Mechanizing Types and Programming Languages: A Companion

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

Contents

Preface

This book provides an introduction to mechanizing the meta-theory of programming languages. Mechanizing formal systems and proofs about them plays an increasingly important role in this area and being literate in mechanizing formal systems and proofs about them has become essential for students and researchers in programming languages and type systems. More importantly, mechanizing programming language theory will provide a deeper understanding, allows easy exploration of variations and extensions, and provides immediate feedback.

This exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. It is written as a companion to B. Pierce's book "Types and Programming Languages (TAPL)" that provides an introduction to how to mechanize the meta-theory of types and programming languages. While it is meant to be read at the same time as TAPL, we provide enough context and background that it should also be easily accessible to a reader who has read TAPL in the past or has already basic knowledge of types and programming languages. We give a roadmap in Fig. 1. The material from the core covers about one semester's worth of material and has been used at McGill University for teaching the course "Language-based security", a course open to advanced undergraduates and beginning graduate students and leaves room to either explore more in depth some mechanizations or explore extensions.

There are also interesting further extensions to explore; for most of those only Beluga code exists at this point without a detailled explanation.

A key question when mechanizing formal systems and proofs is the choice of proof environment. We have chosen here Beluga, a dependently typed programming and proof environment as it directly supports key and common concepts that frequently arise when describing formal systems and derivations within them; in particular it provides infrastructure for modelling variable binders and their scope, it supports first-class contexts to abstract, manage, and manipulate a set of assumptions, it support modelling derivations that depend on assumptions, and has built-in first-class (simultaneous) substitutions. As such it provides one of the most advanced infrastructures for such an endeavor. As we will show, the theory of programming

Topic	TAPL	META
Basics		
Preliminaries	CH 1, CH 2	CH 2
Basic Arith. Terms, Types and Proofs	CH 3, CH 8	
Encoding Basic Terms and Types		CH 2.1, CH 2.3
Encoding Basic Proofs as Functions		CH 3
Intermediate		
Untyped Lambda-calculus	CH 5	
Encoding Variables and Binders		CH 4.1, CH 4.2.1
Typed Lambda-Calculus	CH 8, CH 9	
Encoding Hypothetical Derivations, i.e. Derivations that depend on Assumptions		CH 4.2.2
Encoding Proofs about Closed Derivations		CH 5.1
Encoding Proofs about Hypoth. Derivations		CH 5.2
Advanced		
Normalization	CH 12	CH 7

Figure 1: Roadmap

Topic	TAPL	META
Nameless Representation (De Bruijn)	CH 6	CH 4.1, CH 6.1
References	CH 13	Planned
Exceptions	CH 14	Beluga Code
Subtyping	CH 15	Planned
Recursive Type	CH 20	Planned
Bounded Quantification	CH 26	Beluga Code

Figure 2: Extensions

languages does not require a deep and complicated mathematical apparatus, but can be carried out in a concrete, intuitive, and computational way, when the right abstractions are provided.

Note to instructors If you intend to use this material for your own course, we would love to hear from you. We welcome all comments, questions, and suggestions.

Acknowledgements The following people have helped shape and develop the companion. Agata Murawska deserves thanks for improving the structure and organization of this text and carefully proof-reading parts. Nicolas Jeannerod described the translation from small-step to big-step semantics in Section 3.4. Francisco Ferreira wrote up the normalization proof for the simply typed lambda calculus (Chapter 7) which was originally developed by Andrew Cave. Finally, thanks goes to the undergraduate and graduate students at McGill University who read earlier versions of this manuscript and played with BELUGA as part of the graduate course on types and programming languages.

Chapter 1

Introduction

> Computer Science is a science of abstraction – creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.
> - A. Aho and J. Ullman

Mechanizing formal systems, given via axioms and inference rules, together with proofs about them plays an important role in establishing trust in formal developments. In particular, programming language designers and developers have embraced proof assistants to verify software artifacts (see for example verification of the LLVM [Zhao et al. 2012], VTS [Appel 2011], and CompCert [Leroy 2009])). More generally, mechanizing formal systems and their meta-theory (at least partially) allows us to gain a deeper understanding of these systems and as we are working with more realistic and larger specifications, proof assistants are becoming an essential tool to deal with the growing complexity of languages and proofs about them.

Realistic languages have numerous cases to be considered, and while many of them will be straightforward, the task of verifying them all can be complex. Consequently it can be difficult to define a language correctly, and prove the appropriate theorems – let alone maintain the definition and the associated proofs when the language evolves and changes. Being able to animate the use of formal definitions using proof assistants, allows us to gain a deeper understanding of the systems we are working with. Being able mechanize proofs about formal definitions, forces us to clearly understand each step and leaves no room for error. It also substantially facilitates the maintenance of proofs as the languages evolve and change.

1.1 Approach

A key question in this endeavor is how to represent formal systems and derivations. To encode the formal systems, we need to understand how to represent variables, enforce their scope, and deal with operations such as renaming and substitution. When representing derivations, we face the challenge that they often depend on sets of assumptions; we therefore need to understand how to represent a set of assumptions, how to enforce the scope of assumptions within a derivation, and how to support weakening and substitution properties. The choices we make will ultimately impact our proof developments. It will determine how much effort is required, how feasible a given development is in practice, how reusable and extendable a proof development is, how easy it is to automatically find the proof, and how readable the final proof is.

General purpose proof environments such as Coq [Bertot and Castéran 2004] or Agda [Norell 2007] are built on powerful dependent type theories, but lack direct support for many intricate aspects that arise when we tackle the mechanization of formal systems and proofs. Instead the user chooses among a variety of techniques or libraries for binders that may smooth the path (see Aydemir et al. [1990]). However, this often comes with a heavy price tag - especially for more advanced proofs. Consider for example proofs by logical relations, a proof technique going back to Tait [1967] and later refined by Girard et al. [1990]. Such proofs play a fundamental role to establish rich properties such as contextual equivalence or normalization. The central idea of logical relations is to specify relations on well-typed terms via structural induction on the syntax of types instead of directly on the syntax of terms themselves. Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise. While proofs using logical relations often are not large, they are conceptually intricate, since they rely heavily on various properties of variables and substitutions. For example, Berardi [1990] remarked about his mechanization of Girard's normalization proof:

> "We may think of Girard's proof as an iceberg. In the top of it, we find what we usually consider the real proof; underwater, the most of the matter, consisting of all mathematical preliminaries a reader must know in order to understand what is going on."

To visualize the overhead involved consider Fig. 1.1. Above the water, is the main part of the proof which is often fairly straightforward. However, when mechanizing formal systems we also need to implement bound variables and their scope, work modulo renaming of variable names, keep track and manage a list of assumptions and enforce properties about them, and define substitution together with its equational theory just to name a few. Early work [Berardi 1990; Coquand 1992; Altenkirch

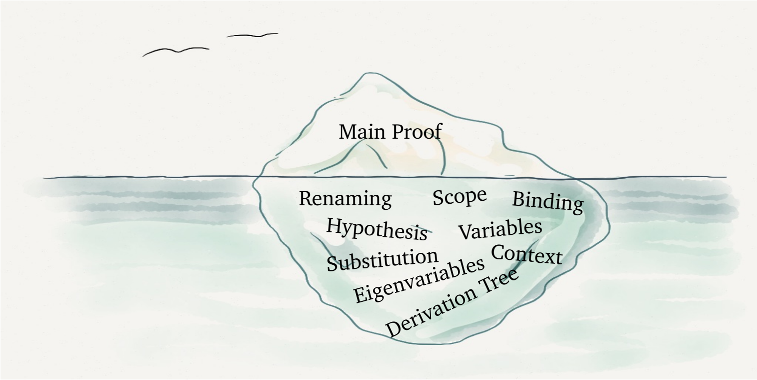

Figure 1.1: Proofs: The tip of the iceberg

1993] within general purpose proof environment represented lambda-terms using (well-scoped) de Bruijn indices which leads to a substantial amount of overhead to prove properties about substitutions such as substitution lemmas and composition of substitution. To improve readability and generally better support such meta-theoretic reasoning, nominal approaches support α-renaming but substitution and properties about them are specified separately; the Isabelle Nominal package [Urban 2008] has been used in a variety of logical relations proofs from proving strong normalization for Moggi's modal lambda-calculus [Doczkal and Schwinghammer 2009] to mechanically verifying the meta-theory of LF itself, including the completeness of equivalence checking [Narboux and Urban 2008; Urban et al. 2011]. However, these approaches still lack the right level of abstraction. It should not matter how we implement variable binding, contexts and substitutions – rather, users should be able to use these concepts first-class.

BELUGA [Pientka and Dunfield 2010; Pientka and Cave 2015] follows this path and provides a sophisticated infrastructure to deal automatically with many bureaucratic details regarding variables and assumptions, while giving up (at this point) some of the computational power present in general proof assistants such as Coq or Agda. We harness the power of abstraction, by defining common concepts such as variable binding, derivations depending on sets of assumptions together with sets of operations and properties. While these concepts, their operations, and properties are grounded and carefully defined theoretically, users do not need to understand any technical details on how these concepts are implemented nor do they need to prove (often standard) lemmas; instead they simply use these concepts abstractly through syntactic constructs provided. As a consequence, mechanizing proofs about formal systems, even advanced proofs by logical relations, does not require a deep and complicated mathematical apparatus, but can be carried out in a direct and intuitive way. In fact, the proof language used for developing the main proof (i.e. what is above the water) is a proof term language for first-order logic with induction over contexts and derivation trees (objects that are first-class). In essence, BELUGA's proof language is comparable to first-order logic with induction over a specific domain – in our case the domain must be rich enough to allow us to describe derivation trees that may depend on assumptions. By doing so we harvest many of the same benefits that abstract data types bring to programming languages [Liskov and Zilles 1974]: we can change the concrete representation implementation of variables, substitutions, and contexts choosing what is most efficient; proof developments are modular and as users do not have direct access to the concrete implementation of variables, substitutions, and contexts, they also cannot add any tempting, but dangerous shortcuts that would corrupt part of the infrastructure; most importantly, proof developments are easier to understand and to read as the high-level steps are not obscured by low-level operations and as a consequence, we stand a chance to automate such proofs.

In this exposition, we introduce BELUGA by example, starting from a simple language of arithmetic expressions (as described for instance in [Pierce 2002, Ch 3, Ch 8]) and showing how to encode this language together with simple inductive proofs in BELUGA. This will allow us to introduce the basic idea of encoding proofs as recursive functions. We then extend our simple language of arithmetic expressions to include functions (as for example described in [Pierce 2002, Ch 5, Ch 9]). This introduces several important ideas: how to encode variables and their scope, how to encode assumptions, and how to characterize derivations that depend on assumptions. Finally, we consider the proof of normalization for the simply typed lambda-calculus (see [Pierce 2002, Ch 12]) and show how to directly represent this proof as a recursive program in BELUGA. Throughout the text we will point to exercises that allow readers to practice and explore some topics further.

Words of wisdom We do not give a formal definition of BELUGA's language. Instead we are looking at examples and highlight how to *use* BELUGA rather than dwelling on how BELUGA is defined. This is in fact how most learn a given programming language – a language is best learned by using it. We believe learning BELUGA is best approached with a blank state of mind. Try not to compare it to functional languages you already know. While BELUGA is a functional language and has even a similar syntax to OCaml, it requires a different way of thinking about programming than what you have been acquainted with when using OCaml or ML.

1.2 Goals

We pursue several goals with this companion:

1. Understand how to define formal systems using higher-order abstract syntax
2. Curry-Howard isomorphism: Proofs = Programs
3. Deeper understanding of programming languages and their properties
4. Provide a tutorial to BELUGA

Part I

Basics: Mechanizing Simple Languages and Proofs

Chapter 2

Simple Language with Arithmetic and Booleans

> "A language that doesn't affect the way you think about programming, is not worth knowing." - Alan Perlis

We begin our gentle introduction to mechanizing languages and proofs by considering a simple language with arithmetic and boolean expressions. In general, definitions of programming languages typically cover three fundamental aspects: the grammar of the language, i.e. what are syntactically well-formed terms in the language, its operational semantics, i.e. how do we execute and evaluate a given term, and its type structure, i.e. what are well-typed expressions. We revisit these concepts for a small language containing booleans and numbers following [Pierce 2002, Ch 3,Ch 8] in preparation for representing this language together with its operational semantics and type system in BELUGA.

2.1 Terms

Let us consider a simple language containing booleans, if-expressions, and numbers together with simple operations that allow us to test whether a given number is zero (see [Pierce 2002, Ch 3, Fig 3-1,Fig 3-2]). We define numbers using z and succ -operation. To analyze and manipulate numbers, we also provide a pred -operator.

$$\text{Terms}\quad M ::= \mathsf{false} \mid \mathsf{true} \mid \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \mid \mathsf{z} \mid \mathsf{succ}\ M \mid \mathsf{pred}\ M \mid \mathsf{iszero}\ M$$

The first question we investigate is how to define and represent terms in the proof environment BELUGA. To represent terms in BELUGA, we declare a type `term` for terms together with constants corresponding to the constructors false, true, z, succ , pred , iszero , etc. More precisely we use the logical framework LF [Harper et al. 1993] for introducing new types such as `term` together with constants `true`, `false`, etc. that can be used to construct terms. All constants are used as prefix operators by default.

```
LF term : type =
| true         : term
| false        : term
| if_then_else : term → term → term → term
| z            : term
| succ         : term → term
| pred         : term → term
| iszero       : term → term
;
```

To illustrate consider a few examples. We write in sanserif font what we would write on paper and use type writer font for its equivalent representation in BELUGA.

if false then z else succ z	is represented as	`if_then_else false z (succ z)`
iszero (pred (succ z))	is represented as	`iszero (pred (succ z))`.

