abort the entire computation. Instantiation of logic variables is a global effect, which instantiates all occurrences in the entire unification problem. Creation of new logic variables with type A and context Γ is written `new(`A`, `Γ`)`.

An important notion in the implementation is abstracted occurrences. They are defined as follows:

$$\alpha ::= \mathbf{0} \mid \mathbf{R} \mid \mathbf{F}(1) \mid \mathbf{F}(2)$$

```
lininvert($X[a_1^{f_1} . a_2^{f_2} \ldots a_{i-1}^{f_{i-1}} . n^{ff'} . s] \doteq M$) =   ($where\ f \neq f'$)
   let ($\alpha$, $M'$) = linprune($n^{ff'}$, $M$)
   let ($\mathbf{R}$, false) = add($ff'$, $\mathbf{R}$, $\alpha$)
   if postpone is set then $cannot\ find\ a\ most\ general\ unifier$
   else if $ff'$=$\mathbf{IL}$ then lininvert($X[a_1^{f_1} . a_2^{f_2} \ldots a_{i-1}^{f_{i-1}} . n^{\mathbf{AL}} . s] \doteq M'$)
   else lininvert($X[a_1^{f_1} . a_2^{f_2} \ldots a_{i-1}^{f_{i-1}} . n^{f'f'} . s] \doteq M'$)
lininvert($X[s] \doteq M$) =   ($where\ s\ is\ a\ pattern\ substitution$)
   solve($X[s] \doteq M$)

prune($n$, $M$) =
   if $n \in_{\mathsf{rig}} M$ or $n \in_{\mathsf{flex},\mathbf{L}} M$ then Fail
   while $n \in M$ do
      $take\ X\ such\ that\ n\ occurs\ in\ the\ ith\ argument\ of\ X\ in\ M$
      let $Y$ = new($A_X$, $\Gamma_X \div i$)
      instantiate $X$ := $Y[\mathsf{weaken}(\Gamma_X; i)]$

mult($\mathbf{0}$, $\alpha$) = $\alpha$
mult($\alpha$, $\mathbf{0}$) = $\alpha$
mult($\mathbf{R}$, $\mathbf{R}$) = Fail
mult($\mathbf{F}$(_), $\mathbf{F}$(_)) = $\mathbf{F}(2)$
mult($\mathbf{R}$, $\mathbf{F}$(_)) = $\mathbf{R}$
mult($\mathbf{F}$(_), $\mathbf{R}$) = $\mathbf{R}$

add(_, $\mathbf{0}$, $\mathbf{0}$) = ($\mathbf{0}$, false)
add($ff'$, $\alpha$, $\mathbf{0}$) = add($ff'$, $\mathbf{0}$, $\alpha$)
add($\mathbf{IA}$, $\mathbf{0}$, $\mathbf{R}$) = ($\mathbf{R}$, false)
add($f\mathbf{L}$, $\mathbf{0}$, $\mathbf{R}$) = Fail
add($\mathbf{IA}$, $\mathbf{0}$, $\mathbf{F}(n)$) = ($\mathbf{F}(n)$, false)
add($\mathbf{IL}$, $\mathbf{0}$, $\mathbf{F}$(_)) = ($\mathbf{0}$, true)
add(_, $\mathbf{R}$, $\mathbf{R}$) = ($\mathbf{R}$, false)
add($ff'$, $\alpha$, $\mathbf{R}$) = add($ff'$, $\mathbf{R}$, $\alpha$)
add($\mathbf{I}f'$, $\mathbf{R}$, $\mathbf{F}(1)$) = ($\mathbf{R}$, false)
add($\mathbf{I}f'$, $\mathbf{R}$, $\mathbf{F}(2)$) = set postpone; ($\mathbf{R}$, false)
add($\mathbf{I}f'$, $\mathbf{F}(1)$, $\mathbf{F}(1)$) = ($\mathbf{F}(1)$, false)
add($\mathbf{I}f'$, $\mathbf{F}$(_), $\mathbf{F}$(_)) = ($\mathbf{F}(2)$, false)
```

Figure 4.8: Linearity pruning algorithm

```
linprune($n^{\mathbf{I}f}$, $n^{\mathbf{I}}$) = ($\mathbf{R}$, $n^{\mathbf{A}}$)
linprune($n^{\mathbf{AL}}$, $n^{\mathbf{A}}$) = ($\mathbf{R}$, $n^{\mathbf{L}}$)
linprune($n^{ff'}$, $m^{f''}$) = ($\mathbf{0}$, $m^{f''}$)
linprune($n^{ff'}$, $\langle M,N \rangle$) =
   let ($\alpha_1$, $M'$) = linprune($n^{ff'}$, $M$)
   let ($\alpha_2$, $N'$) = linprune($n^{ff'}$, $N$)
   let ($\alpha$, $p$) = add($\alpha_1$, $\alpha_2$)
   if $p$ then prune($n$, $\langle M',N' \rangle$)
   ($\alpha$, $\langle M',N' \rangle$)
linprune($n^{ff'}$, $\mathsf{fst}\, M$) = ($\alpha$, $\mathsf{fst}\, M'$)
   where ($\alpha$, $M'$) = linprune($n^{ff'}$, $M$)
linprune($n^{ff'}$, $\mathsf{snd}\, M$) = ($\alpha$, $\mathsf{snd}\, M'$)
   where ($\alpha$, $M'$) = linprune($n^{ff'}$, $M$)
linprune($n^{ff'}$, $M\ N$) =
   let ($\alpha$, $M'$) = linprune($n^{ff'}$, $M$)
   prune($n$, $N$)
   ($\alpha$, $M'\ N$)
linprune($n^{f\mathbf{L}}$, $M @ N$) =
   let ($\alpha$, $M'$) = linprune($n^{f\mathbf{L}}$, $M$)
   prune($n$, $N$)
   ($\alpha$, $M' @ N$)
linprune($n^{\mathbf{IA}}$, $M @ N$) =
   let ($\alpha_1$, $M'$) = linprune($n^{\mathbf{IA}}$, $M$)
   let ($\alpha_2$, $N'$) = linprune($n^{\mathbf{IA}}$, $N$)
   if $\alpha_1$=$\mathbf{F}$(_) and $\alpha_2$=$\mathbf{R}$ then prune($n$, $M'$)
   if $\alpha_1$=$\mathbf{R}$ and $\alpha_2$=$\mathbf{F}$(_) then prune($n$, $N'$)
   (mult($\alpha_1$, $\alpha_2$), $M' @ N'$)
linprune($n^{ff'}$, $M \frown N$) =
   let ($\alpha_1$, $M'$) = linprune($n^{ff'}$, $M$)
   let ($\alpha_2$, $N'$) = linprune($n^{ff'}$, $N$)
   if $\alpha_1$=$\mathbf{F}$(_) and $\alpha_2$=$\mathbf{R}$ then prune($n$, $M'$)
   if $\alpha_1$=$\mathbf{R}$ and $\alpha_2$=$\mathbf{F}$(_) then prune($n$, $N'$)
   (mult($\alpha_1$, $\alpha_2$), $M' \frown N'$)
linprune($n^{ff'}$, $\lambda M$) = ($\alpha$, $\lambda M'$)
   where ($\alpha$, $M'$) = linprune($(n+1)^{ff'}$, $M$)
linprune($n^{ff'}$, $\mathring{\lambda} M$) = ($\alpha$, $\mathring{\lambda} M'$)
   where ($\alpha$, $M'$) = linprune($(n+1)^{ff'}$, $M$)
linprune($n^{ff'}$, $\widehat{\lambda} M$) = ($\alpha$, $\widehat{\lambda} M'$)
   where ($\alpha$, $M'$) = linprune($(n+1)^{ff'}$, $M$)
```

Figure 4.9: Linearity pruning algorithm

```
linprune($n^{\mathbf{I}f'}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{II}} . s]$) =
    let $Y$ = new($A_X$, $\mathsf{strengthen}(\Gamma_X; i; \mathbf{IA})$)
    instantiate $X$ := $Y[\mathsf{linweaken}(i; \mathbf{IA})]$
    ($\mathbf{F}$(1), $Y[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{AA}} . s]$)
linprune($n^{\mathbf{I}f'}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{IA}} . s]$) =
    ($\mathbf{F}$(1), $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{AA}} . s]$)
linprune($n^{\mathbf{I}f'}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{IL}} . s]$) = ($\mathbf{R}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{AL}} . s]$)
linprune($n^{\mathbf{AL}}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{AA}} . s]$) =
    let $Y$ = new($A_X$, $\mathsf{strengthen}(\Gamma_X; i; \mathbf{AL})$)
    instantiate $X$ := $Y[\mathsf{linweaken}(i; \mathbf{AL})]$
    ($\mathbf{R}$, $Y[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{LL}} . s]$)
linprune($n^{\mathbf{AL}}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{AL}} . s]$) = ($\mathbf{R}$, $X[a_1^{f_1} . a_2^{f_2} \dots a_{i-1}^{f_{i-1}} . n^{\mathbf{LL}} . s]$)
linprune($n^{ff'}$, $X[s]$) = ($\mathbf{0}$, $X[s]$) ($\textit{where } n \notin s$)
```

Figure 4.10: Linearity pruning algorithm

The meaning of $\mathbf{0}$ is no occurrence, **R** (read: **R**igid occurrence) means an occurrence that cannot be pruned away, and $\mathbf{F}(n)$ means one or more occurrences all of which are flexible and can be pruned away. Furthermore $\mathbf{F}(1)$ (read: **One F**lexible occurrence[4]) means that the occurrences are uniquely positioned and can be strengthened from intuitionistic to affine, while $\mathbf{F}(2)$ (read: **Two** or more **F**lexible occurrences) means that there are flexible occurrences in both branches of some multiplicative split in the term.

The implementation is split into five functions: `lininvert`, `prune`, `mult`, `add`, and `linprune`.

The functions `mult` and `add` combine abstracted occurrences; `mult` simply calculates the combined occurrence at a multiplicative split, while `add` calculates the combined occurrence at an additive split and returns a boolean indicating whether the occurrences should be pruned away.

The function `prune(`n`, `M`)` prunes away all occurrences of n in M. The pseudo-code implementation of this is quite vague, since it can be done similarly to the regular pruning in e.g. intuitionistic pattern unification.[5]

The function `lininvert` is a top-level loop that goes through all the linear-changing extensions and `linprune` is doing the actual work of traversing the term. The final call to `solve` represents the return to the ordinary pattern unification algorithm.

The invariant of `linprune(`$n^{ff'}$`, `M`)` is that n needs to be strengthened from f to f' in M. The return value `(`α`, `M'`)` is the strengthened term M' and the resulting abstracted occurrence of n in M'. If $ff' = \mathbf{IL}$ then n will only be strengthened from **I** to **A** in M' and another traversal is needed to complete

[4] $\mathbf{F}(1)$ can refer to multiple occurrences in case of additive context splits, but the intuition of a single occurrence is nice.

[5] By using a technique of lazily evaluated hereditary substitutions and a special term denoted "undefined" to represent variables that need to be pruned (as it is done in e.g. Twelf and Celf), all the calls to `prune(`n`, `M`)` can be executed in constant time and the actual pruning can then be done with a single traversal of the term. This avoids a potential exponential blow-up.

the strengthening to $\mathbf{L}$. Since a linear occurrence cannot be pruned away and `linprune(`$n^{\mathbf{AL}}$`,` M`)` strengthens n to be linear then the return value will never be $\mathbf{F}(n)$ in this case. If `linprune` returns an $\alpha \neq \mathbf{F}(2)$ and the `postpone` flag is not set then M' is well-typed in the context where n has been strengthened to f'. If the return value is $\alpha = \mathbf{F}(2)$ then M' is well-typed under the condition that n is pruned away, since it contains several flexible occurrences (just strengthened from intuitionistic to affine) spread across at least one multiplicative context split. And if an $\mathbf{F}(2)$ occurrence is encountered in a position where it should not necessarily be pruned, i.e. at an additive split next to an $\mathbf{R}$ occurrence or at the top-level (which is represented as the final call to `add(`ff'`,` $\mathbf{R}$`,` α`)` in `lininvert`), then we set the `postpone` flag, which indicates that if we cannot fail we have to answer "don't know".

The top-level call to `add(`ff'`,` $\mathbf{R}$`,` α`)` in `lininvert` serves to check the returned α. This is because the top-level requirement about n being available in the context is equivalent to the availability requirement next to a rigid occurrence in an additive split.

Chapter 5

The CLF Type Theory

5.1 Summary

In this chapter we present the CLF type theory. We recall the original version as it was introduced in [CPWW02a] (section 5.2) and discuss its redesign (section 5.3). Then we present our version of CLF (section 5.4). Finally, we extend it with logic variables and redices (section 5.5).

5.2 The original CLF type theory

The CLF type theory is defined in [CPWW02a]. It expands the dependently typed logical framework LF by linear types; it includes the additive fragment directly and restricts the multiplicative fragment to occur within a monadic type constructor $\{S\}$. The syntax of CLF is given below for easy reference.

$$
\begin{array}{rcll}
A, B & ::= & A \multimap B \mid \Pi x{:}A.\, B \mid A \mathbin{\&} B \mid \top \mid \{S\} \mid P & \textit{Asynchronous types} \\
P & ::= & a \mid P\, N & \textit{Atomic type constructors} \\
S & ::= & S_1 \otimes S_2 \mid 1 \mid \exists x{:}A.\, S \mid A & \textit{Synchronous types} \\
\\
N & ::= & \widehat{\lambda} x.\, N \mid \lambda x.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R & \textit{Normal objects} \\
R & ::= & c \mid x \mid R \mathbin{\hat{}} N \mid R\, N \mid \pi_1 R \mid \pi_2 R & \textit{Atomic objects} \\
E & ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M & \textit{Expressions} \\
M & ::= & M_1 \otimes M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\
p & ::= & p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns}
\end{array}
$$

Additionally two important constructs are defined as syntactic sugar. The intuitionistic function space is defined as $A \to B \equiv \Pi x{:}A.\, B$ where x does not occur in B, and the exponential of linear logic is defined as $!A \equiv \exists x{:}A.\, 1$.