2.2 Semantics

Next, we define how to evaluate and execute a given term. Following Pierce [2002], we define a small-step semantics for our little language. To give a deterministic evaluation strategy, it is convenient to first define the values of our language. In Pierce [2002], we distinguished between terms and values in the grammar itself and we think of values as a sub-class of terms; moreover, we further distinguish between numerical values and boolean values. Here we take a slightly different approach and define explicitly what it means to be a value using the judgment M value. We can think of these judgments as predicates on terms which are defined *via* axioms and inference rules. A term M is a a value iff we can give a derivation for M value. We also do not further distinguish between numerical values and booleans. Although this can be done, such a distinction makes many of the theoretical properties needlessly more complex.

$\boxed{V \text{ value}}$: Expression V is a value

$$\frac{}{\mathsf{z\ value}}\ \textsc{V-Zero} \qquad \frac{V\ \mathsf{value}}{(\mathsf{succ}\ V)\ \mathsf{value}}\ \textsc{V-Succ} \qquad \frac{}{\mathsf{true\ value}}\ \textsc{V-True} \qquad \frac{}{\mathsf{false\ value}}\ \textsc{V-False}$$

Here, we identify z as a value; in addition, succ V is a value provided that V is a value. We also identify true and false as values, and say that every numeric value is a value.

We are now ready to define when a term M steps to another term N using the judgment $M \longrightarrow N$ using congruence rules and reduction rules.

$\boxed{M \longrightarrow N}$: Term M steps to term N

Congruence Rules

$$\frac{M \longrightarrow N}{\mathsf{succ}\ M \longrightarrow \mathsf{succ}\ N}\ \textsc{E-Succ} \qquad \frac{M \longrightarrow N}{\mathsf{pred}\ M \longrightarrow \mathsf{pred}\ N}\ \textsc{E-Pred}$$

$$\frac{M \longrightarrow N}{\mathsf{iszero}\ M \longrightarrow \mathsf{iszero}\ N}\ \textsc{E-Iszero}$$

$$\frac{M_1 \longrightarrow M_1'}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow \mathsf{if}\ M_1'\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3}\ \textsc{E-If}$$

Reduction Rules (Axioms)

$$\frac{}{\mathsf{pred}\ \mathsf{z} \longrightarrow \mathsf{z}}\ \textsc{E-PredZero} \qquad \frac{V\ \mathsf{value}}{\mathsf{pred}\ (\mathsf{succ}\ V) \longrightarrow V}\ \textsc{E-PredSucc}$$

$$\frac{}{\mathsf{iszero}\ \mathsf{z} \longrightarrow \mathsf{true}}\ \textsc{E-IszeroZero} \qquad \frac{V\ \mathsf{value}}{\mathsf{iszero}\ (\mathsf{succ}\ V) \longrightarrow \mathsf{false}}\ \textsc{E-IszeroSucc}$$

$$\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow M_2}\ \textsc{E-IfTrue}$$

$$\frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow M_3}\ \textsc{E-IfFalse}$$

Our goal is to represent the relation $M \longrightarrow N$ and describe the derivation trees which correspond to evaluating a given sub-term. To accomplish this we also must be able to construct derivation trees that prove that a given term is a value. Clearly, inductively defined terms can be easily translated into objects of type `term` using the constants `true`, `false`, `if_then_else`, `z`, `suc`, `pred`, and `iszero`. However, we also defined inductively what it means for a term to be a value: we use the axioms V-Zero, V-True, V-False together with the inference rule V-Succ. Similarly, we can translate derivation trees that prove V value and derivation trees that show $M \longrightarrow N$.

In BELUGA, types are powerful enough to encode such definitions about predicates and relations. To represent the judgment V value we define a predicate (type family) `value` and state that it takes terms as argument by declaring its type as `term` → **type**. Then each rule corresponds to a constructor in our data-type definition. A derivation tree then corresponds to an expression formed by these constructors.

```
LF value : term → type =
| v_zero     : value z
| v_succ     : value V → value (succ V)
| v_true     : value true
| v_false    : value false
;
```

In our definition of the constructor `v_succ`, the capital letter `V` describes values and is thought to be universally quantified at the outside. We can hence read the constructor `v_succ` as follows:

> "For all term `V`, if `D` is a derivation for `value V`, then we can form a derivation `(v_succ V D)` for `value (succ V)`."

We mark here the term `V` in green since in practice programmers can omit writing it and BELUGA will infer it. The recipe is "if we do not explicitly quantify over variables in the definition of a constructor, we do not need to pass instantiations for them when constructing objects using the said constructor."

To illustrate consider the following concrete example of proving that succ (succ z) is a value and constructing a derivation for succ (succ z) value.

$$\dfrac{\dfrac{\dfrac{}{\text{z value}}\ \text{V-Zero}}{(\text{succ z})\ \text{value}}\ \text{V-Succ}}{(\text{succ }(\text{succ z}))\ \text{value}}\ \text{V-Succ}$$

is represented as `(v_succ (succ z) (v_succ z v_zero))`

In BELUGA, we bind the derivation to the name `v` by writing

```
let v : [⊢ value (succ (succ z))] = [⊢ v_succ (v_succ v_zero)];
```

Note, we write derivations using ⊢. On the left hand side of this symbol we can list assumptions and the right hand side describes what we claim can be derived from these assumptions. For example, we might say:

```
let w:[x: term, v: value x ⊢ value (succ (succ x))] =
      [x: term, v: value x ⊢ v_succ (v_succ v)];
```

The right hand side can be read as: Assuming `x: term` and `v: value x`, i.e. `v` is the derivation that states that `x` is a value, then `v_succ (v_succ v)` is the witness for the fact that `succ (succ x)` is a value. On paper we usually omit ⊢, if there are no assumptions, i.e. the objects or derivations we describe are closed.

We can similarly encode the small-step relation $M \longrightarrow N$ between the terms M and N using the type family/relation `step`.

```
LF step: term → term → type =
| e_if_true      : step (if_then_else true M2 M3) M2
| e_if_false     : step (if_then_else false M2 M3) M3
| e_pred_zero    : step (pred z) z
| e_pred_succ    : value V
                   → step (pred (succ V)) V
| e_iszero_zero  : step (iszero z) true
| e_iszero_succ  : value V
                   → step (iszero (succ V)) false
| e_if_then_else : step M1 M1'
                   → step (if_then_else M1 M2 M3) (if_then_else M1' M2 M3)
| e_succ         : step M N
                   → step (succ M) (succ N)
| e_pred         : step M N
                   → step (pred M) (pred N)
| e_iszero       : step M N
                   → step (iszero M) (iszero N)
;
```

Next, we show a few examples of how to encode and represent derivations that a concrete term steps to another.

```
let e1 : [⊢ step (pred (succ (pred z))) (pred (succ z))] =
        [⊢ e_pred (e_succ e_pred_zero)] ;

let e2 : [⊢ step (pred (succ z)) z] = [⊢ e_pred_succ v_zero] ;
```

Here `e1` stands for the derivation

$$\dfrac{\dfrac{\dfrac{}{(\mathsf{pred}\ \mathsf{z}) \longrightarrow \mathsf{z}}\ \textsc{E-PredZero}}{(\mathsf{succ}\ (\mathsf{pred}\ \mathsf{z})) \longrightarrow (\mathsf{succ}\ \mathsf{z})}\ \textsc{E-Succ}}{(\mathsf{pred}\ (\mathsf{succ}\ (\mathsf{pred}\ \mathsf{z}))) \longrightarrow (\mathsf{pred}\ (\mathsf{succ}\ \mathsf{z}))}\ \textsc{E-Pred}$$

The name `e2` stands for the derivation consisting of only the rule E-PREDSUCC.

2.3 Typing

Types allow us to approximate the runtime behaviour of programs. Types classify expressions according to their values, i.e. the value they compute. In our simple language with arithmetic and boolean expressions, we have only two types, namely Bool and Nat, corresponding to the two kinds of values.

$$\text{Types}\quad \mathsf{T} \;::=\; \mathsf{Bool} \mid \mathsf{Nat}$$

We encode these types in BELUGA using the LF type `tp` together with two constants `bool` and `nat`.

```
LF tp : type =
| bool : tp
| nat : tp
;
```

Typing rules relate terms to types and are described by the typing judgment $\mathsf{M} : \mathsf{T}$. We recall here simply the typing rules from Pierce [2002].

$\boxed{\mathsf{M} : \mathsf{T}}$: Term M has type T

$$\frac{}{\mathsf{true} : \mathsf{Bool}}\ \textsc{T-True} \qquad \frac{}{\mathsf{false} : \mathsf{Bool}}\ \textsc{T-False} \qquad \frac{}{\mathsf{z} : \mathsf{Nat}}\ \textsc{T-Zero}$$

$$\frac{\mathsf{M}_1 : \mathsf{Bool} \qquad \mathsf{M}_2 : \mathsf{T} \qquad \mathsf{M}_3 : \mathsf{T}}{\mathsf{if}\ \mathsf{M}_1\ \mathsf{then}\ \mathsf{M}_2\ \mathsf{else}\ \mathsf{M}_3 : \mathsf{T}}\ \textsc{T-If}$$

$$\frac{\mathsf{M} : \mathsf{Nat}}{\mathsf{succ}\ \mathsf{M} : \mathsf{Nat}}\ \textsc{T-Succ} \qquad \frac{\mathsf{M} : \mathsf{Nat}}{\mathsf{pred}\ \mathsf{M} : \mathsf{Nat}}\ \textsc{T-Pred} \qquad \frac{\mathsf{M} : \mathsf{Nat}}{\mathsf{iszero}\ \mathsf{M} : \mathsf{Bool}}\ \textsc{T-Iszero}$$

The typing relation between terms and types is encoded in BELUGA using a predicate (type family) `hastype` which is declared with `term → tp →` **type**. Each typing rule corresponds to a constructor in our definition.

```
LF hastype : term → tp → type =
| t_true    : hastype true bool
| t_false   : hastype false bool
| t_zero    : hastype z nat

| t_if      : hastype M1 bool → hastype M2 T → hastype M3 T
              → hastype (if_then_else M1 M2 M3) T

| t_succ    : hastype M nat
```

```
             → hastype (succ M) nat
| t_pred   : hastype M nat
             → hastype (pred M) nat
| t_iszero : hastype M nat
             → hastype (iszero M) bool
;
```

Chapter 3

Proofs by Induction

We now discuss some standard properties about languages which are also discussed in Pierce [2002] and show how we can represent such proofs as total functions that manipulate and analyze derivation trees. In particular, we illustrate:

- How to represent inductive proofs about derivations as total functions.
- How to prove that a case is impossible.
- How to state a property universally about terms and reason by induction on terms (rather than derivations).

We develop each example step-wise using the interactive programming features in BELUGA.

3.1 Type Preservation

The first property we re-visit is type preservation. In particular, we write $\mathsf{M} : \mathsf{T}$ and $\mathsf{M} \longrightarrow \mathsf{N}$ to clearly state that we only consider closed terms. We label here derivations with $\mathcal{D}$ and $\mathcal{S}$ writing $\mathcal{D} :: \mathsf{M} : \mathsf{T}$ and $\mathcal{S} :: \mathsf{M} \longrightarrow \mathsf{N}$ respectively.

Theorem 3.1.1. *If* $\mathcal{D} :: \mathsf{M} : \mathsf{T}$ *and* $\mathcal{S} :: \mathsf{M} \longrightarrow \mathsf{N}$ *then* $\mathsf{N} : \mathsf{T}$.

Proof. By structural induction on the derivation $\mathcal{S} :: \mathsf{M} \longrightarrow \mathsf{N}$. We consider here only a few cases.

Case $\mathcal{S} = \dfrac{}{\mathsf{pred\ z} \longrightarrow \mathsf{z}}\ \textsc{E-PredZero}$

$\mathcal{D}\ ::\ \mathsf{pred\ z : T}$ by assumption
$\mathcal{D}'\ ::\ \mathsf{z : Nat}\quad \text{and} \quad \mathsf{T = Nat}$ by inversion using rule T-Pred
$\quad ::\ \mathsf{z : Nat}$ by rule T-Zero.

Case $\mathcal{S} = \dfrac{\overset{\mathcal{S}'}{\mathsf{M} \longrightarrow \mathsf{N}}}{\mathsf{pred\ M} \longrightarrow \mathsf{pred\ N}}\ \textsc{E-Pred}$

$\mathcal{D}\ ::\ \mathsf{pred\ M : T}$ by assumption
$\mathcal{D}'\ ::\ \mathsf{M : Nat}\quad \text{and} \quad \mathsf{T = Nat}$ by inversion using rule T-Pred
$\mathcal{F}\ ::\ \mathsf{N : Nat}$ by IH using $\mathcal{D}'$ and $\mathcal{S}'$
$\quad ::\ \mathsf{pred\ N : Nat}$ by rule T-Pred. □

An inductive proof as the one here can be interpreted as recursive function where case-analysis in the proof corresponds to case analysis in the program and the appeal to the IH corresponds to making a recursive call. We also note that we wrote $\mathsf{M : T}$ (or $\mathsf{M} \longrightarrow \mathsf{N}$ resp.), but to emphasize that both the typing derivation and the stepping derivation are closed and do not depend on any assumptions we write $\vdash \mathsf{M : T}$ and $\vdash \mathsf{M} \longrightarrow \mathsf{N}$ respectively. This will scale as we generalize our language in Chapter 4 to include also variables, functions, recursion, and other constructs variable-binding constructs.

From a program point of view, we can read the type preservation theorem as: Given a typing derivation $\vdash \mathsf{M : T}$ and a derivation for $\vdash \mathsf{M} \longrightarrow \mathsf{N}$, we return a typing derivation $\vdash \mathsf{N : T}$.

We begin by translating and representing the actual theorem statement in BELUGA. This is straightforward keeping in mind that

On paper judgment	Type in BELUGA
$\vdash \mathsf{M : T}$	`[⊢ hastype M T]`
$\vdash \mathsf{M} \longrightarrow \mathsf{N}$	`[⊢ steps M N]`

```
rec tps: [⊢ hastype M T] → [⊢ step M N] → [⊢ hastype N T] = ? ;
```

Note that → is overloaded. We have used it so far in defining the type families `hastype`, `step`, `value`, and the type `term`. The arrow in these definitions corresponded to the line we draw, when we draw an inference rule to distinguish between the premises and the conclusions. We merely were using the arrow to define syntactic structures.

In BELUGA, we strictly separate between the objects we are constructing (such as derivation trees, terms, etc.) from proofs about them. The type preservation statement *makes a claim about the type of the term we obtain when taking a single step*. In the type of the function tps the function type → is much stronger; it for example allows us to write recursive functions which analyze objects of type [⊢ hastype M T] and [⊢ step M M'] by pattern matching.

Last, we wrote **?**. This is very useful when developing and debugging proofs/programs, since it allows us to describe incomplete proofs/programs and BELUGA will print back to you the assumptions at that given point and the goal which needs to be proven. Let's fill in some of the details.

Introducing assumptions - Writing functions Since we are proving an implication, we introduce two assumptions d:[⊢ hastype M T] and s:[⊢ step M N] and try to establish [⊢ hastype N T]. From a programmer's point of view, we need to build a function that when given d:[⊢ hastype M T] and s:[⊢ step M N] returns a derivation of type [⊢ hastype N T]. We use a concrete syntax similar to ML-like languages writing

```
rec tps: [⊢ hastype M T] → [⊢ step M N] → [⊢ hastype N T] =
fn d ⇒ fn s = ? ;
```

Case analysis - Pattern matching Next, we split the proof into different cases analyzing $\mathcal{S} :: M \longrightarrow N$. This corresponds to pattern matching on s:[⊢ step M N] in our program.

```
fn d ⇒ fn s ⇒ case s of
| [⊢ e_if_true]          ⇒ ?
| [⊢ e_if_false]         ⇒ ?
| [⊢ e_if_then_else S']  ⇒ ?
| [⊢ e_pred_zero]        ⇒ ?
| [⊢ e_pred_succ _]      ⇒ ?
| [⊢ e_iszero_zero]      ⇒ ?
| [⊢ e_iszero_succ _ ]   ⇒ ?
| [⊢ e_pred S']          ⇒ ?
| [⊢ e_succ S']          ⇒ ?
| [⊢ e_iszero S']        ⇒ ?
;
```

We sometimes use _ (underscore) for an argument, if we do not need a name for it, since it does not play a role in the proof. For example, when representing $\mathcal{S} = \dfrac{\mathsf{pred}\ (\mathsf{succ}\ V) \longrightarrow V}{V\ \mathsf{value}}\ \textsc{E-PredSucc}$ we simply write [⊢ e_pred_suc _] since the sub-derivation representing V value is not used in proving that types are preserved.

Convention: Variables describing sub-derivations, i.e. variables occurring inside `[]` must be upper-case. Variables describing proper assumptions in the proof, i.e. variables introduced by **fn**-abstraction, must be lower case.

Proving - Programming Let us now implement the two cases in the type preservation proof we discussed earlier. We start with the case `[e_pred_zero]` which corresponds to the base case in our proof. Pattern matching in `s` has not only generated all the cases, but more importantly it has refined what `M` and `N` stand for. In this particular case, `M = (pred z)` and `N = z`. As a first step in the proof, we analyzed the assumption `d:[⊢hastype (pred z) T]` further. We case-analyzed this assumption and we stated “by inversion on T-Pred” which indicated that there was exactly one case.

While we can write another case-expression analyzing `d` in the proof, Beluga provides syntactic sugar for case-expressions with one case; instead of writing **case** `d` **of** `[ ⊢t_pred D’]` ⇒ **?** we simply write **let** `[ ⊢t_pred D’] = d` **in** **?**. We now have learned that `T = nat`.

```
rec tps: [⊢ hastype M T] → [⊢ step M N] → [⊢ hastype N T] =
fn d ⇒ fn s ⇒ case s of
| [⊢ e_if_true]          ⇒ ?
| [⊢ e_if_false]         ⇒ ?
| [⊢ e_if_then_else S’]  ⇒ ?
| [⊢ e_pred_zero]        ⇒
  let [⊢ t_pred _ ] = d in ?
| [⊢ e_pred_succ _]      ⇒ ?
| [⊢ e_iszero_zero]      ⇒ ?
| [⊢ e_iszero_succ _ ]   ⇒ ?
| [⊢ e_pred S’]          ⇒ ?
| [⊢ e_succ S’]          ⇒ ?
| [⊢ e_iszero S’]        ⇒ ?
;
```

Beluga will compile this partial program and print for the hole

```
______________________________________________________________________________
- Meta-Context: .
______________________________________________________________________________
- Context:
tps: [ ⊢hastype M T] → [ ⊢step M N] → [ ⊢hastype N T]
d: [ ⊢hastype (pred z) nat]
s: [ ⊢step (pred z) z]

==============================================================================
- Goal Type: [ ⊢hastype z nat]
```

We now need to build an object that has type [⊢ hastype z nat]. This can simply be achieved by providing [⊢ t_zero].

For the step case where we are considering the case [⊢ e_pred S'], we also proceeded by analyzing d:[⊢ hastype (pred M') T] by pattern matching. There is only one constructor that could have been used to build d and we hence know that it must be of the form [⊢t_pred D'] where D' stands for a derivation [⊢ hastype N' nat] and we learn that T=nat.

We then appeal to the induction hypothesis in the proof using S and D'. This corresponds to making a recursive call tps [⊢D'] [⊢S'] and we name the resulting derivation [⊢F]. Finally, we construct our derivation [⊢t_pred F] for [⊢hastype (pred N') nat].

```
rec tps: [⊢ hastype M T] → [⊢ step M N] → [⊢ hastype N T] =
fn d ⇒ fn s ⇒ case s of
| [⊢ e_if_true]          ⇒ ?
| [⊢ e_if_false]         ⇒ ?
| [⊢ e_if_then_else S'] ⇒ ?
| [⊢ e_pred_zero]        ⇒
  let [⊢ t_pred _ ] = d in [ ⊢t_zero]
| [⊢ e_pred_succ _]      ⇒ ?
| [⊢ e_iszero_zero]      ⇒ ?
| [⊢ e_iszero_succ _ ]   ⇒ ?
| [⊢ e_pred S']          ⇒
  let [⊢ t_pred D'] = d in
  let [⊢ F] = tps [ ⊢D'] [ ⊢S'] in
    [⊢ t_pred F]
| [⊢ e_succ S']          ⇒ ?
| [⊢ e_iszero S']        ⇒ ?
;
```

When is a program a proof? So far we have just written a functional program; for it to be a proof it needs to be a total function, i.e. it must be defined on all inputs and it must be terminating. We can check that the function is total in BELUGA by writing the following annotation before we start writing the body of the function:/ **total** s (tps m t n d s) /. This annotation states that we claim to implement program tps that is recursive in s. Since in the statement we implicitly quantify over term M and M' as well as the type T at the outside, we write in the totality declaration (tps m t n d s) indicating that we are recursively analyzing the 4th argument (three of them are passed implicitly) passed to tps. The order in which the implicit arguments are listed is irrelevant; what is important is that their number is correct. The full proof is then written as follows:

```
rec tps: [⊢ hastype M T] → [⊢ step M N] → [⊢ hastype N T] =
/ total s (tps m t n d s) /
fn d ⇒ fn s ⇒ case s of
| [⊢ e_if_true] ⇒
  let [⊢ t_if_then_else D D1 D2] = d in [⊢D1]
| [⊢ e_if_false] ⇒
  let [⊢ t_if_then_else D D1 D2] = d in [⊢D2]
| [⊢ e_if_then_else S] ⇒
  let [⊢t_if_then_else D D1 D2] = d in
  let [⊢D'] = tps [⊢D] [⊢S] in
  [⊢ t_if_then_else D' D1 D2]
| [⊢ e_pred_zero] ⇒
  let [⊢t_pred _ ] = d in  [⊢t_zero]
| [⊢ e_pred_succ _ ] ⇒
  let [⊢t_pred (t_succ D) ] = d in [⊢D]
| [⊢ e_iszero_zero] ⇒
  let [⊢t_iszero _] = d in [⊢t_true]
| [⊢ e_iszero_succ _ ] ⇒
  let [⊢t_iszero _] = d in [⊢t_false]
| [⊢ e_pred S] ⇒
  let [⊢t_pred D] = d in
  let [⊢D'] = tps [⊢D] [⊢S] in
  [⊢ t_pred D']
| [⊢e_succ S] ⇒
  let [⊢t_succ D] = d in
  let [⊢D'] = tps [⊢D] [⊢S] in
  [⊢t_succ D']
| [⊢e_iszero S] ⇒
  let [⊢t_iszero D] = d in
  let [⊢D'] = tps [⊢D] [⊢S] in
  [⊢t_iszero D']
;
```

The full proof can be found online in the example directory corresponding to this chapter (see `Part1/type-preservation.bel`).

3.2 Uniqueness of Small-Step Evaluation

Next, we consider the proof that evaluation using the small-step rules yields a unique value. This is an interesting proof because we must argue that values do not step, i.e. there are no rules that apply. For `zero`, `true` and `false` this should be easy, since there is no rule that applies. But how do we argue that *every number* that is a value does not step? - We prove a contradiction. We show inductively that if M is a value and $\mathsf{M} \longrightarrow \mathsf{N}$ then we can derive falsehood (written as $\bot$).

Theorem 3.2.1. *If* $\mathcal{S} :: \mathsf{M} \longrightarrow \mathsf{N}$ *and* $\mathcal{V} :: \mathsf{M}\ \mathsf{value}$ *then* $\bot$.

Proof. By structural induction on the derivation $\mathcal{V} :: \mathsf{M}\ \mathsf{value}$.

Base case $\mathcal{V} = \dfrac{}{\mathsf{z}\ \mathsf{value}}$

$\mathcal{S} :: \mathsf{z} \longrightarrow \mathsf{N}$ by assumption

By inspecting all the existing rules, there exists no M'. Therefore, this assumption is false, and from false we can derive anything; in particular, we can conclude $\bot$.

Step case $\mathcal{V} = \dfrac{\begin{matrix}\mathcal{V}' \\ \mathsf{M}'\ \mathsf{value}\end{matrix}}{(\mathsf{succ}\ \mathsf{M}')\ \mathsf{value}}$

$\mathcal{S} :: (\mathsf{succ}\ M') \longrightarrow \mathsf{N}$ by assumption

$\mathcal{S}' :: \mathsf{M}' \longrightarrow \mathsf{N}'$ and $\mathsf{N} = (\mathsf{succ}\ \mathsf{N}')$ by inversion using E-SUCC

$\bot$ by i.h. using $\mathcal{S}'$ and $\mathcal{V}'$

□

How do we model in a programming environment ⊥ (falsehood)? - In a dependently typed language, we are modelling ⊥ indirectly. Recall that there is no way to for example construct an element of the type `step zero zero`. If we think of the set of elements belonging to the type `step M M'`, then `step zero zero` is not in it, but for example `step (pred zero) zero` is; so is `step (succ (pred zero)) (succ zero)`. Generally speaking, if we define no elements belonging to a type, then the type is guaranteed to be empty and models false. In BELUGA, we can define types without elements simply by declaring a type.

```
not_possible: type.
```

We then can translate the theorem directly into a computation-level type in BELUGA; we have also included the totality declaration, stating that this function is recursively defined on values, i.e. object of type `[ ⊢value M]`.

```
rec values_dont_step : [ ⊢step M N] → [ ⊢value M] → [ ⊢not_possible] =
/ total v (values_dont_step m n s v) /
?
;
```

As before, we introduce the assumption `s` for `[⊢ step M N]` and `v` for `[⊢ value M]`. Then we case-analyze `v`.

```
rec values_dont_step : [ ⊢step M N] → [ ⊢value M] → [ ⊢not_possible] =
/ total v (values_dont_step m n s v) /
fn s ⇒ fn v ⇒ case v of
| [ ⊢v_true]   ⇒ ?
| [ ⊢v_false]  ⇒ ?
| [ ⊢v_z ]     ⇒ ?
| [ ⊢v_s V’]   ⇒ ?
;
```

Let’s consider the case [⊢v_z] : [⊢value zero]. We argued in the proof by “By inspecting all the existing rules, there exists no N.” This corresponds to case-analyzing s; however, there are no cases. In BELUGA, we write **impossible** s for splitting s in the empty context. It is effectively a case-expression without branches.