The equational theory of CLF is α-, β-, and η-equality along with a permutative conversion called let-floating:[1]

$$\big(\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E\big) \equiv \big(\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E\big)$$

[1] We will assume tacit α-renaming throughout this chapter.

This equality is of course subject to the side-condition that the bindings are independent: p_1 and p_2 must bind disjoint sets of variables, no variable bound by p_1 can appear free in R_2, and vice versa.

5.3 Redesigning the CLF type theory

5.3.1 Affine types and $\top$

During the design of a unification algorithm for CLF it became clear that $\top$ was problematic. Due to the complexity and undecidability of general higher-order unification, it was desirable to come up with a pattern unification algorithm building on the previous success for LF [Mil91, DHKP98]. Two of the cornerstones of pattern unification are decomposition and invertible substitutions. We will demonstrate how $\top$ breaks both of these notions.

Consider the following equation in some linear type theory with $\top$ (e.g. CLF)

$$c \mathbin{\hat{}} \langle\rangle \mathbin{\hat{}} (G \mathbin{\hat{}} x) = c \mathbin{\hat{}} \langle\rangle \mathbin{\hat{}} (d \mathbin{\hat{}} H)$$

where c and d are constants, G and H are ground logic variables, and x is a linear variable. A natural approach to such an equation is to decompose it into the two equations $\langle\rangle = \langle\rangle$ and $G \mathbin{\hat{}} x = d \mathbin{\hat{}} H$. But now the second equation is not well-typed no matter how we split the context. This seems to indicate that the original equation has no solution, but because of $\top$ we actually have a solution (in fact a most general unifier):

$$\begin{aligned} G &= \hat{\lambda} x.\, d \mathbin{\hat{}} (H' \mathbin{\hat{}} \langle\rangle) \\ H &= H' \mathbin{\hat{}} \langle\rangle \end{aligned}$$

This means that simple decomposition of equations is not sound when the type theory contains $\top$.

Now let us consider the identity substitution on the context containing a single linear variable x. This substitution is clearly invertible, since it is its own inverse. We write this substitution as x/x and its typing as $x \hat{:} A \vdash x/x : x \hat{:} A$. If $A = \top$ then the substitution is η-equivalent to $\langle\rangle/x$. If we imagine this substitution occurring inside a larger term in which $\langle\rangle$ occurs somewhere else as well, e.g. $c \mathbin{\hat{}} G[\langle\rangle/x] \mathbin{\hat{}} \langle\rangle$, then the typing that contains $x \hat{:} \top \vdash \langle\rangle/x : x \hat{:} \top$ is equivalent to another typing in which our identity substitution is typed in an empty context. I.e. we can push the linear resource $x \hat{:} \top$ to the other $\langle\rangle$. Now the typing looks like $\cdot \vdash \langle\rangle/x : x \hat{:} \top$ and there exists no substitution s with the typing $x \hat{:} \top \vdash s : \cdot$, i.e. we have lost the existence of an inverse substitution. This means that the existence of inverse substitutions is not preserved by the equivalence relation on typings induced by $\top$.

Therefore we decided to redesign CLF into a version without $\top$. But this means that we have to reconsider any development in CLF using $\top$.

We conjecture that most uses of $\top$ can be replaced by using affine resources appropriately. Considering our suite of CLF examples [CPWW02b, WCPW08] this has so far been the case. This is because all the uses of $\top$ that we have encountered have been to allow weakening for some of the linear resources, and in fact this was already remarked in [WCPW08]. In these cases specifying the linear resources that we expect to weaken as affine resources instead is actually more accurate and thus suites the purpose better.

We will consider one such example (the complete example can be found in [CPWW02b]):

Example 5.1 (MiniML with references encoded in CLF)**.** Destination passing style is a useful CLF representation technique for uniformly specifying the semantics of a wide range of programming language constructs, such as lazy evaluation, concurrency, and references.

Destinations can be thought of as a kind of locations, and they are used to link together an expression to be evaluated and the continuation waiting for the result, thus avoiding the explicit representation of continuations or evaluation contexts. In [CPWW02b] destinations are represented by an empty type `dest` and can thus only be constructed as parameters during evaluation. The type `exp` represents the encoding of MiniML syntax. For instance, we might have lambdas and applications encoded as the constructors $\texttt{lam} : (\texttt{exp} \to \texttt{exp}) \to \texttt{exp}$ and $\texttt{app} : \texttt{exp} \to \texttt{exp} \to \texttt{exp}$.

Two type families, $\texttt{eval} : \texttt{exp} \to \texttt{dest} \to \texttt{type}$ and $\texttt{return} : \texttt{exp} \to \texttt{dest} \to \texttt{type}$, are used to represent the semantics. A linear assumption of type $\texttt{eval}\ E\ D$ represents the instruction to evaluate the expression E, and a linear assumption of type $\texttt{return}\ V\ D$ represents a completed evaluation with return value V. The D is the destination that links the two. The semantics is then encoded as linear implications from `eval`s to `return`s.

References are encoded by representing each cell as a linear resource in the context, which can then be overwritten simply by consuming it and recreating it with the new value. An entire evaluation run is represented by the type $\Pi d{:}\texttt{dest}.\, \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V\ d \otimes \top\}$. Reading this from left to right, we first introduce a destination d and then introduce the instruction to evaluate E at destination d. The computation is then finished with result V when we reach a `return` at the same destination d. Here the $\top$ is necessary to consume all the reference cells, which invariably are going to be left over.

This use of $\top$ feels a bit like a hack, but with the original CLF type theory it was necessary. If we reconsider the development in a type theory with affine types instead of $\top$ we find that reference cells can be represented much more naturally as affine resources, because this allows us to weaken them away when they are no longer needed (essentially garbage-collect them), thus removing the need for $\top$.

Replacing $\top$ by an affine implication changes the asynchronous types to:

$$A, B ::= A \multimap B \mid A \multimap\!\!@\, B \mid \Pi x{:}A.\, B \mid A \,\&\, B \mid \{S\} \mid P$$

We will also need to extend the synchronous types to accommodate affine resources, i.e. include the affine modality $S ::= \ldots \mid @A$. This leads to a disparity between the affine and intuitionistic modalities with one being included directly and the other being defined as syntactic sugar. We see this as an incentive to also include the intuitionistic modality directly, and we will return to this below.

5.3.2 Linear and dependent implication from a focused perspective

The development of focused linear logic [And92, LM07] has led to a better understanding of the CLF type theory. It has led to a change in terminology;

asynchronous types are called negative types and synchronous types are called positive types. The inclusions of positive and negative types in each other are called up- and down-shifts. The CLF monad corresponds to the up-shift in focused linear logic and the silent inclusion $S ::= \ldots \mid A$ is the down-shift. We will in the following make this silent inclusion more explicit by writing the down-shift operator explicitly, i.e. $S ::= \ldots \mid {\downarrow}A$.

Focusing has also led to the discovery that the $A \multimap B$ is really an instance of $S \multimap B$ with the positive S restricted to ${\downarrow}A$. This restriction of $\multimap$ is in the light of focusing a bit arbitrary and does not really gain us anything. We will therefore remove it.

Now we can express the affine implication neatly as $A \mathrel{-\!\!@} B \equiv @A \multimap B$. Similarly we should have $A \rightarrow B \equiv {!A} \multimap B$, but this just makes our syntactic sugar definition of $!A$ seem increasingly more clumsy. We will therefore include $!A$ directly and abandon the previous identity $!A \equiv \exists x{:}A.\, 1$.

These choices give rise to new questions. Usually linear logic and dependent types are studied independently, each of them having different definitions of the intuitionistic implication. Putting them together we get $!A \multimap B \equiv A \rightarrow B \equiv \Pi x{:}A.\, B$. Both linear implication and Π are more general than intuitionistic implication, but neither is more general than the other. This raises the question of whether there is a more general construct that can express both $\multimap$ and Π. The answer is indeed yes; a general pattern binding Π that is only allowed to be dependent in the intuitionistic parts of the pattern generalizes both $\multimap$ and Π. This new Π is written $\widehat{\Pi} p{:}S.\, B$, and we have the following identities:

$$\begin{aligned}
\widehat{\Pi} !x{:}!A.\, B &\equiv \Pi x{:}A.\, B \\
\widehat{\Pi} p{:}S.\, B &\equiv S \multimap B && \text{if intuitionistic variables in } p \text{ do not occur in } B \\
\widehat{\Pi} !x{:}!A.\, B &\equiv A \rightarrow B && \text{if } x \text{ does not occur in } B \\
\widehat{\Pi} @x{:}@A.\, B &\equiv A \mathrel{-\!\!@} B
\end{aligned}$$

In the latter case we do not need any side condition on the occurrence of x, since by definition only intuitionistic variables in p are allowed to occur in B in $\widehat{\Pi} p{:}S.\, B$. Similarly, $\widehat{\Pi} {\downarrow}x{:}{\downarrow}A.\, B \equiv {\downarrow}A \multimap B$ holds without any side condition. Now all of $\multimap$, $\rightarrow$, $\mathrel{-\!\!@}$, and Π are simply syntactic sugar with the expected equalities: $!A \multimap B \equiv A \rightarrow B \equiv \Pi x{:}A.\, B$ and $@A \multimap B \equiv A \mathrel{-\!\!@} B$.

5.3.3 Dependent pairs and tensor

Looking at the positive types we can compare our two pair-constructs. When x does not occur in S then $\exists x{:}A.\, S$ and $!A \otimes S$ are very similar. We can generalize the dependent and multiplicative pair into one common construct similar to how we generalized dependent and linear implication in the previous section. The new pair-construct is written $\widehat{\exists} p{:}S_1.\, S_2$ and again only intuitionistic variables in p are allowed to occur in S_2. Our two previous pair-constructs are now syntactic sugar with the following identities:

$$\begin{aligned}
&\widehat{\exists} !x{:}!A.\, S \equiv \exists x{:}A.\, S \\
&\widehat{\exists} p{:}S_1.\, S_2 \equiv S_1 \otimes S_2 && \text{if intuitionistic variables in } p \text{ does not occur in } S_2
\end{aligned}$$

This gives us the following equality: $\exists x{:}A.\, S \equiv {!A} \otimes S$ if x does not occur in S.

5.4 CLF version 2

Now we present the reworked type theory in its entirety. We will use A^- and A^+ for negative and positive types instead of A and S. Additionally, the presentation is changed to use spine notation.

We present the type theory using canonical forms and hereditary substitutions, and thus the only equalities left to check in the typing rules is the ones arising from let-floating.

5.4.1 Syntax

$$\begin{array}{rcll}
K & ::= & \mathsf{type} \mid \Pi x{:}A^-.\, K & \textit{Kinds} \\
A^- & ::= & \widehat{\Pi} p{:}A^+.\, A^- \mid A_1^- \;\&\; A_2^- \mid \{A^+\} \mid a \cdot T & \textit{Negative types} \\
A^+ & ::= & \widehat{\exists} p{:}A_1^+.\, A_2^+ \mid \downarrow A^- \mid {!A^-} \mid @A^- \mid \mathbf{1} & \textit{Positive types} \\
T & ::= & N; T \mid () & \textit{Type spines} \\
M & ::= & \langle\!\langle M_1, M_2 \rangle\!\rangle \mid \downarrow N \mid {!N} \mid @N \mid \mathbf{1} & \textit{Monadic objects} \\
N & ::= & \widehat{\lambda} p.\, N \mid \langle N_1, N_2 \rangle \mid \{E\} \mid H \cdot S & \textit{Normal objects} \\
H & ::= & x \mid c & \textit{Heads} \\
S & ::= & M; S \mid \pi_1; S \mid \pi_2; S \mid () & \textit{Spines} \\
E & ::= & \mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ E \mid M & \textit{Expressions} \\
p & ::= & \langle\!\langle p_1, p_2 \rangle\!\rangle \mid \downarrow x \mid {!x} \mid @x \mid \mathbf{1} & \textit{Patterns} \\
\Gamma, \Phi, \Delta & ::= & \Gamma, x{:}A^- \mid \cdot & \textit{Contexts} \\
\Sigma & ::= & \Sigma, a{:}K \mid \Sigma, c{:}A^- \mid \cdot & \textit{Signatures}
\end{array}$$

5.4.2 Judgments

The typing rules are shown in Figures 5.1 and 5.2. The judgments make use of three contexts; an intuitionistic context Γ, an affine context Φ, and a linear context Δ. The affine and the linear contexts are considered unordered, whereas the intuitionistic context is ordered. As usual, we assume that all variable names are distinct.

Square brackets to the left of the turnstile denote an assumption in left focus.

There is a global signature Σ, which is constant throughout all the rules (except of course the signature validity rules). Σ gives kinds and types for type and term constants. We write $a{:}K \in \Sigma$ and $c{:}A^- \in \Sigma$ for signature lookups.

The substitutions that occur in the judgments are hereditary, which means that the only equality we need to check is let-floating. This is seen as the premise $T \equiv T'$ in the rule for $H \cdot S$.

The definition of hereditary substitution, $[N/x]_B$, and type erasure, $\lfloor A \rfloor$, is given below in section 5.4.3.

Notice that when patterns are bound in types the affine and linear parts are thrown away, thus ensuring that any dependencies are only on the intuitionistic parts.