For the step-case, the translation of the proof to a program is more straightforward: the inversion in the proof is translated to analyzing s; the appeal to the induction hypothesis corresponds to making a recursive call.

```
rec values_dont_step : [ ⊢step M N] → [ ⊢value M] → [ ⊢not_possible] =
/ total v (values_dont_step m m’ s v) /
fn s ⇒ fn v ⇒ case v of
| [ ⊢v_true]   ⇒ impossible s
| [ ⊢v_false]  ⇒ impossible s
| [ ⊢v_z ]     ⇒ impossible s
| [ ⊢v_s V’]   ⇒ let [ ⊢e_succ S’] = s in values_dont_step [ ⊢S’] [ ⊢V’];
```

One may wonder whether we can actually ever execute and run this program; the answer is no, since there is no way to provide a derivation [⊢step M N] and at the same time a proof [⊢value M].

We are now ready to prove that evaluation yields a unique result given the small-step semantics. This will illustrate how we can use the lemma values_dont_step. We only show two cases, but the whole program is available in the example directory corresponding to this chapter (see Part1/unique.bel).

```
LF equal: term → term → type =
| ref: equal T T;
```

```
rec unique : [⊢step M M1] → [⊢step M M2] → [ ⊢equal M1 M2] =
/ total s (unique m m1 m2 s)/
fn s1 ⇒ fn s2 ⇒ case s1 of
| [ ⊢e_if_true]  ⇒
  (case s2 of
   | [⊢e_if_true]          ⇒ [⊢ref]
   | [⊢e_if_then_else D] ⇒ impossible values_dont_step [⊢D] [⊢v_true] )

| [ ⊢e_if_false] ⇒
```

```
    (case s2 of
     | [⊢e_if_false]       ⇒ [⊢ref]
     | [⊢e_if_then_else D] ⇒ impossible values_dont_step [⊢D] [⊢v_false] )

  | [ ⊢e_if_then_else D] ⇒
    (case s2 of
    | [⊢e_if_true]        ⇒ impossible values_dont_step [⊢D] [⊢v_true]
    | [⊢e_if_false]       ⇒ impossible values_dont_step [⊢D] [⊢v_false]
    | [⊢e_if_then_else E] ⇒ let [ ⊢ref] = unique [⊢D] [⊢E] in  [⊢ref] )

  | [ ⊢e_succ D]        ⇒ ?
  | [ ⊢e_pred_zero]     ⇒ ?
  | [ ⊢e_pred_succ V]   ⇒ ?
  | [ ⊢e_pred D]        ⇒ ?
  | [ ⊢e_iszero_zero]   ⇒ ?
  | [ ⊢e_iszero_succ V] ⇒ ?
  | [ ⊢e_iszero D]      ⇒ ?
  ;
```

Let us consider the case where s1 is a derivation ending in [⊢e_if_true], i.e. s1 stands for a derivation of [⊢step (if_then_else true N1 N2) N1]. At this point we know that s2 stands for a derivation [⊢step (if_then_else true N1 N2) M2]. Splitting s2 into cases gives us two sub-cases:

1. We have used the rule e_if_true. In this case, we learn that M2 = N1. Clearly, we can conclude [⊢equal N1 N1] by using [⊢ref] as a witness.

2. We have used the rule e_if_then_else. In this case, we have a sub-derivation D that stands for [⊢step true N'] and M2 = (if_then_else N' N1 N2). We now use the lemma values_dont_step passing [⊢D] and [⊢v_true] (a witness for [⊢value true]. We therefore obtain an object of type [⊢not_possible]; but no elements of this type exist. This case is hence impossible.

Next, consider the case where s1 is a derivation ending in [⊢e_if_then_else D], i.e. s1 has type [⊢step (if_then_else N N1 N2) (if_then_else N' N1 N2)]. Hence D stands for the sub-derivation [⊢step N N']. At this point we know that s2 stands for a derivation [⊢step (if_then_else N N1 N2) M2]. Splitting s2 into cases gives us three sub-cases:

1. We have used the rule e_if_true. As a consequence N = true and M2 = N1. Moreover, D now stands for the sub-derivation [⊢step true N']. Using again the lemma values_dont_step, we show that this is impossible.

2. We have used the rule e_if_false. This case is similar to the above.

3. We have used the rule `e_if_then_else` and `s2` stands for the derivation tree `[⊢e_if_then_else E]` where `E` stands for a sub-derivation `[⊢step N N'']` and `M2 = (if_then_else N' N1 N2)`. We now make a recursive call on the sub-derivations `D` and `E`, i.e. `unique [⊢D] [⊢E]`, which gives us a witness for `[⊢equal N N']`. By inversion using `ref`, we learn that `N = N'`. To conclude the proof we need to provide a witness for `[⊢equal (if_then_else N N1 N2) ( if_then_else N N1 N2)]`. This is easily accomplished by `[⊢ref]`.

   It might look like we should be able to simply make a recursive call and be done. This is however a fallacy, since the type is incorrect. Recall that `ref` takes in an implicit argument for the term we are actually comparing; therefore in the first occurrence `[⊢ref]` stands actually for `[⊢ref N]`, while in the second occurrence it stands for `[⊢ref (if_then_else N N1 N2)]`.

3.3 Termination of Well-Typed Terms

Our goal is to prove that the evaluation of well-typed terms halts. In fact we already proved progress, i.e. evaluation cannot get stuck on well-typed terms, i.e. either a well-typed term yields a value or we can take another step. In this section we prove that we can always evaluate a well-typed term to a final value.

Theorem 3.3.1. *If* $\mathcal{D} :: \mathsf{M} : \mathsf{T}$ *then* M halts*, i.e. there exists a value* V *s.t.* $\mathsf{M} \longrightarrow^* \mathsf{V}$.

We recap our definition of multi-step relations that was the reflexive, transitive closure over the single step relation.

$$\frac{}{\mathsf{M} \longrightarrow^* \mathsf{M}}\ \text{M-REF} \qquad \frac{\mathsf{M} \longrightarrow^* \mathsf{N} \quad \mathsf{N} \longrightarrow^* \mathsf{M}'}{\mathsf{M} \longrightarrow^* \mathsf{M}'}\ \text{M-TR} \qquad \frac{\mathsf{M} \longrightarrow \mathsf{M}'}{\mathsf{M} \longrightarrow^* \mathsf{M}'}\ \text{M-STEP}$$

Evaluation of a term M may clearly not yield a value in one step; in fact we may need to chain multiple steps together. In the proof for showing that well-typed terms terminate, we will see the need for lemmas that justify bigger steps when we evaluate a term.

Lemma 3.3.2 (Multi Step Lemmas)**.**

1. *If* $\mathsf{M} \longrightarrow^* \mathsf{M}'$ *then* $(\textit{pred}\ \mathsf{M}) \longrightarrow^* (\textit{pred}\ \mathsf{M}')$.
2. *If* $\mathsf{M} \longrightarrow^* \mathsf{M}'$ *then* $(\textit{succ}\ \mathsf{M}) \longrightarrow^* (\textit{succ}\ \mathsf{M}')$.
3. *If* $\mathsf{M} \longrightarrow^* \mathsf{M}'$ *then* $(\textit{iszero}\ \mathsf{M}) \longrightarrow^* (\textit{iszero}\ \mathsf{M}')$.

4. *If* $M \longrightarrow^* M'$ *then* $(\textit{if}\ M\ \textit{then}\ M_1\ \textit{else}\ M_2) \longrightarrow^* (\textit{if}\ M'\ \textit{then}\ M_1\ \textit{else}\ M_2)$.

Proof. By structural induction on $\mathcal{S} :: M \longrightarrow^* M'$. We'll only the last proposition. By definition of $\longrightarrow^*$, we consider three cases:

Base case $\quad \mathcal{S} = \dfrac{M \longrightarrow M'}{M \longrightarrow^* M'}\,\textsc{M-Step}$

$$\begin{array}{lr}
\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \longrightarrow \text{if } M_1' \text{ then } M_2 \text{ else } M_3 & \text{by rule } \textsc{E-If} \\
\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \longrightarrow^* \text{if } M_1' \text{ then } M_2 \text{ else } M_3 & \text{by rule } \textsc{M-Step}
\end{array}$$

Base case $\quad \mathcal{S} = \dfrac{}{M_1 \longrightarrow^* M_1}\,\textsc{M-Ref}$

$$\begin{array}{lr}
\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \longrightarrow^* \text{if } M_1' \text{ then } M_2 \text{ else } M_3 & \text{by rule } \textsc{M-Ref}
\end{array}$$

Step case $\quad \mathcal{S} = \dfrac{\overset{\mathcal{S}_1}{M_1 \longrightarrow^* N} \qquad \overset{\mathcal{S}_2}{s \longrightarrow^* M_1'}}{M_1 \longrightarrow^* M_1'}\,\textsc{M-Tr}$

$$\begin{array}{lr}
\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \longrightarrow^* \text{if } N \text{ then } M_2 \text{ else } M_3 & \text{by I.H. on } \mathcal{S}_1 \\
\text{if } s \text{ then } M_2 \text{ else } M_3 \longrightarrow^* \text{if } M_1' \text{ then } M_2 \text{ else } M_3 & \text{by I.H. on } \mathcal{S}_2 \\
\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \longrightarrow^* \text{if } M_1' \text{ then } M_2 \text{ else } M_3 & \text{by rule } \textsc{M-Tr}
\end{array}$$

□

How would we mechanize this proof in Beluga? - This may seem straightforward, but there are some subtleties. In the last statement, we say

$$\text{If } M \longrightarrow^* M' \text{ then } (\text{if } M \text{ then } M_1 \text{ else } M_2) \longrightarrow^* (\text{if } M' \text{ then } M_1 \text{ else } M_2).$$

It is important to realize that M, M', M_1, and M_2 are all universally quantified. When we appeal to the induction hypothesis on $\mathcal{S}_1$ we in fact instantiate and choose the appropriate M_2 and M_3. It might be more precise to rewrite the statement to make this clearer.

$$\begin{gathered}
\text{If } t \longrightarrow^* M' \text{ then for all } M_1 \text{ and } M_2, \\
(\text{if } t \text{ then } M_1 \text{ else } M_2) \longrightarrow^* (\text{if } M' \text{ then } M_1 \text{ else } M_2).
\end{gathered}$$

Mechanizing proofs highlights such subtleties and forces us to understand them and be more precise than in the on paper proof. We can now translate the statement and the proof into BELUGA straitforwardly. We make explicit the quantification in the statement writing `{M2:[⊢ term}{M3:[⊢ term}`. In the program, we use **mlam**-abstraction as the corresponding proof term for introducing a universal quantifier.

```
rec mstep_if_then_else :  [ ⊢ multi_step M M’] →
   {M1:[ ⊢ term]}{M2:[ ⊢ term]}
   [ ⊢ multi_step (if_then_else M M1 M2) (if_then_else M’ M1 M2)] =
/ total ms (mstep_if_then_else _ _ ms)/
fn ms ⇒ case ms of
| [ ⊢ ms_ref]      ⇒ mlam M1 ⇒ mlam M2 ⇒ [ ⊢ ms_ref]
| [ ⊢ ms_step S]   ⇒ mlam M1 ⇒ mlam M2 ⇒ [ ⊢ ms_step (e_if_then_else S)]
| [ ⊢ ms_tr S1 S2] ⇒ mlam M1 ⇒ mlam M2 ⇒
  let [ ⊢ S1’] = mstep_if_then_else [ ⊢ S1] [ ⊢ M1 ] [ ⊢ M2 ] in
  let [ ⊢ S2’] = mstep_if_then_else [ ⊢ S2] [ ⊢ M1 ] [ ⊢ M2 ] in
    [ ⊢ ms_tr S1’ S2’]
;
```

Let us return to the goal of this section, namely of proving that the evaluation of well-typed terms terminates. To prove this statement, we need an additional lemma which justifies that multi-step relations preserve types.

Lemma 3.3.3 (Type preservation for multi-step relation)**.** *If* $\mathsf{M} : \mathsf{T}$ *and* $\mathsf{M} \longrightarrow^* \mathsf{M}'$ *then* $\mathsf{M}' : \mathsf{T}$.

Proof. By structural induction on $\mathsf{M} \longrightarrow^* \mathsf{M}'$. □

Finally, we are ready to consider the proof that evaluation of well-typed terms terminates. We first define M halts as follows:

$$\frac{\mathsf{M} \longrightarrow^* \mathsf{V} \quad \mathsf{V}\ \mathsf{value}}{\mathsf{M}\ \mathsf{halts}}$$

Note that we are encoding existential quantification in the proposition “there exists a value V s.t. $\mathsf{M} \longrightarrow^* \mathsf{V}$ in the theorem by defining a separate judgment M halts. This trick allows us to turn an existential into a universal quantifier, as the rule can be read as: For all M, V, if $\mathsf{M} \longrightarrow^* \mathsf{V}$ and V value then M halts.

Proof. By structural induction on $\mathcal{D} :: \mathsf{M} : \mathsf{T}$. We show a few representative cases.

Case $\mathcal{D} = \dfrac{}{\mathsf{z} : \mathsf{Nat}}\ \text{T-ZERO}$

z value   by V-ZERO rule

$\mathsf{z} \longrightarrow^* \mathsf{z}$ by $\textsc{M-Ref}$

z value by definition $\textsc{V-Zero}$

z halts by definition of halts

Case $$\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\mathsf{N} : \mathsf{Nat}}}{(\mathsf{pred}\ \mathsf{N}) : \mathsf{Nat}}\ \textsc{T-Pred}$$

N halts, i.e. $\exists \mathsf{V}$. s.t. $\mathcal{V}' : \mathsf{V}$ value and $\mathcal{S}' : \mathsf{N} \longrightarrow^* \mathsf{V}$ by I.H. $\mathcal{D}'$

$\boxed{\text{To Prove: } \mathsf{M} \text{ halts, i.e.} \exists W \text{.s.t.} W \text{ value and } \mathcal{S} : (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* W}$

Subcase $\mathcal{V}' = \dfrac{}{\mathsf{z}\ \text{value}}\ \textsc{V-Zero}$ and $\mathsf{V} = \mathsf{z}$

$\mathcal{S}' :: \mathsf{N} \longrightarrow^* \mathsf{z}$ restating assumption $\mathcal{S}'$

$\mathcal{S}_0 :: (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* (\mathsf{pred}\ \mathsf{z})$ by lemma mstep-pred

$\mathcal{S}_1 :: (\mathsf{pred}\ \mathsf{z}) \longrightarrow^* \mathsf{z}$ by $\textsc{M-Step}$ using $\textsc{E-PredZero}$

$\mathcal{S} :: (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* \mathsf{z}$ by $\textsc{M-Tr}$ using $\mathcal{S}_0$ and $\mathcal{S}_1$

$\exists W$.s.t.W value and $\mathcal{S} : (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* W$ by choosing $W = \mathsf{z}$

$(\mathsf{pred}\ \mathsf{N})$ halts by definition of halts

Subcase $\mathcal{V}' = \dfrac{\overset{\mathcal{W}}{\mathsf{V}'\ \text{value}}}{(\mathsf{succ}\ \mathsf{V}')\ \text{value}}\ \textsc{V-Succ}$ and $\mathsf{V} = \mathsf{succ}\ \mathsf{V}'$

$\mathcal{S}' :: \mathsf{N} \longrightarrow^* (\mathsf{succ}\ \mathsf{V}')$ restating assumption $\mathcal{S}'$

$\mathcal{S}_0 :: (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* (\mathsf{pred}\ (\mathsf{succ}\ \mathsf{V}'))$ by lemma mstep-pred

$\mathcal{S}_1 :: (\mathsf{pred}\ (\mathsf{succ}\ \mathsf{V}')) \longrightarrow^* \mathsf{V}'$ by $\textsc{M-Step}$ using $\textsc{E-PredSucc}$ and $\mathcal{W}$

$\mathcal{S} :: (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* \mathsf{V}'$ by $\textsc{M-Tr}$ using $\mathcal{S}_0$ and $\mathcal{S}_1$

$\exists W$.s.t.W value and $\mathcal{S} : (\mathsf{pred}\ \mathsf{N}) \longrightarrow^* W$ by choosing $W = \mathsf{V}'$

$(\mathsf{pred}\ \mathsf{N})$ halts by definition of halts

Subcase $\mathcal{V}' = \dfrac{}{\mathsf{true}\ \text{value}}\ \textsc{V-True}$ and $\mathsf{V} = \mathsf{true}$

$\mathcal{S}' :: \mathsf{N} \longrightarrow^* \mathsf{true}$ restating assumption $\mathcal{S}'$

$\mathcal{F} :: \mathsf{true} : \mathsf{Nat}$ by type preservation for multi-step relations using $\mathcal{D}'$

$\bot$

Subcase $\mathcal{V}' = \dfrac{}{\mathsf{false}\ \text{value}}\ \textsc{V-False}$ and $\mathsf{V} = \mathsf{false}$

Similar to the case where V = true.

□

We now discuss how to mechanize this proof as a program. As a first step, we must encode the statement of the theorem as a type.

```
LF halts: term → type =
| result: multi_step M V → value V
        → halts M;

rec terminate : [⊢ hastype M T] → [ ⊢ halts M] =
/ total d (terminate m t d)/
fn d ⇒ ? ;
```

Again we encode the case analysis in the proof as a case analysis in the program splitting on the assumption d. We show below the cases we discussed in detail above, however the full proof is implemented in the file evaluation.bel.

For the case where we have d:[⊢ hastype z nat] by [⊢t_zero], we return [⊢result ms_ref (v_zero)] that stands for a proof [⊢ halts z]. For the case where we have d:[⊢hastype (pred N) nat] by [⊢t_pred D] and D stands for a sub-derivation [⊢ hastype N nat]. By the induction hypothesis on D (i.e. modelled via the recursive call), we obtain a proof that [⊢ halts N]. By inversion, we know that this proof has the following shape: [⊢result MS W] where MS stands for [⊢multistep N R] and W stands for a proof [⊢value R]. We now case-analyze [⊢value R].

If R=z and we have a derivation [⊢v_zero], we call our lemma mstep_pred with MS to obtain a derivation MS' for [⊢multi_step (pred N) (pred z)]. What remains is to build a proof for [⊢halts (pred N)]. First, we build a proof [⊢ v_zero] that [⊢value z]. Second, we build a proof for [⊢multi_step (pred N) z] using transitivity together with MS' and the derivation [⊢ms_step e_pred_zero] for [⊢multi_step (pred N) z].

If R=succ W and we have a derivation (v_succ V), we call our lemma mstep_pred with MS to obtain a derivation MS' for [⊢multi_step (pred N) (pred (succ W))]. What remains is to build a proof for [⊢halts (pred N)] using transitivity together with MS' and the derivation [⊢ms_step (e_pred_succ V)].

If R=true or R=false then we know by the type preservation lemma for multi-step relations implemented by the function multi_tps that we cannot take a step.

```
rec terminate : [⊢hastype M T] → [ ⊢halts M] =
/ total d (terminate m t d)/
fn d ⇒ case d of
| [ ⊢t_true]  ⇒ ?
| [ ⊢t_false] ⇒ ?
| [ ⊢t_if_then_else D D1 D2] ⇒ ?
| [ ⊢t_zero] ⇒ [ ⊢result ms_ref v_zero]
| [ ⊢t_succ D] ⇒ ?
| [ ⊢t_pred D] ⇒ (case terminate [ ⊢D ] of
   | [ ⊢result MS (v_zero)] ⇒
     let [ ⊢MS’]         = mstep_pred [ ⊢MS] in
       [ ⊢result (ms_tr MS’ (ms_step e_pred_zero)) v_zero]
   | [ ⊢result MS (v_succ V)] ⇒
     let [ ⊢MS’]         = mstep_pred [ ⊢MS] in
       [ ⊢result (ms_tr MS’ (ms_step (e_pred_succ V)))  V]
   | [ ⊢result MS v_true] ⇒ impossible multi_tps [⊢D] [⊢MS]
   | [⊢result MS v_false] ⇒ impossible multi_tps [⊢D] [⊢MS]
 )
| [⊢t_iszero D] ⇒ ? ;
```

3.4 Relating Small-Step and Big-Step Semantics

3.4.1 Definition of the Big-Step semantics

Two styles of operational semantics are in common use. The one used so far and most commonly used in Pierce [2002] is called the *small-step* style, because the definition of the evaluation relation shows how individual steps of computation are used to rewrite a term, bit by bit, until it eventually becomes a value. On top of this, we define a multi-step evaluation relation that allows us to talk about terms evaluating (in many steps) to values. An alternative style, called *big-step* semantics (or sometimes *natural* semantics), directly formulates the notion of “the term M evaluates to the final value V”, written $M \Downarrow V$. The big-step evaluation rules for our language of boolean and arithmetic expressions can be described as follows:

$\boxed{M \Downarrow V}$: Term M evaluates to value V

$$\frac{V\ \mathsf{value}}{V \Downarrow V}\ \textsc{B-Value} \qquad \frac{M \Downarrow V}{\mathsf{succ}\ M \Downarrow \mathsf{succ}\ V}\ \textsc{B-Succ}$$

$$\frac{M \Downarrow \mathsf{z}}{\mathsf{pred}\ M \Downarrow \mathsf{z}}\ \textsc{B-PredZero} \qquad \frac{M \Downarrow \mathsf{succ}\ V}{\mathsf{pred}\ M \Downarrow V}\ \textsc{B-PredSucc}$$

$$\frac{M \Downarrow \mathsf{z}}{\mathsf{iszero}\ M \Downarrow \mathsf{true}}\ \textsc{B-IszeroZero} \qquad \frac{M \Downarrow \mathsf{succ}\ V}{\mathsf{iszero}\ M \Downarrow \mathsf{false}}\ \textsc{B-IszeroSucc}$$

$$\frac{M_1 \Downarrow \mathit{true} \quad M_2 \Downarrow V}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \Downarrow V}\ \textsc{B-IfTrue} \qquad \frac{M_1 \Downarrow \mathit{false} \quad M_3 \Downarrow V}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \Downarrow V}\ \textsc{B-IfFalse}$$

Following the previous principle of encoding judgments such as M ⇓ V using predicates (type families), we define in BELUGA the relation bigstep between two terms using an LF type family. In the rules, we do not enforce that we always return a value. Rather this is a property we can prove and often referred to as value soundness.

```
LF bigstep : term → term → type =
   | b_value       : value V
                     → bigstep V V
   | b_succ        : bigstep M V
                     → bigstep (succ M) (succ V)

   | b_pred_zero   : bigstep M z
                     → bigstep (pred M) z
   | b_pred_succ   : bigstep M (succ V)
                     → bigstep (pred M) V

   | b_iszero_zero : bigstep M z
                     → bigstep (iszero M) true
   | b_iszero_succ : bigstep M (succ V)
                     → bigstep (iszero M) false

   | b_if_true     : bigstep M1 true  → bigstep M2 V
                     → bigstep (if_then_else M1 M2 M3) V
   | b_if_false    : bigstep M1 false → bigstep M3 V
                     → bigstep (if_then_else M1 M2 M3) V
;
```

3.4.2 Value soundness

We will now prove two interesting properties about values in the big-step semantics. First, we prove that an expression always evaluates to a value:

Lemma 3.4.1 (Value Soundness for the Big-step semantics).
If M ⇓ V*, then* V *is a value.*

```
rec bstep_value : [⊢ bigstep M V] → [⊢ value V] =
/ total b (bstep_value _ _ b) /
  fn b ⇒ case b of
  | [⊢ b_value E] ⇒ [⊢ E]
  | [⊢ b_succ B]  ⇒ let [⊢ E] = bstep_value [⊢ B] in [⊢ v_succ E]

  | [⊢ b_pred_zero B] ⇒ bstep_value [⊢ B]
  | [⊢ b_pred_succ B] ⇒ let [⊢ v_succ E] = bstep_value [⊢ B] in [⊢ E]

  | [⊢ b_iszero_zero B] ⇒ [⊢ v_true]
```

```
   | [⊢ b_iszero_succ B] ⇒ [⊢ v_false]

   | [⊢ b_if_true  B1 B2] ⇒ bstep_value [⊢ B2]
   | [⊢ b_if_false B1 B3] ⇒ bstep_value [⊢ B3]
;
```

Second, we prove that evaluation yields a unique value. Note that this is **not** equivalent to saying that only the B-VALUE rule can apply. For example there are two different ways of proving that succ true ⇓ succ true:

$$\dfrac{\dfrac{\dfrac{}{\text{true value}}\ \text{V-TRUE}}{\text{succ true value}}\ \text{V-SUCC}}{\text{succ true} \Downarrow \text{succ true}}\ \text{B-VALUE} \qquad \dfrac{\dfrac{\dfrac{}{\text{true value}}\ \text{V-TRUE}}{\text{true} \Downarrow \text{true}}\ \text{B-VALUE}}{\text{succ true} \Downarrow \text{succ true}}\ \text{B-SUCC}$$

What we prove here for the big-step semantics is the same property as the "values don't step" for the small-step semantics, except that in our case, values can continue evaluating over and over. Values always evaluate to themselves.