Signatures

$\boxed{\vdash \Sigma : \mathsf{sig}}$

$$\frac{}{\vdash \cdot : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \quad \cdot \vdash_\Sigma K : \mathsf{kind}}{\vdash \Sigma, a{:}K : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \quad \cdot \vdash_\Sigma A^- : \mathsf{type}}{\vdash \Sigma, c{:}A^- : \mathsf{sig}}$$

Kinds

$\boxed{\Gamma \vdash_\Sigma K : \mathsf{kind}}$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type} : \mathsf{kind}} \qquad \frac{\Gamma \vdash_\Sigma A^- : \mathsf{type} \quad \Gamma, x{:}A^- \vdash_\Sigma K : \mathsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A^-.\, K : \mathsf{kind}}$$

Types

$\boxed{\Gamma \vdash_\Sigma A^- : \mathsf{type}}$

$$\frac{\Gamma \vdash_\Sigma A^+ : \mathsf{type} \quad \Gamma; \cdot; \cdot; p{:}A^+ \vdash \Gamma'; \Phi; \Delta \quad \Gamma' \vdash_\Sigma A^- : \mathsf{type}}{\Gamma \vdash_\Sigma \widehat{\Pi} p{:}A^+.\, A^- : \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma A^+ : \mathsf{type}}{\Gamma \vdash_\Sigma \{A^+\} : \mathsf{type}}$$

$$\frac{\Gamma \vdash_\Sigma A_1^- : \mathsf{type} \quad \Gamma \vdash_\Sigma A_2^- : \mathsf{type}}{\Gamma \vdash_\Sigma A_1^- \mathbin{\&} A_2^- : \mathsf{type}} \qquad \frac{a{:}K \in \Sigma \quad \Gamma; [K] \vdash_\Sigma T : \mathsf{type}}{\Gamma \vdash_\Sigma a \cdot T : \mathsf{type}}$$

$\boxed{\Gamma \vdash_\Sigma A^+ : \mathsf{type}}$

$$\frac{\Gamma \vdash_\Sigma A_1^+ : \mathsf{type} \quad \Gamma; \cdot; \cdot; p{:}A^+ \vdash \Gamma'; \Phi; \Delta \quad \Gamma' \vdash_\Sigma A_2^+ : \mathsf{type}}{\Gamma \vdash_\Sigma \widehat{\exists} p{:}A_1^+.\, A_2^+ : \mathsf{type}}$$

$$\frac{\Gamma \vdash_\Sigma A^- : \mathsf{type}}{\Gamma \vdash_\Sigma {\downarrow}A^- : \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma A^- : \mathsf{type}}{\Gamma \vdash_\Sigma {!}A^- : \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma A^- : \mathsf{type}}{\Gamma \vdash_\Sigma @A^- : \mathsf{type}} \qquad \frac{}{\Gamma \vdash_\Sigma \mathbf{1} : \mathsf{type}}$$

Type spines

$\boxed{\Gamma; [K] \vdash_\Sigma T : \mathsf{type}}$

$$\frac{}{\Gamma; [\mathsf{type}] \vdash_\Sigma () : \mathsf{type}} \qquad \frac{\Gamma; \cdot; \cdot \vdash_\Sigma N \Leftarrow A^- \quad \Gamma; [K[N/x]_{\lfloor A^- \rfloor}] \vdash_\Sigma T : \mathsf{type}}{\Gamma; [\Pi x{:}A^-.\, K] \vdash_\Sigma N; T : \mathsf{type}}$$

Patterns

$\boxed{\Gamma; \Phi; \Delta; p{:}A^+ \vdash \Gamma'; \Phi'; \Delta'}$

$$\frac{\Gamma; \Phi; \Delta; p_1{:}A_1^+ \vdash \Gamma'; \Phi'; \Delta' \quad \Gamma'; \Phi'; \Delta'; p_2{:}A_2^+ \vdash \Gamma''; \Phi''; \Delta''}{\Gamma; \Phi; \Delta; \langle\!\langle p_1, p_2 \rangle\!\rangle{:}\widehat{\exists} p_1{:}A_1^+.\, A_2^+ \vdash \Gamma''; \Phi''; \Delta''}$$

$$\frac{}{\Gamma; \Phi; \Delta; {\downarrow}x{:}{\downarrow}A^- \vdash \Gamma; \Phi; (\Delta, x{:}A^-)} \qquad \frac{}{\Gamma; \Phi; \Delta; {!}x{:}{!}A^- \vdash (\Gamma, x{:}A^-); \Phi; \Delta}$$

$$\frac{}{\Gamma; \Phi; \Delta; @x{:}@A^- \vdash \Gamma; (\Phi, x{:}A^-); \Delta} \qquad \frac{}{\Gamma; \Phi; \Delta; \mathbf{1}{:}\mathbf{1} \vdash \Gamma; \Phi; \Delta}$$

Figure 5.1: Revised CLF type theory

Terms

$$\boxed{\Gamma; \Phi; \Delta \vdash_\Sigma N \Leftarrow A^-}$$

$$\frac{\Gamma; \Phi; \Delta; p{:}A^+ \vdash \Gamma'; \Phi'; \Delta' \quad \Gamma'; \Phi'; \Delta' \vdash_\Sigma N \Leftarrow A^-}{\Gamma; \Phi; \Delta \vdash_\Sigma \widehat{\lambda} p.\, N \Leftarrow \widehat{\Pi} p{:}A^+.\, A^-}$$

$$\frac{\Gamma; \Phi; \Delta \vdash_\Sigma N_1 \Leftarrow A_1^- \quad \Gamma; \Phi; \Delta \vdash_\Sigma N_2 \Leftarrow A_2^-}{\Gamma; \Phi; \Delta \vdash_\Sigma \langle N_1, N_2 \rangle \Leftarrow A_1^- \mathbin{\&} A_2^-} \qquad \frac{\Gamma; \Phi; \Delta \vdash_\Sigma E \Leftarrow A^+}{\Gamma; \Phi; \Delta \vdash_\Sigma \{E\} \Leftarrow \{A^+\}}$$

$$\frac{\Gamma; \Phi; \Delta; [H] \vdash_\Sigma S \Rightarrow a \cdot T' \quad T \equiv T'}{\Gamma; \Phi; \Delta \vdash_\Sigma H \cdot S \Leftarrow a \cdot T}$$

$$\boxed{\Gamma; \Phi; \Delta \vdash_\Sigma M \Leftarrow A^+}$$

$$\frac{\Gamma; \Phi_1; \Delta_1 \vdash_\Sigma M_1 \Leftarrow A_1^+ \quad \Gamma; \Phi_2; \Delta_2 \vdash_\Sigma M_2 \Leftarrow A_2^+[M_1/p]_{\lfloor A_1^+ \rfloor}}{\Gamma; (\Phi_1, \Phi_2); (\Delta_1, \Delta_2) \vdash_\Sigma \langle\!\langle M_1, M_2 \rangle\!\rangle \Leftarrow \widehat{\exists} p{:}A_1^+.\, A_2^+} \qquad \frac{}{\Gamma; \Phi; \cdot \vdash_\Sigma \mathbf{1} \Leftarrow \mathbf{1}}$$

$$\frac{\Gamma; \Phi; \Delta \vdash_\Sigma N \Leftarrow A^-}{\Gamma; \Phi; \Delta \vdash_\Sigma {\downarrow} N \Leftarrow {\downarrow} A^-} \qquad \frac{\Gamma; \cdot; \cdot \vdash_\Sigma N \Leftarrow A^-}{\Gamma; \Phi; \cdot \vdash_\Sigma {!}N \Leftarrow {!}A^-} \qquad \frac{\Gamma; \Phi; \cdot \vdash_\Sigma N \Leftarrow A^-}{\Gamma; \Phi; \cdot \vdash_\Sigma @N \Leftarrow @A^-}$$

$$\boxed{\Gamma; \Phi; \Delta \vdash_\Sigma E \Leftarrow A^+}$$

$$\frac{\Gamma; \Phi_1; \Delta_1; [H] \vdash_\Sigma S \Rightarrow \{A_2^+\} \quad \Gamma; \Phi_2; \Delta_2; p{:}A_2^+ \vdash \Gamma'; \Phi'; \Delta' \quad \Gamma'; \Phi'; \Delta' \vdash_\Sigma E \Leftarrow A_1^+}{\Gamma; (\Phi_1, \Phi_2); (\Delta_1, \Delta_2) \vdash_\Sigma \mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ E \Leftarrow A_1^+}$$

$$\frac{\Gamma; \Phi; \Delta \vdash_\Sigma M \Leftarrow A^+}{\Gamma; \Phi; \Delta \vdash_\Sigma M \Leftarrow A^+}$$

Term spines

$$\boxed{\Gamma; \Phi; \Delta; [H] \vdash_\Sigma S \Rightarrow A^-}$$

$$\frac{(\Gamma_1, x{:}A_1^-, \Gamma_2); \Phi; \Delta; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}{(\Gamma_1, x{:}A_1^-, \Gamma_2); \Phi; \Delta; [x] \vdash_\Sigma S \Rightarrow A_2^-} \qquad \frac{\Gamma; \Phi; \Delta; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; (\Phi, x{:}A_1^-); \Delta; [x] \vdash_\Sigma S \Rightarrow A_2^-}$$

$$\frac{\Gamma; \Phi; \Delta; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; \Phi; (\Delta, x{:}A_1^-); [x] \vdash_\Sigma S \Rightarrow A_2^-} \qquad \frac{c{:}A_1^- \in \Sigma \quad \Gamma; \Phi; \Delta; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; \Phi; \Delta; [c] \vdash_\Sigma S \Rightarrow A_2^-}$$

$$\boxed{\Gamma; \Phi; \Delta; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}$$

$$\frac{\Gamma; \Phi_1; \Delta_1 \vdash_\Sigma M \Leftarrow A^+ \quad \Gamma; \Phi_2; \Delta_2; [A_1^-[M/p]_{\lfloor A^+ \rfloor}] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; (\Phi_1, \Phi_2); (\Delta_1, \Delta_2); [\widehat{\Pi} p{:}A^+.\, A_1^-] \vdash_\Sigma (M; S) \Rightarrow A_2^-}$$

$$\frac{\Gamma; \Phi; \Delta; [A_i^-] \vdash_\Sigma S \Rightarrow A_3^-}{\Gamma; \Phi; \Delta; [A_1^- \mathbin{\&} A_2^-] \vdash_\Sigma \pi_i; S \Rightarrow A_3^-} \qquad \frac{}{\Gamma; \Phi; \cdot; [A^-] \vdash_\Sigma () \Rightarrow A^-}$$

Figure 5.2: Revised CLF type theory

Context Validity Assumption. Anything to the left of a turnstile is assumed to be valid in the intuitionistic context. That is, for any $x{:}A^-$ occurring in Φ or Δ we assume $\Gamma \vdash_\Sigma A^- : \mathsf{type}$, and for any split of the intuitionistic context $\Gamma = \Gamma_1, x{:}A^-, \Gamma_2$ we assume $\Gamma_1 \vdash_\Sigma A^- : \mathsf{type}$. Additionally, any kind or type occurring in focus on the left, $[K]$ or $[A^-]$, is assumed to be valid in Γ, that is, $\Gamma \vdash_\Sigma K : \mathsf{kind}$ and $\Gamma \vdash_\Sigma A^- : \mathsf{type}$. Finally, any type occurring to the right of a $\Leftarrow$ is assumed to be valid in Γ.

When the rules in Figures 5.1 and 5.2 are read from bottom to top and subgoals are checked from left to right, the type system works as a bidirectional type checking algorithm. Judgments of the form $\cdots \vdash_\Sigma \cdot \Leftarrow \cdot$ checks that the term has a given type, and judgments of the form $\cdots \vdash_\Sigma \cdot \Rightarrow \cdot$ infers a type. Notice that if we assume $\vdash \Sigma : \mathsf{sig}$ then every rule preserves the Context Validity Assumption.

5.4.3 Hereditary substitution

The hereditary substitution function is indexed by a simple type B, which maintains the type of the object being substituted. This type index can be left out, but it allows one to easily check that hereditary substitution is indeed terminating. Also, it corresponds nicely to the type ascription needed for explicit redices (see section 5.5.1 below).

The syntax of simple types and the erasure function is as follows:

$$\begin{array}{lll} B^- ::= B^+ \multimap B^- \mid B_1^- \mathbin{\&} B_2^- \mid \{B^+\} \mid a & & \textit{Negative types} \\ B^+ ::= B_1^+ \otimes B_2^+ \mid {\downarrow}B^- \mid {!}B^- \mid @B^- \mid \mathbf{1} & & \textit{Positive types} \end{array}$$

$$\begin{array}{rcl@{\qquad}rcl} \lfloor \widehat{\Pi} p{:}A^+.\, A^- \rfloor &=& \lfloor A^+ \rfloor \multimap \lfloor A^- \rfloor & \lfloor \widehat{\exists} p{:}A_1^+.\, A_2^+ \rfloor &=& \lfloor A_1^+ \rfloor \otimes \lfloor A_2^+ \rfloor \\ \lfloor A_1^- \mathbin{\&} A_2^- \rfloor &=& \lfloor A_1^- \rfloor \mathbin{\&} \lfloor A_2^- \rfloor & \lfloor {\downarrow}A^- \rfloor &=& {\downarrow}\lfloor A^- \rfloor \\ \lfloor \{A^+\} \rfloor &=& \{\lfloor A^+ \rfloor\} & \lfloor {!}A^- \rfloor &=& {!}\lfloor A^- \rfloor \\ \lfloor a \cdot T \rfloor &=& a & \lfloor @A^- \rfloor &=& @\lfloor A^- \rfloor \\ &&& \lfloor \mathbf{1} \rfloor &=& \mathbf{1} \end{array}$$

The head of a type with respect to a spine $\mathsf{hd}(B^-, S)$ is given as:

$$\begin{array}{l} \mathsf{hd}(B^+ \multimap B^-, M; S) = \mathsf{hd}(B^-, S) \\ \mathsf{hd}(B_1^- \mathbin{\&} B_2^-, \pi_1; S) = \mathsf{hd}(B_1^-, S) \\ \mathsf{hd}(B_1^- \mathbin{\&} B_2^-, \pi_2; S) = \mathsf{hd}(B_2^-, S) \\ \mathsf{hd}(B^-, ()) = B^- \end{array}$$

The complete definition of hereditary substitution is given in Figures 5.3 and 5.4. The side-condition that $H \neq x$ should be read as $H = y \neq x$ or $H = c$, as section 5.5 below will expand the syntax of heads. In Figure 5.4 F denotes an arbitrary syntactic class. Note, that the mutually recursive hereditary substitution functions are terminating by a lexicographic ordering of the type label and the term.

$$\mathsf{type}[N/x]_{B^-} = \mathsf{type}$$
$$(\Pi y{:}A^-.\,K)[N/x]_{B^-} = \Pi y{:}A^-[N/x]_{B^-}.\,K[N/x]_{B^-}$$

$$(\widehat{\Pi} p{:}A^+.\,A^-)[N/x]_{B^-} = \widehat{\Pi} p{:}A^+[N/x]_{B^-}.\,A^-[N/x]_{B^-}$$
$$(A_1^- \mathbin{\&} A_2^-)[N/x]_{B^-} = A_1^-[N/x]_{B^-} \mathbin{\&} A_2^-[N/x]_{B^-}$$
$$\{A^+\}[N/x]_{B^-} = \{A^+[N/x]_{B^-}\}$$
$$(a \cdot T)[N/x]_{B^-} = a \cdot T[N/x]_{B^-}$$

$$(\widehat{\exists} p{:}A_1^+.\,A_2^+)[N/x]_{B^-} = \widehat{\exists} p{:}A_1^+[N/x]_{B^-}.\,A_2^+[N/x]_{B^-}$$
$$(\downarrow A^-)[N/x]_{B^-} = \downarrow A^-[N/x]_{B^-}$$
$$(!A^-)[N/x]_{B^-} = !A^-[N/x]_{B^-}$$
$$(@A^-)[N/x]_{B^-} = @A^-[N/x]_{B^-}$$
$$\mathbf{1}[N/x]_{B^-} = \mathbf{1}$$

$$(N_1;T)[N_2/x]_{B^-} = N_1[N_2/x]_{B^-};T[N_2/x]_{B^-}$$
$$()[N/x]_{B^-} = ()$$

$$\langle\!\langle M_1, M_2\rangle\!\rangle[N/x]_{B^-} = \langle\!\langle M_1[N/x]_{B^-}, M_2[N/x]_{B^-}\rangle\!\rangle$$
$$(\downarrow N_1)[N_2/x]_{B^-} = \downarrow N_1[N_2/x]_{B^-}$$
$$(!N_1)[N_2/x]_{B^-} = !N_1[N_2/x]_{B^-}$$
$$(@N_1)[N_2/x]_{B^-} = @N_1[N_2/x]_{B^-}$$
$$\mathbf{1}[N/x]_{B^-} = \mathbf{1}$$

$$(\widehat{\lambda} p.\,N_1)[N_2/x]_{B^-} = \widehat{\lambda} p.\,N_1[N_2/x]_{B^-}$$
$$\langle N_1, N_2\rangle[N_3/x]_{B^-} = \langle N_1[N_3/x]_{B^-}, N_2[N_3/x]_{B^-}\rangle$$
$$\{E\}[N/x]_{B^-} = \{E[N/x]_{B^-}\}$$
$$(H \cdot S)[N/x]_{B^-} = H \cdot S[N/x]_{B^-} \quad \text{if } H \neq x$$
$$(x \cdot S)[N/x]_{B^-} = \mathsf{Redex}_{B^-}(N \cdot S[N/x]_{B^-})$$

$$(M;S)[N/x]_{B^-} = M[N/x]_{B^-};S[N/x]_{B^-}$$
$$(\pi_1;S)[N/x]_{B^-} = \pi_1;S[N/x]_{B^-}$$
$$(\pi_2;S)[N/x]_{B^-} = \pi_2;S[N/x]_{B^-}$$
$$()[N/x]_{B^-} = ()$$

$$(\mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ E)[N/x]_{B^-} = \mathsf{let}\ \{p\} = H \cdot S[N/x]_{B^-}\ \mathsf{in}\ E[N/x]_{B^-}$$
$$\quad \text{if } H \neq x$$
$$(\mathsf{let}\ \{p\} = x \cdot S\ \mathsf{in}\ E)[N/x]_{B^-} = \mathsf{MRedex}_{B^+}(\{p\} = N'\ \mathsf{in}\ E[N/x]_{B^-})$$
$$\quad \text{where } N' = \mathsf{Redex}_{B^-}(N \cdot S[N/x]_{B^-}) \text{ and } \{B^+\} = \mathsf{hd}(B^-, S)$$

Figure 5.3: Hereditary substitution

$$
\begin{aligned}
&F[\langle\!\langle M_1, M_2\rangle\!\rangle / \langle\!\langle p_1, p_2\rangle\!\rangle]_{B_1^+ \otimes B_2^+} = F[M_1/p_1]_{B_1^+}[M_2/p_2]_{B_2^+}\\
&F[\downarrow N/\downarrow x]_{\downarrow B^-} = F[N/x]_{B^-}\\
&F[!N/!x]_{!B^-} = F[N/x]_{B^-}\\
&F[@N/@x]_{@B^-} = F[N/x]_{B^-}\\
&F[\mathbf{1}/\mathbf{1}]_{\mathbf{1}} = F
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Redex}_{B^+ \multimap B^-}(\widehat{\lambda} p.\, N \cdot M; S) = \mathsf{Redex}_{B^-}(N[M/p]_{B^+} \cdot S)\\
&\mathsf{Redex}_{B_1^- \& B_2^-}(\langle N_1, N_2\rangle \cdot \pi_1; S) = \mathsf{Redex}_{B_1^-}(N_1 \cdot S)\\
&\mathsf{Redex}_{B_1^- \& B_2^-}(\langle N_1, N_2\rangle \cdot \pi_2; S) = \mathsf{Redex}_{B_2^-}(N_2 \cdot S)\\
&\mathsf{Redex}_{B^-}(N \cdot ()) = N
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{MRedex}_{B^+}(\{p_1\} = \{\mathsf{let}\ \{p_2\} = H \cdot S\ \mathsf{in}\ E_1\}\ \mathsf{in}\ E_2) =\\
&\qquad \mathsf{let}\ \{p_2\} = H \cdot S\ \mathsf{in}\ \mathsf{MRedex}_{B^+}(\{p_1\} = \{E_1\}\ \mathsf{in}\ E_2)\\
&\mathsf{MRedex}_{B^+}(\{p\} = \{M\}\ \mathsf{in}\ E) = E[M/p]_{B^+}
\end{aligned}
$$

Figure 5.4: Hereditary substitution

$$
\begin{aligned}
&((N_1 : A^-) \cdot S)[N_2/x]_{B^-} = (N_1[N_2/x]_{B^-} : A^-[N_2/x]_{B^-}) \cdot S[N_2/x]_{B^-}\\
&(\mathsf{let}\ \{p\} = (N_1 : A^-) \cdot S\ \mathsf{in}\ E)[N_2/x]_{B^-} =\\
&\qquad \mathsf{let}\ \{p\} = (N_1[N_2/x]_{B^-} : A^-[N_2/x]_{B^-}) \cdot S[N_2/x]_{B^-}\ \mathsf{in}\ E[N_2/x]_{B^-}
\end{aligned}
$$

Figure 5.5: Hereditary substitution with redices

5.5 CLF with logic variables and redices

A calculus of canonical forms and hereditary substitutions as we have presented in section 5.4 above is quite nice to work with from a theoretical perspective. But when it comes to an actual implementation, it is nice to have a bit more flexibility.

5.5.1 Redices

We can add explicit redices by allowing normal objects to occur as heads:

$$H ::= \cdots \mid N : A^-$$

Notice that the explicit redex $(N : A^-) \cdot S$ corresponds to the hereditarily evaluated implicit redex $\mathsf{Redex}_{\lfloor A^- \rfloor}(N \cdot S)$.

We have added an explicit type ascription, since this allows us to incorporate type checking of redices directly in the bidirectional type checking algorithm. The typing rule is as follows:

$$
\frac{\Gamma \vdash_\Sigma A_1^- : \mathsf{type} \quad \Gamma; \Phi_1; \Delta_1 \vdash_\Sigma N \Leftarrow A_1^- \quad \Gamma; \Phi_2; \Delta_2; [A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; (\Phi_1, \Phi_2); (\Delta_1, \Delta_2); [N : A_1^-] \vdash_\Sigma S \Rightarrow A_2^-}
$$

$$\begin{aligned}&\mathsf{Redex}_{B^-}((H \cdot S_1) \cdot S_2) = H \cdot (S_1 +\!\!+\, S_2)\\&\mathsf{MRedex}_{B^+}(\{p\} = H \cdot S \ \mathsf{in}\ E) = \mathsf{let}\ \{p\} = H \cdot S \ \mathsf{in}\ E\end{aligned}$$

Figure 5.6: Hereditary substitution without the η-long restriction

$$\begin{aligned}&(M; S_1) +\!\!+\, S_2 = M; (S_1 +\!\!+\, S_2)\\&(\pi_1; S_1) +\!\!+\, S_2 = \pi_1; (S_1 +\!\!+\, S_2)\\&(\pi_2; S_1) +\!\!+\, S_2 = \pi_2; (S_1 +\!\!+\, S_2)\\&() +\!\!+\, S = S\end{aligned}$$

Figure 5.7: Spine concatenation

It is important to notice that explicit redices and hereditarily evaluated implicit redices are not at odds with each other; they can easily coexist in the same type system. The required extension of the hereditary substitution definition is shown in Figure 5.5.

Explicit redices relax the β-normal requirement of canonical forms. We can also relax the requirement of terms being η-long by relaxing the typing rule of $H \cdot S$ to allow arbitrary types:

$$\frac{\Gamma; \Phi; \Delta; [H] \vdash_\Sigma S \Rightarrow A^{-\prime} \quad A^- \equiv A^{-\prime}}{\Gamma; \Phi; \Delta \vdash_\Sigma H \cdot S \Leftarrow A^-}$$

The additional cases for hereditary substitution is shown in Figure 5.6; $S_1 +\!\!+\, S_2$ denotes the concatenation of two spines and is defined in Figure 5.7.

Now we can express the equational theory directly as a collection of equations. The equations for β correspond directly to the definition of hereditary substitution.

We write $\overline{p}$ for the monadic object that is syntactically equal to the pattern p.

The rules of course have all the usual side-conditions about variables and their scope.

Let-floating equality

$$\begin{aligned}&\mathsf{let}\ \{p_1\} = H_1 \cdot S_1 \ \mathsf{in}\ \mathsf{let}\ \{p_2\} = H_2 \cdot S_2 \ \mathsf{in}\ E \equiv\\&\mathsf{let}\ \{p_2\} = H_2 \cdot S_2 \ \mathsf{in}\ \mathsf{let}\ \{p_1\} = H_1 \cdot S_1 \ \mathsf{in}\ E\end{aligned}$$

β-equality

$$(\widehat{\lambda} p.\, N : \widehat{\Pi} p{:}A^+.\, A^-) \cdot (M; S) \equiv (N[M/p]_{\lfloor A^+ \rfloor} : A^-[M/p]_{\lfloor A^+ \rfloor}) \cdot S$$

$$(N : A^-) \cdot () \equiv N$$

$$(\langle N_1, N_2 \rangle : A_1^- \ \&\ A_2^-) \cdot (\pi_1; S) \equiv (N_1 : A_1^-) \cdot S$$

$$(\langle N_1, N_2 \rangle : A_1^- \ \&\ A_2^-) \cdot (\pi_2; S) \equiv (N_2 : A_2^-) \cdot S$$

$$\begin{array}{l}
(X[s] \cdot S)[N/x]_{B^-} = X[s[N/x]_{B^-}] \cdot S[N/x]_{B^-} \\
(\mathsf{let}\ \{p\} = X[s] \cdot S\ \mathsf{in}\ E)[N/x]_{B^-} = \\
\qquad \mathsf{let}\ \{p\} = X[s[N/x]_{B^-}] \cdot S[N/x]_{B^-}\ \mathsf{in}\ E[N/x]_{B^-} \\
\cdot\, [N/x]_{B^-} = \cdot \\
(N_1/y^f, s)[N_2/x]_{B^-} = N_1[N_2/x]_{B^-}/y^f, s[N_2/x]_{B^-} \text{ where } f \in \{\mathbf{I}, \mathbf{A}, \mathbf{L}\}
\end{array}$$

Figure 5.8: Hereditary substitution with logic variables

$$(H \cdot S_1 : A^-) \cdot S_2 \equiv H \cdot (S_1 +\!\!+\, S_2)$$

$$\mathsf{let}\ \{p_1\} = \{\mathsf{let}\ \{p_2\} = H \cdot S\ \mathsf{in}\ E_1\} : \{A^+\} \cdot ()\ \mathsf{in}\ E_2 \equiv$$

$$\mathsf{let}\ \{p_2\} = H \cdot S\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = \{E_1\} : \{A^+\} \cdot ()\ \mathsf{in}\ E_2$$

$$\mathsf{let}\ \{p\} = \{M\} : \{A^+\} \cdot ()\ \mathsf{in}\ E \equiv E[M/p]_{\lfloor A^+ \rfloor}$$

η-equality

$$H \cdot S \equiv \widehat{\lambda} p.\, H \cdot (S +\!\!+\, \overline{p}; ())$$

$$H \cdot S \equiv \langle H \cdot (S +\!\!+\, \pi_1; ()), H \cdot (S +\!\!+\, \pi_2; ()) \rangle$$

$$H \cdot S \equiv \{\mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ \overline{p}\}$$

5.5.2 Logic variables

Logic variables are also added by extending the syntax of heads:

$$H ::= \cdots \mid X[s]$$

Logic variables have an associated substitution, which is just a list of single-variable substitutions for intuitionistic, affine, and linear variables:

$$s ::= \cdot \mid N/x^{\mathbf{I}}, s \mid N/x^{\mathbf{A}}, s \mid N/x^{\mathbf{L}}, s$$

A substitution can be considered unordered, except for the intuitionistic parts, which must maintain their relative ordering. This is because the corresponding intuitionistic context is also ordered.