Lemma 3.4.2. *If* $V_1 \Downarrow V_2$ *with* V_1 *a value, then* $V_1 = V_2$.

```
rec bstep_value_stagnate : [⊢ bigstep V1 V2] → [⊢ value V1] → [⊢ equal V1 V2]
   =
/ total e (bstep_unique_stagnate _ _ _ e) /
  fn b ⇒ fn e ⇒ case e of
  | [⊢ v_zero]    ⇒ let [⊢ b_value _] = b in [⊢ ref]
  | [⊢ v_true]    ⇒ let [⊢ b_value _] = b in [⊢ ref]
  | [⊢ v_false]   ⇒ let [⊢ b_value _] = b in [⊢ ref]
  | [⊢ v_succ E'] ⇒
    (case b of
     | [⊢ b_value _] ⇒ [⊢ ref]
     | [⊢ b_succ B'] ⇒
       let [⊢ ref] = bstep_unique_stagnate [⊢ B'] [⊢ E'] in [⊢ ref]
    )
;
```

We now prove that the small-step and big-step semantics for this language coincide, *i.e.* $M \longrightarrow^* V$ if and only if $M \Downarrow V$, for any value V. We will prove the two directions of the "if and only if" separately.

3.4.3 Part 1: If $M \Downarrow V$ then $M \longrightarrow^* V$

Note that in this direction, we do not need to state that V is a value, since the Value Soundness Lemma (Lemma 3.4.1) gives us this property for free.

This part of the proof follows exactly the one described in Pierce [2002]. In the proof, we rely on the Multi Step Lemmas (Lemma 3.3.2) which we proved in the last section. These lemmas justify bigger steps when evaluating a term with the small-steps semantics which will be useful when we translate a sequence of steps to a derivation tree in our big-step semantics. Let us now write the actual proof:

Proposition 3.4.3 (Big-step to small-step)**.** *If* $M \Downarrow V$ *then* $M \longrightarrow^* V$.

Proof. By induction on the proof $\mathcal{B} :: M \Downarrow V$:

$$\textbf{Case}\quad \mathcal{B} = \dfrac{V\ \mathsf{value}}{V \Downarrow V}\ \textsc{B-Value}$$

$V \longrightarrow^* V$ by reflexivity of $\longrightarrow^*$, i.e. $\textsc{M-Ref}$

$$\textbf{Case}\quad \mathcal{B} = \dfrac{\begin{array}{c}\mathcal{B}' \\ M \Downarrow \mathsf{z}\end{array}}{\mathsf{pred}\ M \Downarrow \mathsf{z}}\ \textsc{B-PredZero}$$

$M \longrightarrow^* \mathsf{z}$ by i.h. on $\mathcal{B}'$
$\mathsf{pred}\ M \longrightarrow^* \mathsf{pred}\ z$ by lemma 3.3.2
$\mathsf{pred}\ \mathsf{z} \longrightarrow \mathsf{z}$ by $\textsc{E-PredZero}$
$\mathsf{pred}\ \mathsf{z} \longrightarrow^* \mathsf{z}$ by $\textsc{M-Step}$
$\mathsf{pred}\ M \longrightarrow^* \mathsf{z}$ by $\textsc{M-Tr}$

$$\textbf{Case}\quad \mathcal{B} = \dfrac{\begin{array}{c}\mathcal{B}_1 \\ M_1 \Downarrow \mathsf{true}\end{array} \qquad \begin{array}{c}\mathcal{B}_2 \\ M_2 \Downarrow V\end{array}}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \Downarrow V}\ \textsc{B-IfTrue}$$

$M_1 \longrightarrow^* \mathsf{true}$ by i.h. using $\mathcal{B}_1$
$M_2 \longrightarrow^* V$ by i.h. using $\mathcal{B}_2$
$\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow^* \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3$ by lemma 3.3.2
$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow M_2$ by $\textsc{E-IfTrue}$
$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow^* M_2$ by $\textsc{M-Step}$
$\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow^* V$ by $\textsc{M-Tr}$ □

Encoding this proof is in fact straightforward. We translate the theorem statement into a type and the inductive proof into a recursive program.

```
rec bstep_to_mstep : [⊢ bigstep M V] → [⊢ multi_step M V] =
/ total b (bstep_to_mstep _ _ b) /
  fn b ⇒ case b of
```

```
| [⊢ b_value _] ⇒ [⊢ ms_ref]

| [⊢ b_pred_zero B] ⇒
  let [⊢ S] = bstep_to_mstep [⊢ B] in
  let [⊢ S’] = mstep_pred [⊢ S] in
  [⊢ ms_tr S’ (ms_step e_pred_zero)]

| [⊢ b_if_true  B1 B2] : [⊢ bigstep (if_then_else _ M2 M3) _] ⇒
  let [⊢ S1] = bstep_to_mstep [⊢ B1] in
  let [⊢ S2] = bstep_to_mstep [⊢ B2] in
  let [⊢ S1’] = mstep_if_then_else [⊢ S1] [⊢ M2] [⊢ M3] in
  [⊢ ms_tr (ms_tr S1’ (ms_step e_if_true)) S2]
```

There is however one very interesting aspect that arises in the translation of the on-paper proof into the program. Revisit the line

```
| [⊢ b_if_true  B1 B2] : [⊢ bigstep (if_then_else _ M2 M3) _] ⇒
```

We give a type annotation to the pattern. This is not necessary for type reconstruction, but rather we want to be able to name the term `M2` and `M3` as we want to and must pass them explicitly when we use the lemma `mstep_if_then_else`. Type annotations can also be used in let-expressions and alternatively, we could have written the following:

```
| [⊢ b_if_true  B1 B2] ⇒
   let [⊢ B1] : [⊢ bigstep M2 V] = [⊢ B1] in
   let [⊢ B]  : [⊢ bigstep (if_then_else _ M2 M3) _]  = b in
   let [⊢ S1] = bstep_to_mstep [⊢ B1] in
   let [⊢ S2] = bstep_to_mstep [⊢ B2] in
   let [⊢ S1’] = mstep_if_then_else [⊢ S1] [⊢ M2] [⊢ M3] in
   [⊢ ms_tr (ms_tr S1’ (ms_step e_if_true)) S2]
```

3.4.4 Part 2: If $M \longrightarrow^* V$ with V a value then $M \Downarrow V$

This direction is not as immediate as the first one. This comes essentially from the fact that the big-step semantics relates a term and a value, while the small step semantics relates two terms. In this proof, we exploit the fact that eventually we also reach a value using the small-step semantics and moreover that in both, the small step and big step semantics, values evaluates to itself.

Lemma 3.4.4. *If* $M \longrightarrow N$ *and* $N \Downarrow V$ *then* $M \Downarrow V$.

The statement of the lemma can be directly translated into BELUGA.

```
rec step_bstep_to_bstep : [⊢ step M N] → [⊢ bigstep N V] → [⊢ bigstep M V] =
/ total s (step_bstep_to_bstep _ _ _ s _) /
  fn s ⇒ fn b ⇒ ?;
```

The proof of this lemma is the hardest part. Indeed, once we have this lemma, we will only have to iterate over the $\longrightarrow$ to get the same property for $\longrightarrow^*$:

Lemma 3.4.5. *If* $M \longrightarrow^* N \Downarrow V$ *then* $M \Downarrow V$*.*

This lemma is translated into BELUGA directly as

```
rec mstep_bstep_to_bstep : [⊢ multi_step M N] → [⊢ bigstep N V]
                           → [⊢ bigstep M V] =
/ total s (mstep_bstep_to_bstep _ _ _ s _) /
  fn s ⇒ fn b ⇒ ?;
```

Finally we will just have to apply this lemma with $N = V$ to obtain what we want:

Proposition 3.4.6. *If* $M \longrightarrow^* V$ *with* V *a value then* $M \Downarrow V$*.*

Or in BELUGA:

```
rec mstep_to_bstep : [⊢ multi_step M V] → [⊢ value V] → [⊢ bigstep M V] =
/ total s (mstep_to_bstep _ _ s _) /
  fn s ⇒ fn v ⇒ ? ;
```

Let us now return to the proof of Lemma 3.4.4:

Proof of lemma 3.4.4. By induction on the proof $\mathcal{S} :: M \longrightarrow N$.

Case $\mathcal{S} = \dfrac{}{\text{if true then } M_2 \text{ else } M_3 \longrightarrow M_2}\ \textsc{E-IfTrue}$

$M_2 \Downarrow V$ by assumption
$\text{true} \Downarrow \text{true}$ by B-VALUE

$$\dfrac{\dfrac{\dfrac{}{\text{true value}}\ \textsc{V-True}}{\text{true} \Downarrow \text{true}}\ \textsc{B-Value} \qquad M_2 \Downarrow V}{\text{if true then } M_2 \text{ else } M_3 \Downarrow V}\ \textsc{B-IfTrue}$$

Case $\mathcal{S} = \dfrac{\mathsf{V\ value}}{\mathsf{pred\ succ\ } V \longrightarrow V}\ \textsc{E-PredSucc}$

$V \Downarrow V'$ by assumption
$V = V'$ by Lemma 3.4.2

$$\dfrac{\dfrac{\dfrac{V\ \mathsf{value}}{\mathsf{succ}\ V\ \mathsf{value}}\ \textsc{V-Succ}}{\mathsf{succ}\ V \Downarrow \mathsf{succ}\ V}\ \textsc{B-Value}}{\mathsf{pred}\ (\mathsf{succ}\ V) \Downarrow V}\ \textsc{B-PredSucc}$$

Case $\mathcal{S} = \dfrac{M_1 \longrightarrow M_1'}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \longrightarrow \mathsf{if}\ M_1'\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3}\ \textsc{E-If}$

Let's notice that after such a step, there can be three big-step rules:

- A B-VALUE rule, but this is not possible since a $\mathsf{if}\ M_1'\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3$ cannot be a value.
- A B-IFTRUE rule. In that case, we have a proof for $M_1' \Downarrow \mathsf{true}$ and an other one for $M_2 \Downarrow V$. We can then apply our induction hypothesis on $M_1 \longrightarrow M_1' \Downarrow \mathsf{true}$, obtain a proof for $M_1 \Downarrow \mathsf{true}$ and conclude:

$$\dfrac{M_1 \Downarrow \mathsf{true} \qquad M_2 \Downarrow V}{\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \Downarrow V}\ \textsc{B-IfTrue}$$

- A B-IFFALSE rule, which is exactly the same thing.

Case $\mathcal{S} = \dfrac{M \longrightarrow N}{\mathsf{succ}\ M \longrightarrow \mathsf{succ}\ N}\ \textsc{E-Succ}$

Once again, after this step, we can have two different big-step rules:

- A B-VALUE rule. Which means that $\mathsf{succ}\ N$ is a value, which means that N is a value. We can then use the induction hypothesis on $M \longrightarrow N \Downarrow N$ and get a proof of $M \Downarrow N$. And then conclude:

$$\dfrac{M \Downarrow N}{\mathsf{succ}\ M \Downarrow \mathsf{succ}\ N}\ \textsc{B-Succ}$$

- A B-Succ rule. Which means we have something like that:

  $$\dfrac{M \longrightarrow N \Downarrow V'}{\mathsf{succ}\ M \longrightarrow \mathsf{succ}\ N \Downarrow \mathsf{succ}\ V'}\ \textsc{E-Succ}and\textsc{B-Succ}$$

  We apply the induction hypothesis, obtain a proof of $M \Downarrow V'$ and we can conclude:

  $$\dfrac{M \Downarrow V'}{\mathsf{succ}\ M \Downarrow \mathsf{succ}\ V'}\ \textsc{B-Succ}$$

□

```
rec step_bstep_to_bstep : [⊢ step M N] → [⊢ bigstep N V] → [⊢ bigstep M V] =
/ total s (step_bstep_to_bstep _ _ _ s _) /
  fn s ⇒ fn b ⇒
  let [⊢ B] = b in
  case s of

  | [⊢ e_if_true] ⇒
    [⊢ b_if_true (b_value v_true) B]

  | [⊢ e_pred_succ E] ⇒
    let [⊢ ref] = bstep_values_stagnate [⊢ B] [⊢ E] in
    [⊢ b_pred_succ (b_value (v_succ E))]

  | [⊢ e_if_then_else S] ⇒
    (case [⊢ B] of
     | [⊢ b_value E] ⇒ impossible [⊢ E]
     | [⊢ b_if_true  B1’ B2’] ⇒
       let [⊢ B1] = step_bstep_to_bstep [⊢ S] [⊢ B1’] in
       [⊢ b_if_true  B1 B2’]
     | [⊢ b_if_false B1’ B3’] ⇒
       let [⊢ B1] = step_bstep_to_bstep [⊢ S] [⊢ B1’] in
       [⊢ b_if_false B1 B3’]
    )

  | [⊢ e_succ S] ⇒
    (case [⊢ B] of
     | [⊢ b_value E] ⇒
       let [⊢ v_succ E’] = [⊢ E] in
       let [⊢ B’] = step_bstep_to_bstep [⊢ S] [⊢ b_value E’] in
       [⊢ b_succ B’]
     | [⊢ b_succ B] ⇒
       let [⊢ B’] = step_bstep_to_bstep [⊢ S] [⊢ B] in
       [⊢ b_succ B’]
    )
```

Part II

Intermediate: Mechanizing Languages with Binders

Chapter 4

Variables, Binders, and Assumptions

So far we have considered the representation of a simple language with arithmetic expressions and booleans together with proofs about it. In this chapter, we grow this language to include variables, functions, and function applications, or more generally constructs that allow us to abstract over variables. We begin by considering a small fragment of the lambda-calculus where we define terms using variables, function abstraction, and function application. We refer the reader to for example [Pierce 2002, Ch 5, Ch 9] for a more detailed introduction.

$$\text{Terms} \quad M, N ::= x \mid \lambda x.M \mid M\ N$$

The main question we are interested here in is the following: How do we represent this grammar in an proof assistant or programming environment? – Clearly, we need to face the issue of how to represent variables.