The extension of hereditary substitution to CLF with logic variables is shown in Figure 5.8.

We write $\overline{s}$ for the intuitionistic part of s:

$$\overline{\cdot} = \cdot \qquad \overline{N/x^{\mathbf{I}}, s} = N/x^{\mathbf{I}}, \overline{s} \qquad \overline{N/x^{\mathbf{A}}, s} = \overline{s} \qquad \overline{N/x^{\mathbf{L}}, s} = \overline{s}$$

Erasure $\lfloor \cdot \rfloor$ is extended to contexts by pointwise application.

Let once again F denote an arbitrary syntactic class. The hereditary substitution of all the variables in a substitution s for their occurrences in F is

$$\frac{X :: A_1^- \text{ in } (\Gamma_X; \Phi_X; \Delta_X) \in \Psi \qquad \Gamma; \Phi_1; \Delta_1 \vdash_\Sigma s \Leftarrow \Gamma_X; \Phi_X; \Delta_X \quad \Gamma; \Phi_2; \Delta_2; [A_1^-[\overline{s}]_{\lfloor \Gamma_X \rfloor; \cdot; \cdot}] \vdash_\Sigma S \Rightarrow A_2^-}{\Gamma; (\Phi_1, \Phi_2); (\Delta_1, \Delta_2); [X[s]] \vdash_\Sigma S \Rightarrow A_2^-}$$

$$\frac{x \notin \mathrm{FV}(\Phi') \cup \mathrm{FV}(\Delta') \quad \Gamma; \cdot; \cdot \vdash_\Sigma N \Leftarrow A^-[\overline{s}]_{\lfloor \Gamma' \rfloor; \cdot; \cdot} \quad \Gamma; \Phi; \Delta \vdash_\Sigma s \Leftarrow \Gamma'; \Phi'; \Delta'}{\Gamma; \Phi; \Delta \vdash_\Sigma N/x^{\mathbf{I}}, s \Leftarrow (\Gamma', x{:}A^-); \Phi'; \Delta'}$$

$$\frac{\Gamma; \Phi_1; \cdot \vdash_\Sigma N \Leftarrow A^-[\overline{s}]_{\lfloor \Gamma' \rfloor; \cdot; \cdot} \quad \Gamma; \Phi_2; \Delta \vdash_\Sigma s \Leftarrow \Gamma'; \Phi'; \Delta'}{\Gamma; (\Phi_1; \Phi_2); \Delta \vdash_\Sigma N/x^{\mathbf{A}}, s \Leftarrow \Gamma'; (\Phi', x{:}A^-); \Delta'}$$

$$\frac{\Gamma; \Phi_1; \Delta_1 \vdash_\Sigma N \Leftarrow A^-[\overline{s}]_{\lfloor \Gamma' \rfloor; \cdot; \cdot} \quad \Gamma; \Phi_2; \Delta_2 \vdash_\Sigma s \Leftarrow \Gamma'; \Phi'; \Delta'}{\Gamma; (\Phi_1; \Phi_2); (\Delta_1; \Delta_2) \vdash_\Sigma N/x^{\mathbf{L}}, s \Leftarrow \Gamma'; \Phi'; (\Delta', x{:}A^-)}$$

$$\frac{}{\Gamma; \Phi; \cdot \vdash_\Sigma \cdot \Leftarrow \cdot; \cdot; \cdot}$$

Figure 5.9: Logic variable and substitution typing

written $F[s]_{\Gamma;\Phi;\Delta}$[2] and defined as follows:

$$\begin{aligned}
&F[\cdot]_{\cdot;\cdot;\cdot} = F \\
&F[N/x^{\mathbf{I}}, s]_{(\Gamma, x:B^-);\Phi;\Delta} = F[N/x]_{B^-}[s]_{\Gamma;\Phi;\Delta} \\
&F[N/x^{\mathbf{A}}, s]_{\Gamma;(\Phi, x:B^-);\Delta} = F[N/x]_{B^-}[s]_{\Gamma;\Phi;\Delta} \\
&F[N/x^{\mathbf{L}}, s]_{\Gamma;\Phi;(\Delta, x:B^-)} = F[N/x]_{B^-}[s]_{\Gamma;\Phi;\Delta}
\end{aligned}$$

Each logic variable has a specific type and makes sense in a specific set of contexts all of which is given by the contextual modal context Ψ [NPP08]:

$$\Psi ::= \cdot \mid \Psi, (X :: A^- \text{ in } (\Gamma; \Phi; \Delta))$$

The contextual modal context Ψ could be added to all the typing judgments, but we choose to keep it implicit, as it remains constant throughout all the rules. Instead we simply write $X :: A^- \text{ in } (\Gamma; \Phi; \Delta) \in \Psi$ for the lookup of X in Ψ.

We assume that Ψ is valid, i.e. that for any $X :: A^- \text{ in } (\Gamma; \Phi; \Delta) \in \Psi$ the context triple $\Gamma; \Phi; \Delta$ is valid in the sense specified in the Context Validity Assumption and that $\Gamma \vdash_\Sigma A^- : \mathsf{type}$ holds.

Hereditary instantiation of a logic variable $X :: A^- \text{ in } (\Gamma; \Phi; \Delta)$ to a normal object N with $\Gamma; \Phi; \Delta \vdash_\Sigma N \Leftarrow A^-$ can be done by replacing all occurrences of $X[s] \cdot S$ with $\mathsf{Redex}_{\lfloor A^- \rfloor}(N[s]_{\lfloor \Gamma \rfloor; \lfloor \Phi \rfloor; \lfloor \Delta \rfloor} \cdot S)$.

The typing rules for logic variables and substitutions are given in Figure 5.9. They are the same as we presented them in chapter 3, except that they have been extended to dependent types and written in named form.

The requirement that x cannot be free in the types occurring in Φ' and Δ' in the typing of intuitionistic substitution extensions can easily be fulfilled, e.g. by requiring that affine and linear extensions are typed first, and thus Φ' and Δ' would be empty and the requirement $x \notin \mathrm{FV}(\Phi') \cup \mathrm{FV}(\Delta')$ vacuously true. If we are using de Bruijn indices the requirement also becomes trivially true, since

[2] The context triple in $[s]_{\Gamma;\Phi;\Delta}$ is the erased type of s just as B^- is the erased type of N in $[N/x]_{B^-}$.

we would have a total ordering on the variables in context triples consistent with the dependencies and the order of the substitution extensions.

Chapter 6

Celf — an Implementation of CLF

6.1 Introduction

Celf [SNS08, Clf] is to CLF [CPWW02a] what Elf [Pfe91] is to LF [HHP93], hence the name. That is, Celf is an implementation of the CLF type theory with a logic programming-based operational semantics. Many of the design choices in Celf come from Twelf [PS99] and we will make several references to the similarities below.

The system description of Celf [SNS08] describes version 1.x, which implemented the original CLF type theory. As we described in chapter 5, we have redesigned the type theory and extended it with affine types. The current Celf is at the time of writing version 2.6.

In this chapter we will describe the interface and the implementation along with the various theoretical contributions that support the implementation. The goal is to allow other researchers a sufficient understanding of the implementation to make it possible to extend the Celf implementation with additional capabilities, such as support for meta-theoretical reasoning.

Celf is implemented in Standard ML, and the source code is available for download from `http://www.twelf.org/~celf`.

6.2 Core functionality

At its core Celf answers the following two questions:

1. Given a signature Σ, is it the case that $\vdash \Sigma : \mathsf{sig}$?

2. Given a type A^- mentioning perhaps some number of logic variables, does there exist an instantiation θ of the logic variables and a term N such that $\cdot; \cdot; \cdot \vdash_\Sigma N \Leftarrow \theta(A^-)$?

The latter question is given a partial answer in the following sense: The proof search algorithm is a logic programming interpreter treating the signature Σ as a logic program and the type A^- as a query in that program. The semantics is based on Lollimon [LPPW05], an extension of Lolli [HM94], the linear sibling of λ-Prolog.

6.3 Using Celf

6.3.1 Commandline

Celf is run from the commandline in the following way:

```
celf <options> <filename>
```

The commandline option `-h` prints the available commandline options and exits. Otherwise Celf reads the specified file, parses it as a CLF signature, and executes the given queries.

6.3.2 Grammar

The grammar is as follows:

$$\begin{aligned}
\mathit{Program} &::= \mathit{Decl} \mid \mathit{Decl}\ \mathit{Program}\\
\mathit{Decl} &::= \mathit{TypeConst} \mid \mathit{TermConst} \mid \mathit{TypeAbbrev} \mid \mathit{TermAbbrev} \mid \mathit{Query}\\
\mathit{TypeConst} &::= \mathit{Iden}\ \texttt{:}\ \mathit{Kind}\texttt{.}\\
\mathit{TermConst} &::= \mathit{Iden}\ \texttt{:}\ \mathit{NegType}\texttt{.}\\
\mathit{TypeAbbrev} &::= \mathit{Iden}\ \texttt{:}\ \texttt{type}\ \texttt{=}\ \mathit{NegType}\texttt{.}\\
\mathit{TermAbbrev} &::= \mathit{Iden}\ \texttt{:}\ \mathit{NegType}\ \texttt{=}\ \mathit{Term}\texttt{.}\\
\mathit{Query} &::= \texttt{\#query}\ \mathit{NumOpt}\ \mathit{NumOpt}\ \mathit{NumOpt}\ \mathit{Number}\ \mathit{NegType}\texttt{.}\\
\mathit{NumOpt} &::= \texttt{*} \mid \mathit{Number}
\end{aligned}$$

An identifier *Iden* is any non-empty sequence of letters, digits, and/or any of the characters “`-<>=/|_'*#+&~;$?`” excluding a few keywords and symbols with special meaning. Also, numbers are not identifiers. The distinguished keywords and special symbols are “`type`”, “`Pi`”, “`PI`”, “`#1`”, “`#2`”, “`Exists`”, “`EXISTS`”, “`let`”, “`in`”, “`#query`”, “`-o`”, “`o-`”, “`:`”, “`.`”, “`_`”, “`&`”, “`{`”, “`}`”, “`*`”, “`\`”, “`<`”, “`>`”, “`,`”, “`=`”, “`[`”, “`]`”, “`(`”, “`)`”, “`->`”, “`<-`”, “`@`”, “`!`”, “`-@`”, and “`@-`”.

Identifiers starting with an uppercase letter are treated specially and should therefore not be used as signature constants or abbreviations (see section 6.3.4 below).

Anything following a “`%`” is treated as a comment and ignored.

The syntax of kinds, types, and terms is that of the reworked CLF type theory (see section 5.4). We show the ASCII encoding in Table 6.1.

6.3.3 Abbreviations and redices

Besides queries and signature declarations, Celf also supports type and term abbreviations. These are simply expanded whenever they are encountered. This can potentially introduce redices in terms, so as a consequence Celf supports general redices and thus does not require terms to be in canonical form. The extension to the type theory is described in section 5.5.1.

CLF	Celf
Π	`Pi`
$\widehat{\Pi}$	`PI`
$t_1 \to t_2$	t_1 `->` t_2 or t_2 `<-` t_1
$t_1 \mathbin{-\!\!@}\, t_2$	t_1 `-@` t_2 or t_2 `@-` t_1
$t_1 \multimap t_2$	t_1 `-o` t_2 or t_2 `o-` t_1
$\exists$	`Exists`
$\widehat{\exists}$	`EXISTS`
$\otimes$	`*`
$t \cdot t_1; \ldots; t_n; ()$	$t\ t_1\ \ldots\ t_n$
$\downarrow t$	t
$\langle\!\langle t_1, t_2 \rangle\!\rangle$	`[`t_1`,` t_2`]`
$\widehat{\lambda}$	`\`
π_1	`#1`
π_2	`#2`

Table 6.1: Syntax of CLF vs. Celf

6.3.4 Implicit parameters and type inference

As it is also the case in Twelf, Celf supports implicit parameters [Pie10]. In type and term declarations any freely occurring names starting with uppercase letters are treated as being implicitly Π-quantified. When checking a declaration Celf will infer the type of all such parameters, and when using a type or term constant Celf will automatically infer and apply implicit arguments to the constant corresponding to the implicit Πs.

This design is known from Twelf to be very practical and reduce the amount of typing immensely.

Freely occurring names starting with uppercase letters in queries are treated differently. In queries such names stand for logic variables to which Celf should find the most general instantiation during the proof search. This is also consistent with the design of Twelf.

Celf also allows binders with or without type ascriptions; if the ascriptions are missing Celf will infer the type. Additionally, terms can also be inferred; a special term "`_`" (underscore[1]) is allowed, which will be automatically filled in by unification during type inference.

If the types are sufficiently underspecified the inference algorithm will of course fail. In this case the user can supply additional type information at arbitrary positions in terms with the following type ascription construct:

$$\mathit{Term} ::= \cdots \mid \texttt{(}\ \mathit{Term} : \mathit{NegType}\ \texttt{)}$$

6.3.5 Modality inference

Consider a simple signature for the representation of natural numbers:

```
nat : type.
z : nat.
```

[1] Not to be confused with _ (undefined) introduced below in section 6.4.6.

```
s : nat -> nat.
```

With this signature the natural number 3 would be represented as

```
s !(s !(s !z))
```

since nat → nat is equal to !nat ⊸ nat, whereas in Twelf and LF it would be written as

```
s (s (s z))
```

This can be a nuisance when porting Twelf signatures to Celf, but since the type of the constructor s is known we can easily infer that the intuitionistic modalities should be there. Therefore Celf allows the user to omit intuitionistic and affine modalities in applications whenever the head of the term is a signature-defined constant, in which case Celf will infer the missing modalities and insert them.

6.3.6 Queries

In Twelf a query is supplied two optional numbers; the expected number of solutions and the number of solutions to look for. In Celf a query has four numerical arguments, three of which are optional.

The first argument gives a crude control over the forward-chaining proof search by setting an upper limit on the number of consecutive such steps. If a * is given then no bound is imposed.

The second argument is the expected number of solutions to the query. If the number of solutions found differs from the given number Celf will report that the query failed and terminate. If a * is given no such check is made.

The third argument sets a limit on the number of solutions to look for and will stop the search when the specified number is found. If a * is given Celf will instead search for all solutions.

The fourth argument is the number of times to execute the query. As the forward-chaining involves non-deterministic committed choices running a query multiple times can lead to different results. Celf will execute the query the given number of times or, if the second argument is given, stop when the specified expected number of solutions is found. In this case the query will only fail if the expected number of solutions cannot be found in any of the runs.

Running Celf with the commandline option `-hquery` will print a summary of the four query arguments and exit.

6.3.7 Double checking

The type checking part of Celf is fairly complicated; it involves type inference, higher-order unification for linear and affine types including linearity pruning, implicit parameter inference, normalization, and implicit Π-quantification.

In order to allow the user a higher level of trust in the correctness of this procedure a double checking feature is supported similar to how double checking in Twelf works. When a declaration is completely normalized and every unspecified type is inferred then the simple bidirectional type checking algorithm presented in section 5.4 can be used to check the declaration. This algorithm is implemented separately in Celf without any reference to e.g. higher-order unification. If Celf is run with the commandline option `-d` every declaration is double checked in this way.

6.4 Implementation

Celf is a complicated piece of software, and its implementation draws on many different theoretical contributions. In this section we present some of the theoretical background specific to the implementation, which is not already present in chapters 2 through 5.

6.4.1 Resource management

The type system specified in section 5.4 and 5.5 includes non-deterministic splits of both the linear and the affine context. In order to implement this efficiently, both for type checking, type inference, and proof search, a linear resource management system is used. This means that the judgments are transformed to use an input and an output context in order to allow lazy context splitting.

For e.g. normal objects we write $\Gamma; (\Phi_I \triangleright \Phi_O); (\Delta_I \triangleright \Delta_O) \vdash^{IO}_{\Sigma} N \Leftarrow A^-$ where Φ_I and Δ_I are given and Φ_O and Δ_O are synthesized as the subsets of Φ_I and Δ_I, respectively, that do not occur in N. Thus, we have that this judgment is equivalent to $\Gamma; (\Phi_I - \Phi'_O); (\Delta_I - \Delta_O) \vdash_{\Sigma} N \Leftarrow A^-$ where $\Phi'_O \subseteq \Phi_O$.

With this setup, the context splits can be determined lazily by threading the contexts through the premises of the typing rules. The rule for pairs, which include a context split, e.g. becomes:

$$\dfrac{\begin{array}{l}\Gamma; (\Phi_I \triangleright \Phi_M); (\Delta_I \triangleright \Delta_M) \vdash^{IO}_{\Sigma} M_1 \Leftarrow A_1^+ \\ \qquad\qquad\qquad \Gamma; (\Phi_M \triangleright \Phi_O); (\Delta_M \triangleright \Delta_O) \vdash^{IO}_{\Sigma} M_2 \Leftarrow A_2^+[M_1/p]_{\lfloor A_1^+ \rfloor}\end{array}}{\Gamma; (\Phi_I \triangleright \Phi_O); (\Delta_I \triangleright \Delta_O) \vdash^{IO}_{\Sigma} \langle\!\langle M_1, M_2 \rangle\!\rangle \Leftarrow \widehat{\exists} p{:}A_1^+.\, A_2^+}$$

A detailed account of linear resource management can be found in [CHP00] and the extension to affine types is straightforward.

6.4.2 Explicit substitutions through recursion schemes

Chapter 3 introduced the explicit substitution calculus that is the main data structure in the implementation of Celf. A natural comparison is the implementation of Twelf, which is based on $\lambda\sigma$. In Twelf closures are all over the source code; every single place an object is manipulated it occurs as a closure with an associated substitution. This has been the source of several hard-to-find bugs in Twelf — usually in the form of a forgotten change from s to $1 \,.\, s \circ \uparrow$ when going beneath a binder.

In Celf Wang and Murphy's recursion schemes [WM02] are used — among other things — to isolate the treatment of explicit substitutions to a single module. This means that the rest of the code can treat closures and substitutions as if they were being eagerly and hereditarily evaluated, while in reality they are evaluated lazily allowing maximal composition of substitutions in nested closures. This leaves evaluation order unspecified in terms of whether substitutions are composed or not, but this is sound due to our strong normalization result (chapter 2).

This approach has other benefits as well, such as automatically generated maps and folds for all the syntax datatypes and support for different views on the syntax.

```
signature TYP = sig
  (* abstract type t *)
  type t
  (* public datatype 't F that reveals the
     constructors of the type t *)
  type 't F
  (* isomorphism t = t F *)
  val inj : t F -> t
  val prj : t -> t F
  (* Fmap (fn x => x) == (fn x => x)
     Fmap (f o g) == (Fmap f) o (Fmap g) *)
  val Fmap : ('t1 -> 't2) -> 't1 F -> 't2 F
end
```

Figure 6.1: Abstract recursion signature

The basic idea is to have an abstract type `t` and a public datatype `'t F`, which is the idealized representation of the abstract type `t` except all recursive instances of `t` have been replaced by a type variable `'t`. This means that the type `t` is isomorphic to `t F`. Figure 6.1 shows the basic ML signature where `inj` and `prj` represents the isomorphism. The function `Fmap` corresponds to the action of `F` on morphisms when viewing `F` as a categorical functor.[2]

From this signature we can e.g. define a general fold function in the following way:

```
fun fold f x = f (Fmap (fold f) (prj x))
```

The type of `fold` is `('a F -> 'a) -> t -> 'a`.

Looking at the implementation of Celf, we can consider, e.g., the representation of spines:

```
datatype 'sp spineF
     = Nil
     | LApp of monadObj * 'sp
     | ProjLeft of 'sp
     | ProjRight of 'sp
```

This definition implements `TYP` with `t` = `spine` and `F` = `spineF` where `spine` is an abstract type. Conceptually, we can think of `spine`, `inj`, and `prj` as being defined in the following way:

```
datatype spine = FixSpine of spine spineF
fun inj s = FixSpine s
fun prj (FixSpine s) = s
```

The main parts of the codebase can now work with spines using pattern matching on the constructors defined by `spineF` by means of `prj` and `inj` or by using the function `fold` defined above.

However, the underlying definition of the abstract type `spine` is really:

[2] In general, concrete polymorphic ML datatypes built from sums and products can naturally be viewed as categorical functors $\mathcal{T}^n \to \mathcal{T}$, where $\mathcal{T}$ is the category of ML types and n is the number of free type variables in the datatype declaration.

```
signature TYP2 = sig
  type a
  type t
  type ('a, 't) F
  val inj : (a, t) F -> t
  val prj : t -> (a, t) F
  val Fmap : ('a1 -> 'a2) * ('t1 -> 't2)
      -> ('a1, 't1) F -> ('a2, 't2) F
end
```

Figure 6.2: Abstract recursion signature with one additional type parameter

```
datatype spine
    = FixSpine of spine spineF
    | SClos of spine * subst
```

The additional constructor SClos represents a delayed hereditary substitution, which is never seen directly, as it is handled by the implementation of prj.

This scheme allows us to implement lazily evaluated hereditary substitutions that allow maximal composition, and still confine the handling of substitutions to a single module.

Further benefits come from the fact that the spineF datatype is simply one view on the underlying datastructure, and we can easily define several others. The Celf implementation, e.g., has a canonical forms view, which automatically reduces any redices written by the user by means of lazily evaluated hereditary substitutions.

Taking this one step further (and beyond what is presented as the general scheme in [WM02]), we generalize the signature TYP to mention whatever other types t contain. The ML signature with one additional type is given in Figure 6.2.[3] For the type spine we can implement this generalized signature with a = monadObj, t = spine, and F = spineFF where spineFF is given by:

```
datatype ('m, 'sp) spineFF
    = Nil
    | LApp of 'm * 'sp
    | ProjLeft of 'sp
    | ProjRight of 'sp
```

This generalized signature TYP2 allows us to define a general map function that traverses the structure of t and applies a function to every a:

```
fun map f x = inj (Fmap (f, map f) (prj x))
```

The type of map is (a -> a) -> t -> t. In the case of spine the function map gets the type (monadObj -> monadObj) -> spine -> spine and thus applies a given function to every monadic object in the spine.

[3] The Celf source code also defines signatures TYP3 and TYP4, which, similarly to TYP2, generalize TYP to an F with three and four type variables, respectively.

6.4.3 Contexts

In chapter 5 we presented the CLF type theory with a triple of contexts, two of which were unordered in order to give a more flexible presentation. In chapters 3 and 4 we instead used a single context corresponding to the triple of contexts with an additional total order on all the variables. In the implementation we use de Bruijn indices and thus the latter representation.

To store the context we use Chris Okasaki's purely functional random-access lists [Oka95], which improves lookup and update of the ith element in an n-element list to $O(\min\{i, \log n\})$ while keeping head, cons, and tail $O(1)$.

6.4.4 Type inference

In order to do general type inference, Celf splits the process in two stages.

First a Hindley-Milner inference algorithm is applied to obtain approximate types, i.e. simple types, for every term. This relies on unification of simple types and is well understood.

Then the full bidirectional type checking algorithm is used to check dependent types. This relies on unification of dependent types, but since the simple types are known at this point it reduces to unification of terms, which we will describe below in section 6.4.6.

This structuring is similar to the Twelf implementation.

6.4.5 Proof search

The implementation of the proof search algorithm is structured around a *success continuation* with backtracking being the default case. At each choice point every choice is tried and then backtracked. Solutions are considered the exceptional case and are not collected explicitly; instead, whenever a solution is found the success continuation is applied to it before backtracking.

The proof search also employs a slight extension of the resource management system for the linear context. An extra must-occur context is carried around as a subcontext of the linear context. This additional context specifies that the mentioned variables cannot occur in any other position, i.e. the current position is not part of a multiplicative context split in which the other half might consume the linear resources. This allows the proof search algorithm to fail and backtrack earlier and thus improve its termination behavior. This extension is also described in [CHP00]

The specification of the proof search behavior is given in [LPPW05].

6.4.6 Unification

Unification plays a central role in both type inference and proof search. In chapter 4 we introduced a pattern unification algorithm for a type system closely related to CLF. We also gave an extension to linear-changing pattern substitutions by linearity pruning.

In order to apply this unification algorithm we first relax the pattern requirement beyond linear-changing pattern substitutions. Any unification problem outside the pattern fragment, which cannot be brought inside the pattern fragment by linear pruning, is postponed until further instantiations from other unification equations put it back in the pattern fragment. This extension to a unification constraint simplification system is called the dynamic pattern fragment and described in detail in the intuitionistic case in [Ree09].

The extension to the dynamic pattern fragment has some subtle consequences for pruning and the occurs check, which were initially overlooked in the presentation in [DHKP98], but these have subsequently been addressed and handled by Reed [Ree09].

Besides the extension to the dynamic fragment, we also need to extend the algorithm, which already covers linear and affine types, to cover the entire CLF type theory. This involves two things. First, the mere existence of the equality theory of CLF has a slight impact on the unification algorithm from chapter 4 (the intersection case changes a bit). Second, we need to compare and unify expression objects. We return to both of these issues below, but first we will consider the implementation of pruning in the dynamic pattern fragment.

Pruning by means of undefined terms

Consider the unification problem $X[s] \doteq N$ where s is a pattern substitution and N is not $X[t]$ for some t. In this case we need to perform some number of pruning steps, an occurs check, and finally compute $N[s^{-1}]$. To implement this efficiently we do all of this in a single traversal of N.

The key implementation tool is the introduction of a special term $\_$ called *undefined*, which is used in the undefined extensions of the inverse s^{-1} (see Definition 4.4.9). Thus, any $\_$ in $N[s^{-1}]$ corresponds to a variable in N not occurring in s, which therefore needs to be pruned. Now we simply form, evaluate, and traverse $N[s^{-1}]$ looking for $\_$ and X.