The most straightforward answer to this question is to use a standard "named" representation of λ-terms, where variables are treated as labels or strings. In this approach, one has to explicitly handle α-conversion when defining any operation on the terms. The standard Variable Convention of Barendregt, which is often employed in on-paper proofs, is one such approach where α-conversion is applied as needed to ensure that:

(i) bound variables are distinct from free variables, and

(ii) all binders bind variables not already in scope.

In practice this approach is cumbersome, inefficient, and often error-prone. It has therefore led to the search for different representations of such terms. Many such approaches exist (see Aydemir et al. [1990] for a good overview), here we will focus on two of them:

- De Bruijn indices: As the name already indicates, this approach goes back to Nicolaas Govert de Bruijn who used it in the implementation of Automath. Automath was a formal language in the 60s, developed for expressing complete mathematical theories in such a way that an included automated proof checker can verify their correctness. De Bruijn indices are also fundamental to more advanced techniques such as *explicit substitutions* [Abadi et al. 1990].

- Higher-order abstract syntax [Pfenning and Elliott 1988]: an appeal to higher-order representations where the binders are modelled using functions. These functions are typically very weak: they only model the scope of variables in an abstract syntax tree, not allowing for recursion or pattern matching. In such representations, the issues of α-equivalence, substitution, etc. are identified with the same operations in a meta-logic.

It is worth pointing out that although we may prefer one of these two representations for modelling binders in a proof and programming environment, the named representation of λ-terms has one important advantage: it can be immediately understood by others because the variables can be given descriptive names. Thus, even if a system uses De Bruijn indexes internally, it will present a user interface with names.

4.1 De Bruijn Indices

De Bruijn's idea was that we can represent terms more straightforwardly and avoid issues related to α-renaming by choosing a canonical representation of variables. Variable occurrences point directly to their binders rather than referring to them by name. This is accomplished by replacing named variable by a natural number where the number k stands for “the variable bound by the k'th enclosing λ.

Here are some examples:

$\lambda x.x$	$\lambda\ 1$
$\lambda x.\lambda y.x\ y$	$\lambda\ \ \lambda\ 2\ 1$
$\lambda x.(\lambda y.x\ y)\ x$	$\lambda\ (\lambda\ 2\ 1)\ 1$

De Bruijn representations are common in compiler and theorem proving systems which rely on canonical representation of terms. They are however tedious to manage. In particular, the same variable may have different indices depending on where it occurs! This can make term manipulations extremely challenging.

We can define a grammar for de Bruijn terms more formally as

$$\begin{array}{lll} \text{Indices} & I & ::= 1 \mid \uparrow I \\ \text{De Bruijn Terms} & T, S & ::= I \mid \lambda\, T \mid T\ S \end{array}$$

The index 3 is represented as $\uparrow(\uparrow 1)$. The distinction between indices and de Bruijn Terms is not really necessary; often we see the following definition

$$\text{De Bruijn Terms} \quad T, S ::= 1 \mid \uparrow T \mid \lambda\, T \mid T\ S$$

This allows us to shift arbitrary de Bruijn terms. Intuitively, the meaning of shifting $\uparrow(\lambda\ 1\ 2)$ is that we increase by 1 all free variable indices in this term and the result would be $\lambda\ 1\ 3$. We leave out the exact definition of shifting arbitrary terms (see Pierce [2002] for details).

Let us first consider the translation between lambda-terms and their corresponding de Bruijn representation.

$\boxed{\Gamma \vdash M \leadsto T}$: Term M with the free variables in Γ is translated to the de Bruijn representation T

$$\frac{\Gamma, x \vdash M \leadsto S}{\Gamma \vdash \lambda x.M \leadsto \lambda\ S}\ \textsc{Tr-Lam} \qquad \frac{\Gamma \vdash M \leadsto T \quad \Gamma \vdash N \leadsto S}{\Gamma \vdash M\ N \leadsto T\ S}\ \textsc{Tr-App}$$

$$\frac{}{\Gamma, x \vdash x \leadsto 1}\ \textsc{Tr-Top} \qquad \frac{\Gamma \vdash x \leadsto I \quad y \neq x}{\Gamma, y \vdash x \leadsto \uparrow I}\ \textsc{Tr-Next}$$

If we translate a lambda-term in the context Γ to its de Bruijn representation, then the de Bruijn representation is in fact closed, i.e. it does not contain any variables declared from Γ.

To translate a de Bruijn term to a lambda-term, we accummulate variables in a context and we look up the position of a variable in it.

4.2 Higher-Order Abstract Syntax

In general, managing binders and bound variables is a major pain. So, several alternatives have been and are being developed.

4.2.1 Representing variables

In Beluga (as in Twelf and Delphin), we support higher-order abstract syntax : our foundation, called the logical framework LF [Harper et al. 1993] allows us to represent binders via binders in our data-language.

For example, we can declare a type `term` in Beluga, which has two constructors.

```
LF term : type =
| app : term  → term  → term
| lam : (term → term) → term
;
```

The constructor `app` takes in two arguments; both of them must be expressions. The constructor `lam` takes in one argument which is in fact a function! Note that for simplicity, we do not represent the type annotation on the function which is present in our grammar. Let's look at a few examples:

On Paper (Object language)	LF/Beluga (Meta-language)
$\lambda x.x$	`lam λx. x`
$\lambda x.\lambda y.x\ y$	`lam λx. lam λy. app x y`
$\lambda w.\lambda v.w\ v$	`lam λx.lam λy. app x y)`
$\lambda w.(\lambda v.v\ w)\ w$	`lam λx. (app (lam λv. app v x) x)`

Note that the type of `λx.x` is `exp → exp`. So, we represent binders via lambda-abstractions in our meta-language. This idea goes back to Church. One major advantage is that we push all α-renaming issues to the Beluga developer. It is not the user's business anymore to manipulate indices or α-convert terms manually; instead, the user inherits these properties directly from the meta-language. Of course, Beluga developers and implementors have to still battle with de Bruijn indices and all the issues around variables.

Why is this great for the user of Beluga (or any other such system such as Twelf, Delphin, Hybrid, etc): not only does this higher-order representation support α-renaming, but we also get substitution for free! Why? - The meta-language is itself a lambda-calculus, and as every lambda-calculus it comes with some core properties such as α-renaming and β-reduction. So, if we have `lam λx. lam λy. app x y` and we would like to replace `x` in `lam λy. app x y` with the term `lam λz.z`, then we simply say `(λy. lam λy. app x y) (lam λz.z)`, i.e. we apply the LF-abstraction `(λy. lam λy. app x y)` to an argument.

This will come in particularly handy when we are representing our small-step evaluation rules. Let us recall our rules for evaluating function application.

$$\boxed{M \longrightarrow M'} : \text{Term } M \text{ steps to term } M'$$

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \text{value}}{V\ N \longrightarrow V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \text{value}}{(\lambda x.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

To represent evaluation, we revisit our type family `step` and define three constructors, each one corresponding to one of the rules in the operational semantics. The representation for E-App2 and E-App1 follows the previous ideas and is straightforward. For representing the rule E-App-Abs, we take advantage of the fact that LF-functions (i.e. `M` in `lam M` has type `term → term` and denotes a LF-function!) can be applied to an argument. Hence, we can model the substitution $[V/x]M$ by simply writing `M V`.

```
LF step: term → term → type =
| e_app_1    : step M M’
             → step (app M N) (app M’ N)
| e_app_2    : step N N’ → value V
              → step (app V N) (app V N’)
| e_app_abs : value V
              → step (app (lam M) V) (M V)
;
```

We can then use these constructors `e_app_1`, `e_app_2`, and `e_app_abs` to build objects that correspond directly to derivations using the rules E-App2, E-App1, and E-App-Abs. This follows the same principles as in the previous chapter.

Exercises

Ex. 4.1 : Extend the language with a let-value-construct.

Ex. 4.2 : Extend the language with a let-name-construct.

Ex. 4.3 : Extend the language with a match-construct that pattern matches on numbers.

Ex. 4.4 : Extend the language with recursion.

Ex. 4.5 : Extend the proof for uniqueness of evaluation we developed in the previous Chapter in Section 3.2.

4.2.2 Representing assumptions

We now consider how to represent typing derivations. Recall that we can represent typing derivations with explicit contexts and without (i.e. Gentzen-style).

$$\boxed{M : T} \quad M \text{ has type } T \text{ (implicit contexts)}$$

$$\frac{\begin{array}{c}\overline{x : T}\ u \\ \vdots \\ M : S\end{array}}{\lambda x.M : T \to S}\ \textsc{T-Abs}^{x,u} \qquad \frac{M : T \to S \quad N : T}{M\ N : S}\ \textsc{T-App}$$

We call the rule T-Abs parametric in x and hypothetical in u. In the implicit context formulation, we simply reason directly from assumptions.

$$\frac{\dfrac{\dfrac{\overline{x : \mathsf{Nat} \to \mathsf{Nat}}\ u \quad \overline{y : \mathsf{Nat}}\ v}{x\ y : \mathsf{Nat}}\ \textsc{T-App}}{\lambda y.x\ y : \mathsf{Nat} \to \mathsf{Nat}}\ \textsc{T-Abs}^{y,v}}{\lambda x.\lambda y.x\ y : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}}\ \textsc{T-Abs}^{x,u}$$

As an alternative, we can re-state the rules using an explicit context for bookkeeping; this also is often useful when we want to state properties about our system and about contexts in particular. To make also the relationship between the term M and the type T more explicit, we re-formulate the previous typing rules using the judgment: $\Gamma \vdash \mathsf{hastype}\ M\ T$ which can be read as: "term M has type T in the context Γ."

$$\boxed{\Gamma \vdash \mathsf{hastype}\ M\ T} \quad \text{Term } M \text{ has type } T \text{ in the context } \Gamma \text{ (explicit context)}$$

$$\frac{u : \mathsf{hastype}\ x\ T \in \Gamma}{\Gamma \vdash \mathsf{hastype}\ x\ T}\ u \qquad \frac{\Gamma, x, u : \mathsf{hastype}\ x\ T \vdash \mathsf{hastype}\ M\ S}{\Gamma \vdash \mathsf{hastype}\ (\lambda x.M)\ (T \to S)}\ \textsc{T-Abs}^{x,u}$$

$$\frac{\Gamma \vdash \mathsf{hastype}\ M\ (T \to S) \quad \Gamma \vdash \mathsf{hastype}\ N\ T}{\Gamma \vdash \mathsf{hastype}\ (M\ N)\ S}\ \textsc{T-App}$$

It should be intuitively clear that these two formulations of the typing rules are essentially identical; while the first set of rules use a two-dimensional representation the second set of rules makes the context of assumptions explicit and provides an explicit rule for looking up variables.

When we encode typing rules as a data-type, the first formulation with implicit contexts is particularly interesting and elegant. Why? - Because, we can read the rule T-Abs in the implicit context formulation as follows: $\lambda x.M$ has type $T \to S$, if given a variable x and an assumption u that stands for $x : T$ we can show that M has type S, i.e. we can construct a derivation for $M : S$.

Note that “Given x and u, we can construct a derivation $M : S$” is our informal description of a function that takes x and u as input and returns a derivation $M : S$. This is a powerful idea, since viewing it as a function directly enforces that the scope of x and u is only in the derivation for $M : S$. It also means that if we prove a term N for x and $N : T$ for u, we should be able to return a derivation $M : S$ where every x has been replaced by N and every use of u has been replaced by the proof that $N : T$. As a consequence, the substitution lemma that we have proved for typing derivations can be obtained for free by simply applying the function that stands for ‘Given x and u, we can construct a derivation $M : S$”

Let’s make this idea concrete. We define the relation `hastype` as a type in LF follows:

```
LF hastype: term → tp → type =
| t_lam : ({x:term} hastype x T → hastype (M x) S)
        → hastype (lam M) (arr T S)
| t_app:  hastype M1 (arr T S) → hastype M2 T
        → hastype (app M1 M2) S
;
```

Note that the argument to the constructor `t_lam` must be of type `({x:term} hastype x T → hastype (M x) S)`. We write here curly braces for universal quantification expressing directly more formally the sentence “Given a variable `x` and an assumption `hastype x T`, we can construct `hastype (M x) S`.”

One might ask, why do we have to write `hastype (M x) S` and why can we not write `hastype M S`? - Let’s look carefully at the types for each of the arguments. We note that we wrote `(lam M)` and we also know that `lam` takes in one argument that has type `term → term`, i.e. it is an LF-function. Hence writing `hastype M S` would be giving you a type-error, since the relation `hastype` expects an object of type `term`, not of type `term → term`. But is `(M x)`? What does it correspond to in the informal rule? - In the informal rule, we required that x is new. It might have been clearer to not re-use the variable name x that was occurring bound in $\lambda x.M$. We restate our previous rule T-Abs where we make the possibly necessary renaming explicit below.

$$\frac{\begin{array}{c}\overline{y : T}^{\;u}\\ \vdots \\ [y/x]M : S\end{array}}{\lambda x.M : T \to S}\ \text{T-ABS}^{y,u}$$

Here we see that indeed we replace all occurrences of x in M with a new variable y. It is exactly this kind of renaming that is happening, when we write `hastype (M x) S`.