As explained in [Ree09] the occurs check consists of a few cases: Any occurrence of $X[t]$ where t is a (possibly linear-changing) pattern substitution can be replaced by $\_$ as it corresponds to a variable that should be pruned. Any occurrence of X in a strongly rigid position indicates that there is no possible solution, where a strongly rigid position is a position not within an argument to a logic variable, nor within an argument to an ordinary variable. Any occurrence of $X[t]$ where t is not a (linear-changing) pattern substitution in a position that is not strongly rigid means that the equation is postponed as a constraint.

If we encounter an $\_$ in a rigid position in $N[s^{-1}]$, we can fail. If we encounter a $Y[t \circ s^{-1}]$ where $Y :: A_Y$ in Γ_Y and $t \circ s^{-1}$ is a (potentially linear-changing) pattern substitution except for some occurrences of $\_$, we can form the weakening substitution $w = \mathsf{weakens}(t \circ s^{-1}; \Gamma_Y)$ that prunes all the necessary variables with the following generalization of $\mathsf{weaken}(\cdot;\cdot)$ (see Pruning in section 4.5.2) to multiple variables.

$$\begin{array}{lcll} \mathsf{weakens}(\uparrow^n; \Gamma) & = & \mathsf{id} & \\ \mathsf{weakens}(n^{f'f} \mathbin{.} s; \Gamma, A^f) & = & 1^{ff} \mathbin{.} \mathsf{weakens}(s;\Gamma) \circ \uparrow & \\ \mathsf{weakens}(\_^f \mathbin{.} s; \Gamma, A^f) & = & \mathsf{weakens}(s;\Gamma) \circ \uparrow & \text{if } f \in \{\mathbf{I}, \mathbf{A}\} \end{array}$$

Thus, all the necessary pruning is done by instantiating $Y \leftarrow Z[w]$, and if $\mathsf{weakens}(t \circ s^{-1}; \Gamma_Y)$ does not exist we have an $\_$ in a linear argument of Y and can fail.

If we encounter $Y[t \circ s^{-1}]$ where t is not a pattern substitution, we can still traverse $t \circ s^{-1}$ looking for $\_$ and X. We cannot, however, perform any pruning at this point, since for e.g. $Y[\ldots Z[\ldots \_ \ldots] \ldots]$ we do not know whether it is Y or Z which should project the $\_$ away, or if it perhaps should be neither, in case some other argument to e.g. Y can project the $\_$ away.

We represent an $\_$ occurring outside a substitution as a raised exception, since any encountered $\_$ generally needs to be propagated to the top of the term in which it is encountered. However, when traversing a term in a flexible position this is no longer the case, and we must take care not to propagate an $\_$ too far. Consider e.g. the terms $\widehat{\lambda}!x.\, c \cdot !\_; ()$ and $\widehat{\lambda}!x.\, x \cdot !\_; ()$. The first is equivalent to $\_$, whereas the second is not. The second term might e.g. eventually be applied to an argument that projects away its first argument. However, such a term might occur in a position where this is impossible, and indeed $c \cdot (\widehat{\lambda}!x.\, x \cdot !\_; ()); ()$ is equivalent to $\_$.

Generally, for $\_$ in flexible positions, we can only propagate an $\_$ occurring in a spine if the head is not a logic variable, nor a variable for which a logic variable can potentially be substituted. Also we can only propagate an $\_$ occurring in one branch of an additive pair if the pair is not the argument of a logic variable, nor the argument of a variable for which a logic variable can potentially be substituted.

If we, after our traversal of $N[s^{-1}]$, have made sure that X does not occur and that we have no $\_$ in rigid positions, we can instantiate X to $N[s^{-1}]$ even if there are still remaining occurrences of $\_$ in flexible positions. In that case we just introduce an existential constraint for every logic variable $Y[t]$ with an $\_$ nested in t. We can then later recheck these constraints to see whether the nested occurrences of $\_$ have been removed by instantiations or whether they can be pruned away.

Intersection in CLF unification

When going through the unification algorithm from chapter 4 all the reasoning carries over to CLF except for the **intersection** rules. This is because the canonical forms of CLF are unique only up to let-floating, and therefore there exists objects that are equal under a permutation of variables, e.g. the following term:

$$\begin{aligned}&\{\mathsf{let}\ \{\mathbf{1}\} = c \cdot {\downarrow}x\ \mathsf{in}\ \mathsf{let}\ \{\mathbf{1}\} = c \cdot {\downarrow}y\ \mathsf{in}\ \mathbf{1}\} \equiv\\ &\{\mathsf{let}\ \{\mathbf{1}\} = c \cdot {\downarrow}y\ \mathsf{in}\ \mathsf{let}\ \{\mathbf{1}\} = c \cdot {\downarrow}x\ \mathsf{in}\ \mathbf{1}\}\end{aligned}$$

Consider the case $X[s] \doteq X[t]$ where s and t are pattern substitutions. Even though we cannot apply **intersection** directly, we can still apply pruning repeatedly for every variable occurring in s and not in t or vice versa. It is important to notice that each pruning step $X \leftarrow Y[w]$ might enable further pruning as $w \circ s$ and $w \circ t$ have fewer variables occurring in them compared to s and t. When no more pruning is possible, we can invert one of the substitutions and simplify the equation to $Z = Z[u]$ where u is a permutation with $\Gamma_Z \vdash u : \Gamma_Z$. If u is the identity, we are done. If on the other hand u is a nontrivial permutation, we cannot express the set of most general unifiers in a finite way without additional assumptions about the signature, so in that case we will postpone it as a constraint.

In the source code we represent pattern substitutions in the following way. For every substitution extension $n^{f'f} \,.\, s$ the combination of the two linearity flags $f'f$ have the following six possibilities: **II**, **AA**, **LL**, **IL**, **IA**, and **AL**. There is, however, no point at which we need to distinguish the first three cases to make algorithmic choices. Therefore we represent the six possibilities with only four different constructors[4]:

```
datatype subMode = ID | INT4LIN | INT4AFF | AFF4LIN
```

The first constructor, `ID`, represents either **II**, **AA**, or **LL**, and the latter three constructors represent the remaining three possibilities. Additionally, we represent undefined extensions $\_^f\,.\,s$ without their linearity flag f, as it is never needed. These two representational choices simplify several parts of the code that deal with substitutions.

With this representation we also get a nice commutation lemma for pattern substitutions and linear-changing identity substitutions, due to the fact that renaming variables and changing linearity flags are mostly orthogonal operations.

Lemma 6.4.1. *Let t be an arbitrary substitution, and let s be (the Celf source code representation of) a pattern substitution with possible occurrences of $\_$. Then there exists a linear-changing identity substitution i_1 with $t = i_1 \circ s$ if and only if there exists a linear-changing identity substitution i_2 with $t = s \circ i_2$.*

Proof. The nth extension in t is either $m^{f'f}$ or $\_^f$, i.e. $n^f[t]$ is either $m^{f'}$ or $\_$. We can decompose t into a linear-changing identity substitution and a pattern substitution in either order, $t = i_1 \circ s_1 = s_2 \circ i_2$, where i_1 and i_2 are linear-changing identity substitutions and s_1 and s_2 are pattern substitutions. In the first case the nth extensions in s_1 and s_2 are $m^{f'f'}$ and m^{ff}, respectively, and in the second case they are $\_^{f'}$ and $\_^f$, respectively. In both cases the Celf source code representation is the same, since the flags match (we use the `ID` constructor) and we do not need to represent the flag on an undefined extension $\_^f$. Thus, the source code representation of s_1 and s_2 are identical. □

Considering the definition of $\mathsf{weakens}(\cdot;\cdot)$ above we see that the context argument is only used to check the well-typedness of the weakening substitution, which is equivalent to not pruning linear variables. We can easily decouple the computation of the weakening substitution and the validity check, thus writing $\mathsf{weakens}(s)$ when we postpone the check against the context.

Consider the case $X[s][i_1] \doteq X[t][i_2]$ where s and t are pattern substitutions and i_1 and i_2 are linear-changing identity substitutions. By commutation of pattern substitutions and linear-changing identity substitutions, the first pruning step is $w_1 = \mathsf{weakens}(s \circ t^{-1})$. After instantiation and commutation the second pruning step is $w_2 = \mathsf{weakens}(w_1 \circ s \circ t^{-1} \circ w_1^{-1})$. We can continue computing $w_{n+1} = \mathsf{weakens}(w_n \circ \cdots \circ w_1 \circ s \circ t^{-1} \circ w_1^{-1} \circ \cdots \circ w_n^{-1})$ until w_{n+1} becomes the identity and we have reached a fixed point $w = w_n \circ \cdots \circ w_1$. At this point we can do all the pruning in one step as $X \leftarrow Z[w]$ skipping all the intermediate logic variable instantiations. Now we are left with $Z[i] = Z[u][i']$ for a permutation u and two linear-changing identity substitutions i and i'. If, as above, u is a non-trivial permutation we have to postpone the equation, but if u is the identity, the domain and codomain of i and i' are equal, which implies $i = i'$, and we are done. Thus, we see that we can completely ignore linear-changing identity substitutions in the implementation of the unification of $X[s][i_1] \doteq X[t][i_2]$.

[4] The names `INT4LIN`, etc. are chosen as they signify substituting an *int*uitionistic variable *for* a *lin*ear variable, etc.

Unifying CLF expression objects

Pushing a substitution s under a single-variable binder changes s to $1^{ff} \,.\, s \circ \uparrow$. When we work with patterns, which potentially bind many variables, we need to apply this repeatedly. Let $\#p$ denote the number of binders in a pattern p and define $\mathcal{D}_n(s)$ as follows:[5]

$$\mathcal{D}_0(s) = s \qquad \mathcal{D}_{n+1}(s) = \mathcal{D}_n(1^{ff} \,.\, s \circ \uparrow)$$

Pushing a substitution s under a binder p thus changes s to $\mathcal{D}_{\#p}(s)$.

In the following we will leave out the end-of-spine () in most cases to increase readability.

An expression generally takes the form

$$\begin{aligned} E = {} & \mathsf{let}\ \{p_1\} = H_1 \cdot S_1\ \mathsf{in} \\ & \mathsf{let}\ \{p_2\} = H_2 \cdot S_2\ \mathsf{in} \\ & \qquad\qquad \vdots \\ & \mathsf{let}\ \{p_n\} = H_n \cdot S_n\ \mathsf{in} \\ & M \end{aligned}$$

It is instructive to consider what happens when such a sequence of let-bindings is substituted for a logic variable occurring as the head in another let-binding. If we have $\mathsf{let}\ \{p\} = X[s]\ \mathsf{in}\ E'$ and instantiate X to $\{E\}$ we get

$$\begin{aligned} & \mathsf{let}\ \{p_1\} = (H_1 \cdot S_1)[s]\ \mathsf{in} \\ & \mathsf{let}\ \{p_2\} = (H_2 \cdot S_2)[\mathcal{D}_{\#p_1}(s)]\ \mathsf{in} \\ & \qquad\quad \vdots \\ & \mathsf{let}\ \{p_n\} = (H_n \cdot S_n)[\mathcal{D}_{\#p_1+\cdots+\#p_{n-1}}(s)]\ \mathsf{in} \\ & E'[\mathcal{D}_{\#p}(\uparrow^{\#p_1+\cdots+\#p_n})][M[\mathcal{D}_{\#p_1+\cdots+\#p_n}(s)]/p] \end{aligned}$$

I.e. a logic variable occurring as the head of a let-binding, such as in $\mathsf{let}\ \{p\} = X\ \mathsf{in}\ E'$, can by instantiation turn into an arbitrary number of let-bindings — including zero. Moreover, the variables bound in E' by p can have arbitrary terms substituted for them.

Now we consider the unification problem $E_1 \doteq E_2$ between two expressions. If the length of the let-sequence in both expressions is zero we have $M_1 \doteq M_2$, which we can simply decompose. If only one of them has length zero, the other must consist entirely of logic variables as the head of every let-binding, which consequently has to be instantiated to let-sequences of length zero.

Let $\mathsf{let}\ \{p\} = X[i_1][s]\ \mathsf{in}\ E \doteq M'$ where s is a pattern substitution and i_1 is a linear-changing identity substitution. We would like to construct an M matching p from fresh logic variables and instantiate X to $\{M\}$, but there is a problem. If p is e.g. $\downarrow y_1 \otimes \downarrow y_2$ then $M = \downarrow Y_1 \otimes \downarrow Y_2$, but since we have a multiplicative context split, we cannot compute the contexts of Y_1 and Y_2.

We can solve this problem by noticing that a unification constraint that is postponed only because of unresolved linearity pruning represents an unresolved context split.

[5] We do not need to know f as the two flags are equal and therefore represented by `ID`.

Let $X :: A_X$ in Γ_X and take i to be the linear-changing identity substitution with $\mathcal{I}(\Gamma_X) \vdash i : \Gamma_X$ where $\mathcal{I}(\Gamma)$ is defined as follows:

$$\mathcal{I}(\cdot) = \cdot \quad \mathcal{I}(\Gamma, A^{\mathbf{I}}) = \mathcal{I}(\Gamma), A^{\mathbf{I}} \quad \mathcal{I}(\Gamma, A^{\mathbf{A}}) = \mathcal{I}(\Gamma), A^{\mathbf{I}} \quad \mathcal{I}(\Gamma, A^{\mathbf{L}}) = \mathcal{I}(\Gamma), A^{\mathbf{I}}$$

We leave $\mathcal{I}(\Gamma)$ undefined for used affine and used linear assumptions as we assume that they have been pruned from X.

Since the domain of i equals the domain of i_1 and i is the maximally linear-changing identity substitution with domain Γ_X, there exists a linear-changing identity substitution i_2 such that $i = i_1 \circ i_2$. Now we construct M in the context $\mathcal{I}(\Gamma_X)$, and since this context has no linear or affine assumptions it can be trivially split. This means that we can postpone the instantiation of X as the equation $X[i] \doteq \{M\}$.

Take i'_2 to be the linear-changing identity substitution such that $i_2 \circ s = s \circ i'_2$ (see Lemma 6.4.1 above). Considering the original unification equation we apply i'_2 to both sides and get:

$$\begin{aligned} M'[i'_2] &\doteq \mathsf{let}\ \{p\} = X[i_1][s][i'_2]\ \mathsf{in}\ E[\mathcal{D}_{\#p}(i'_2)] \\ &= \mathsf{let}\ \{p\} = X[i][s]\ \mathsf{in}\ E \\ &\doteq \mathsf{let}\ \{p\} = \{M\}[s]\ \mathsf{in}\ E \\ &= E[M[s]/p] \end{aligned}$$

This means that we can continue solving the unification problem $M'[i'_2] \doteq E[M[s]/p]$ as if we had instantiated X. Eventually, the logic variables introduced to form M are hopefully going to be instantiated, and linearity pruning will be able to deal with the remaining $X[i] \doteq \{M\}$.

A few things require comments. The equality $E[\mathcal{D}_{\#p}(i'_2)] = E$ might seem counterintuitive. But i'_2 is the identity on all variables except for a few linear and affine variables that are going to occur in X, and therefore they do not occur in E. Also, the trick of applying a substitution to both sides of the equation and going from there is obviously sound, but it is also complete since i'_2 is injective.

Now we are left with the case $E_1 \doteq E_2$ where the let-sequences in E_1 and E_2 both have non-zero lengths. Let $E_1 = \mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ E'_1$ where H is not a logic variable.

At this point we are going to introduce non-determinism. We non-deterministically choose a specific let-binding in E_2 to unify with $H \cdot S$. We let-float it to the top of E_2 such that $E_2 = \mathsf{let}\ \{p'\} = H' \cdot S'\ \mathsf{in}\ E'_2$ where $\{p'\} = H' \cdot S'$ is the chosen binding. If H' is not a logic variable we decompose the unification problem into:

$$H \cdot S \doteq H' \cdot S' \quad \wedge \quad E'_1 \doteq E'_2$$

If H' is a logic variable $X[i][s]$ under a linear-changing identity substitution i and a pattern substitution s, we instantiate X such that

$$X[i][s] = \{\mathsf{let}\ \{p\} = H \cdot S\ \mathsf{in}\ \mathsf{let}\ \{p'\} = Y[s']\ \mathsf{in}\ \overline{p'}\}$$

where Y is a fresh logic variable and $s' = \mathcal{D}_{\#p}(s)$ (this might induce a strengthening of Y due to linearity pruning with respect to i). The context of Y can be calculated from the context of X by synthesizing the type of $H \cdot S$. Our unification problem is now reduced to:

$$E'_1 \doteq \mathsf{let}\ \{p'\} = Y[s']\ \mathsf{in}\ E'_2[\mathcal{D}_{\#p'}(\uparrow^{\#p})]$$

```
nat : type.
z : nat.
s : nat -> nat.

list : type.
nil : list.
cons : nat -> list -> list.

mset : type. % multisets represented as the type 'mset -> {1}'
cell : mset -> nat -> {1}.

same : list -> (mset -> {1}) -> type.
same_nil : same nil (\!m. {1}).
same_cons : same (cons N L) (\!m. {let {1} = cell m N in
        let {1} = S !m in 1})
   <- same L S.

#query * * * 1 same L
   (\!m. {let {1} = cell m (s z) in let {1} = cell m z in 1}).
```

Figure 6.3: List and multiset encoding in Celf

This is slightly simplified, as we will not let-float the entire instantiation of X to the top of E_2, but only the part that is to unify with $H \cdot S$, and leave the newly created Y at the depth of X — otherwise we would cause an unnecessary restriction on variable occurrences in Y.

A few things might happen to disrupt the steps outlined above. It might be the case that $H \cdot S$ cannot unify with $H' \cdot S'$, e.g. if the heads do not match up. Or we might not be able to let-float due to dependencies on bound variables in E_2. In the latter case the dependencies might go away through instantiations if the conflicting variables are bound in a let-binding with a logic variable as the head. This means that many of our possible non-deterministic choices can be disregarded by simple checks.

If the number of non-deterministic choices left is either one or zero, we can continue with the remaining unification problem or fail, respectively. If the number of choices is two or more we need to resolve the remaining non-determinism in some way.[6] We do this in two different ways. If we are doing type reconstruction we are looking for most general unifiers and we therefore postpone the constraint. If we are doing proof search, however, we will introduce a backtracking point and try all the possibilities as illustrated by example 6.1 below.

Example 6.1. The Celf signature in Figure 6.3 shows an encoding of lists and multisets of natural numbers. The relation `same L M` expresses that the list `L` and the multiset `M` contain the same elements.

The output of the query is the following:

[6] In the special case where one of the choices has syntactically identical $H \cdot S$ and $H' \cdot S'$ and $\#p = 0$, this choice is at least as general as any other and we can commit to this particular choice without loosing any solutions.

```
Query (*, *, *, 1) same #L (\!m. {
    let {1} = cell !m !(s !z) in
    let {1} = cell !m !z in 1}).
Solution: same_cons !(same_cons !same_nil)
 #L = cons !(s !z) !(cons !z !nil)
Solution: same_cons !(same_cons !same_nil)
 #L = cons !z !(cons !(s !z) !nil)
```

The query asks for all permutations of $\{1, 0\}$ and gets two results as expected, due to the exhaustive exploration of the non-deterministic unification during proof search as mentioned above.

We have now covered all cases for $E_1 \doteq E_2$ except one. It might be that both let-sequences begin with a logic variable. In the general case we postpone the equation, but if $\{E_1\}$ can η-reduce to $X[s]$ we can simply solve the equation as $X[s] \doteq \{E_2\}$. This can only be done in general if X does not occur as a head in the let-sequence of E_2. In the latter case we might be able to η-convert $\{E_2\}$ to $X[t]$ and solve it as an intersection problem.

Chapter 7

Conclusion and Future Work

In this thesis I have presented several theoretical results to support the implementation of substructural logical frameworks. I have proven strong normalization for the explicit substitution calculus $\lambda\sigma$, which can be used to implement any λ-calculus-based system and is already used in Twelf and Celf. I have designed a linear and affine type system for explicit substitutions, which solves the problem of type-preservation for a type-oblivious reduction semantics, thus enabling a simple and flexible implementation. And I have used this type system to define the pattern fragment for higher-order unification in the presence of linear and affine types along with a deterministic unification algorithm. Furthermore, I have extended the unification algorithm to the linear-changing pattern fragment in order to bridge the gap to the intuitionistic pattern fragment.

I have then extended and implemented the substructural type theory and logical framework CLF. I have named this implementation Celf. This implementation builds directly on top of all the theoretical results presented, and it supports, among other things, intuitionistic, affine, and linear types, the representation of concurrency and resources, implicit parameter and type inference, proof search, and an advanced unification constraint simplification algorithm for the dynamic affine and linear pattern fragment.

With the implementation of Celf, a natural next step could be to explore the representation and validation of meta-theory, thus turning Celf into a full-fledged proof assistant. There are two obvious directions this exploration could take. The first is to develop a methodology for the encoding of meta-theoretical proofs directly in the type theory in a way that supports automated verification and to develop these verification algorithms. This approach would mimic the philosophy of Twelf. The second alternative is to develop a meta-logic on top of the type theory mimicking the approach taken by the proof assistant Abella. Abella already provides the ∇ quantifier to lift variable binding to the meta-logic, and it would have to be explored how such a quantifier could be extended to lift affine and linear assumptions.

Bibliography

[Abea] Abella. http://abella.cs.umn.edu/.

[Abeb] Girard's proof of strong normalization of the simply-typed lambda-calculus. http://abella.cs.umn.edu/examples/. Abella examples.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(04):375–416, 1991.

[And92] J.M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[Bar00] Bruno Barras. Programming and Computing in HOL. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 17–37, London, UK, 2000. Springer-Verlag.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03359-9_6.

[BGM$^+$07] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *21th Conference on Automated Deduction (CADE-21)*, pages 391–397. Springer Verlag, LNAI 4603, 2007.

[Bie94] G. Bierman. *On Intuitionistic Linear Logic.* PhD thesis, University of Cambridge, 1994.

[BR95] Roel Bloo and Kristoffer H. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *Computer Science in the Netherlands (CSN'95)*, pages 62–72, 1995.

[CdPR99] Iliano Cervesato, Valeria de Paiva, and Eike Ritter. Explicit Substitutions for Linear Logical Frameworks: Preliminary Results. In A. Felty, editor, *Workshop on Logical Frameworks and Meta-languages — LFM'99*, Paris, France, 28 September 1999.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *Journal of the ACM (JACM)*, 43(2):362–397, 1996.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient Resource Management for Linear Logic Proof Search. *Theoretical Computer Science*, 232(1-2):133–163, February 2000.

[CHR92] P.-L. Curien, T. Hardin, and A. Ríos. Strong Normalization of Substitutions. *Mathematical Foundations of Computer Science 1992*, pages 209–217, 1992.

[Clf] Celf – A Logical Framework for Deductive and Concurrent Systems. `http://www.twelf.org/~celf`.

[CNR08] Avik Chaudhuri, Prasad Naldurg, and Sriram Rajamani. A type system for data-flow integrity on Windows Vista. *SIGPLAN Notices*, 43(12):9–20, 2008.

[Coq] The Coq Proof Assistant. `http://coq.inria.fr/`.

[CP96] Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science, LICS'96*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CP97] Iliano Cervesato and Frank Pfenning. Linear higher-order preunification. In *Twelfth Annual Symposium on Logic in Computer Science — LICS'97*, pages 422–433. IEEE Computer Society Press, 1997.

[CP03] Iliano Cervesato and Frank Pfenning. A Linear Spine Calculus. *Journal of Logic Computation*, 13(5):639–688, October 2003.

[CP10] Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236, Paris, France, August 2010. Springer.

[CPWW02a] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University. Department of Computer Science, 2002.

[CPWW02b] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University. Department of Computer Science, 2002.

[DHKP98] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via Explicit Substitutions: The Case of Higher-Order Patterns. Rapport de Recherche 3591, INRIA, December 1998. Preliminary version appeared at JICSLP'96.

[Gac08] Andrew Gacek. The Abella Interactive Theorem Prover (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008.

[GdPR00] N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. *Logic Journal of IGPL*, 8(1):7, 2000.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. Preliminary version appeared in *Symposium on Logic in Computer Science*, 1987.

[HM94] Joshua S. Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[Hue75] Gérard Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[Isa] Isabelle. `http://isabelle.in.tum.de/`.

[Kes07] Delia Kesner. The theory of calculi with explicit substitutions revisited. In *Computer Science Logic*, pages 238–252. Springer, 2007.

[LM07] C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. *Lecture Notes in Computer Science*, 4646:451–465, 2007.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 35–46, Lisbon, Portugal, 2005.

[Mel95] Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. *Typed Lambda Calculi and Applications*, pages 328–334, 1995.

[Mil91] Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Muñ98] César Muñoz. Dependent Types with Explicit Substitutions: A Meta-theoretical Development. In *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 294–316. Springer, 1998.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System Description: Teyjus – A Compiler and Abstract Machine Based Implementation of lambda-Prolog. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 287–291. Springer, 1999.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *Transactions on Computational Logic*, 9(3), 2008.

[NW98] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms. a generalization of environment. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[Oka95] Chris Okasaki. Purely Functional Random-Access Lists. In *Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 86–95. ACM, 1995.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In *Logical frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pie10] Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. Submitted for publication, August 2010.

[PS99] Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[PS08] Adam Poswolksy and Carsten Schürmann. Practical Programming with Higher-Order Encodings and Dependent Types. In Sophia Drossopoulou, editor, *17th European Symposium on Programming, ESOP'08*, pages 93–107, Budapest, Hungary, 2008. Springer Verlag LNCS4960.

[Ree09] Jason Reed. Higher-order constraint simplification in dependent type theory. In *LFMTP '09: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages*, pages 49–56, New York, NY, USA, 2009. ACM.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 313–323, New York, NY, USA, 1998. ACM.

[SN10] Anders Schack-Nielsen. The $\lambda\sigma$-Calculus and Strong Normalization. Technical Report ITU-TR-2010-132, IT University of Copenhagen, October 2010.

[SNS08] Anders Schack-Nielsen and Carsten Schürmann. System Description: Celf – A Logical Framework for Deductive and Concurrent Systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, pages 320–331, Sydney, Australia, 2008.

[SNS10a] Anders Schack-Nielsen and Carsten Schürmann. Curry-Style Explicit Substitutions for the Linear and Affine Lambda Calculus. In Jürgen Giesl and Reiner Hähnle, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, UK, 2010. Springer Berlin / Heidelberg. 10.1007/978-3-642-14203-1_1.

[SNS10b] Anders Schack-Nielsen and Carsten Schürmann. Pattern Unification for the Lambda Calculus with Linear and Affine Types. *Electronic Proceedings in Theoretical Computer Science*, 34:101–116, 2010. In Proceedings LFMTP 2010.

[SPM03] M. R. Shinwell, A.M. Pitts, and M.J.Gabbay. FreshML: Programmming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.

[WCPW08] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying Properties of Concurrent Computations in CLF. *Electronic Notes in Theoretical Computer Science*, 199:67–87, 2008. Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04).

[WM02] Daniel C. Wang and Tom Murphy VII. Programming with Recursion Schemes. Agere Systems / Carnegie Mellon University, unpublished manuscript, 2002.