Let us revisit the typing derivation for $\lambda x.\lambda y.x\;y : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$.

$$\dfrac{\dfrac{\dfrac{\overline{x : (\mathsf{Nat} \to \mathsf{Nat})}^{\,u} \quad \overline{y : \mathsf{Nat}}^{\,v}}{x\;y : \mathsf{Nat}}\ \text{T-App}}{\lambda y.x\;y : \mathsf{Nat} \to \mathsf{Nat}}\ \text{T-Abs}^{y,v}}{\lambda x.\lambda y.x\;y : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}}\ \text{T-Abs}^{x,u}$$

How would we encode it? - First we translate the typing judgment to representation; then we construct an object of this type that will correspond to the typing derivation.

```
let d:[⊢ hastype (lam λx.lam λy.app x y) (arrow (arrow nat nat) (arrow nat nat))]
    =
   [⊢ t_lam λx.λu. t_lam λy.λv. t_app u v]
```

Chapter 5

Proofs by Induction - Revisited

5.1 Type Preservation

Let us revisit the type preservation proof for the functions and function application, in particular we concentrate on the case for abstractions.

Theorem 5.1.1. *If* $\mathcal{D} :: \ \vdash \mathsf{hastype}\ M\ T$ *and* $\mathcal{S} :: M \ \longrightarrow\ N$ *then* $\vdash N : T$.

Proof. By structural induction on the derivation $\mathcal{S} :: M \ \longrightarrow\ N$.

$$\textbf{Case}\quad \mathcal{S} = \frac{\overset{\mathcal{V}}{V\ \mathsf{value}}}{(\lambda x.M)\ V \ \longrightarrow\ [V/x]M}\ \text{E-APP-ABS}$$

$$\begin{array}{lll}
 & ::\ \vdash \mathsf{hastype}\ ((\lambda x.M)\ V)\ T & \text{by assumption} \\
\mathcal{D}_1 & ::\ \vdash \mathsf{hastype}\ (\lambda x.M)\ (S \to T) & \\
\mathcal{D}_2 & ::\ \vdash \mathsf{hastype}\ V\ S & \text{by inversion using rule T-APP} \\
\mathcal{D} & ::\ x, u : \mathsf{hastype}\ x\ S \vdash \mathsf{hastype}\ M\ T & \text{by inversion on } \mathcal{D}_1 \text{ using rule T-ABS} \\
 & ::\ \vdash \mathsf{hastype}\ [V/x]M\ T & \text{by substitution lemma using } V \text{ and } \mathcal{D}_2 \text{ in } \mathcal{D}.
\end{array}$$

□

The proof below reflects the structure of the proof. Case-analyzing `s` that stands for `[⊢ step M N]` yields three different cases.

The case where we have `[⊢ e_app_abs V]` corresponds directly to the case in the proof above that we wrote out explicitly where `V` corresponds to $\mathcal{V}$. We then use inversion to analyze our assumption `[⊢hastype (app (lam M) V) T]`. We have written the two inversion steps as one nested pattern in Beluga. More importantly, the subderivation $\mathcal{D} :: u : \mathsf{hastype}\ x\ S \vdash \mathsf{hastype}\ M\ T$ in the proof is represented as `λx.λu. D` where `D` has type `hastype (M x) T` in the context `x:term, u:hastype x S`.

Recall that earlier we remarked that the typing rule for functions does make two assumptions: that we have a fresh variable `x` and an assumption `hastype x S` which we call `u` here. In the proof we then replaced all occurrences of x by the value V and all assumptions $V : S$ are replaced by the proof $\mathcal{D}_2$.

$$\cfrac{\begin{array}{c}\mathcal{D}^{x,u}\\ x, u : \mathsf{hastype}\ x\ S \vdash \mathsf{hastype}\ M\ T\end{array}}{\vdash \mathsf{hastype}\ (\lambda x.M)\ (S \to T)}\ \text{T-ABS}^{x,u}$$

replacing x by V in $\mathcal{D}$ yields $\quad \begin{array}{c}[V/x]\mathcal{D}^{u}\\ u : \mathsf{hastype}\ V\ S \vdash \mathsf{hastype}\ ([V/x]M)\ T\end{array}$

replacing u by $\mathcal{D}_2$ in $\mathcal{D}$ yields $\quad \begin{array}{c}[\mathcal{D}_2/u, V/x]\mathcal{D}\\ \mathsf{hastype}\ ([V/x]M)\ T\end{array}$

In BELUGA where substitutions are first-class, we simply associate the derivation $\mathcal{D}$ with the substitution `[_, D2]`; the underscore stands for the value V whose name is not explicitly available in the program. Beluga's type reconstruction will however make sure that that the underscore is exactly the value `D2` refers to.

```
rec tps: [ ⊢ hastype M T] → [ ⊢ step M N] → [ ⊢ hastype N T] =
/ total s (tps m t n d s)/
fn d ⇒ fn s ⇒ case s of
| [ ⊢ e_app_1 S1] ⇒
  let [ ⊢ t_app D1 D2] = d in
  let [ ⊢ F1] = tps  [ ⊢ D1] [ ⊢ S1] in
    [ ⊢ t_app F1 D2 ]

| [ ⊢ e_app_2 S2 _ ] ⇒
  let [ ⊢ t_app D1 D2] = d in
  let [ ⊢ F2] = tps  [ ⊢ D2] [ ⊢ S2] in
    [ ⊢ t_app D1 F2]

| [ ⊢ e_app_abs V] ⇒
  let [ ⊢ t_app (t_lam λx.λu. D) D2] = d in
    [ ⊢ D[_,  D2]]
;
```

5.2 Type Uniqueness

We also sometimes prove properties that hold only for non-empty contexts. One such example is proving type uniqueness. In fact, this property does not hold unless we add some type annotations. So far we have been working with lambda-terms $\lambda x.M$. However, consider the identity function $\lambda x.x$ - it has many types, not just one. However, annotating the variable bound by a lambda-abstractions will be sufficient to guarantee that every lambda-term has a unique type.

We therefore also revise our definition of terms and typing rules slightly, highlighting the new parts in green. Finally we define type equality explicitly using reflexivity.

```
LF term : type =
| app : term  → term  → term
| lam : tp →(term → term) → term
;

LF hastype: term → tp → type =
| t_lam : ({x:term} hastype x T → hastype (M x) S)
        → hastype (lam T M) (arr T S)
| t_app:  hastype M1 (arr T S) → hastype M2 T
        → hastype (app M1 M2) S
;

LF eq: term → term → type =
| refl: eq  M M ;
```

Let us now revisit the proof of type uniqueness. Note that as we traverse abstractions, we are collecting assumptions about variables and their types. We are therefore not able to prove every term has a unique type in the empty context, but must state it more generally. To do so, we silently revert to an explicit context formulation of our typing rules, since this proves to be more convenient. To make the structure of the proof even more apparent, we already use our LF encoding to describe our typing judgments. This will also make the translation of this proof into a Beluga program much easier.

Theorem 5.2.1 (Type uniqueness).
If $\mathcal{D} :: \Gamma \vdash \mathsf{hastype}\ M\ T$ *and* $\mathcal{C} :: \Gamma \vdash \mathsf{hastype}\ M\ S$ *then* $\mathsf{eq}\ T\ S$.

Proof. By structural induction on the typing derivation $\mathcal{D} :: \Gamma \vdash \mathsf{hastype}\ M\ T$.

Case $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash \mathsf{hastype}\ M\ (\mathsf{arr}\ T\ S)} \qquad \overset{\mathcal{D}_2}{\Gamma \vdash \mathsf{hastype}\ N\ T}}{\Gamma \vdash \mathsf{hastype}\ (\mathsf{app}\ M\ N)\ S}\ \text{T-APP}$$

$$\mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma \vdash \mathsf{hastype}\ M\ (\mathsf{arr}\ T'\ S')} \qquad \overset{\mathcal{C}_2}{\Gamma \vdash \mathsf{hastype}\ N\ T'}}{\Gamma \vdash \mathsf{hastype}\ (\mathsf{app}\ M\ N)\ S'}\ \textsc{T-App}$$

$\mathcal{E} :: \mathsf{eq}\ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T\ {}'S')$ by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$
$\mathcal{E} :: \mathsf{eq}\ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T\ S)$ and $S = S'$ and $T = T'$ by inversion on reflexivity.

Therefore there is a proof for $\mathsf{eq}\ S\ S'$ by reflexivity (since we know $S = S'$).

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x, u : \mathsf{hastype}\ x\ T \vdash \mathsf{hastype}\ M\ S}}{\Gamma \vdash \mathsf{hastype}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \textsc{T-Abs}$

$$\mathcal{C} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x, u : \mathsf{hastype}\ x\ T \vdash \mathsf{hastype}\ M\ S'}}{\Gamma \vdash \mathsf{hastype}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \textsc{T-Abs}$$

$\mathcal{E} :: \mathsf{eq}\ S\ S'$ by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$
$\mathcal{E} :: \mathsf{eq}\ S\ S$ and $S = S'$ by inversion using reflexivity

Therefore there is a proof for $\mathsf{eq}\ (\mathsf{arr}\ T\ S)\ (\mathsf{arr}\ T\ S')$ by reflexivity.

Case $\mathcal{D} = \dfrac{x, u : \mathsf{hastype}\ x\ T \in \Gamma}{\Gamma \vdash \mathsf{hastype}\ x\ T}\ u \qquad \mathcal{C} = \dfrac{x, v : \mathsf{hastype}\ x\ S \in \Gamma}{\Gamma \vdash \mathsf{hastype}\ x\ S}\ v$

Every variable x is associated with a unique typing assumption (property of the context), hence $v = u$ and $S = T$.

□

There are a number of interesting observations we can make about this proof:

- We rely on the fact that every assumption is unique and there are not two assumptions about the same variable; this is in fact implicitly enforced in the rule T-Abs where we ensure that the variable is new.
- We extend our context in the rule T-Abs.

- We reason about equality using reflexivity. We note that by using our rule `refl`, we are able to learn that two types are actually the same (i.e. $T = T'$).
- We have an explicit variable (base) case, as we stated our judgments within a context Γ.

The encoding of this proof is in fact straightforward in Beluga thanks to the support provided. We first describe the shape (i.e. type) of our context using a *schema declaration*. Just as types classify terms, schemas classify contexts. We observe that in our typing rules, we always introduce a variable x and the assumption hastype x T at the same time. To denote that these two assumptions always come in pairs, we write the keyword **block**.

```
schema tctx = some [t:tp] block (x:term,u:hastype x t);
```

The schema `tctx` describes a context containing assumptions `x:term`, each associated with a typing assumption `hastype x t` for some type `t`. Formally, we are using a dependent product Σ (used only in contexts) to tie `x` to `hastype x t`. We thus do not need to establish separately that for every variable there is a unique typing assumption: this is inherent in the definition of `tctx`. The schema classifies well-formed contexts and checking whether a context satisfies a schema will be part of type checking. As a consequence, type checking will ensure that we are manipulating only well-formed contexts, that later declarations overshadow previous declarations, and that all declarations are of the specified form.

To illustrate, we show some well-formed and some ill-formed contexts.

Context	Is of schema `tctx`?

`b1:`**block**`(x:term,u:hastype x (arr nat nat)),`
`b2:`**block**`(y:term,u:hastype y nat)`
	yes

`x:term, u:hastype x (arr nat nat)`	no (not grouped in blocks)
`y:term`	no; typing assumption for `y` is missing
`b:`**block**`(x:term,u:hastype y nat)`	no (`y` is free)

`b1:`**block**`(x:term,u:hastype x (arr nat nat)),`
`b2:`**block**`(y:term,u:hastype x nat)`
	no (wrong binding structure)

Let us now show the type of a recursive function in Beluga which corresponds to the type uniqueness theorem.

We can read this type as follows: For every context γ of schema `tctx`, given a derivation for `hastype M T[]` in the context γ and a derivation for `hastype M S[]` in the context γ, we return a derivation showing that `eq T S` in the empty context. Although we quantified over the context γ at the outside, it need not be passed explicitly to a function of this type, but Beluga will be able to reconstruct it.

```
rec unique:(γ:tctx)[γ ⊢hastype M T[] ] → [γ ⊢hastype M S[] ] → [ ⊢equal T S]
    =
/ total d (unique _ _ _ _ d) /
fn d ⇒ fn f ⇒ case d of
| [γ ⊢t_app D1 D2] ⇒
  let[γ ⊢t_app F1 F2] = f in
  let[ ⊢ref]  = unique  [γ ⊢D1] [γ ⊢F1] in
    [ ⊢ref]

|[γ ⊢t_lam λx.λu. D] ⇒
  let[γ ⊢t_lam λx.λu. F] = f in
  let[ ⊢ref] = unique [γ,b:block(x:term,u:hastype x _)⊢D[… b.1 b.2] ]
                      [γ,b⊢F[… b.1 b.2] ] in
   [ ⊢ref]

| [γ ⊢#q.2] ⇒          % d : hastype #q.x T
  let[γ ⊢#r.2] = f  in  % f : hastype #q.x T'
    [ ⊢e_ref]
;
```

Listing 5.1: Type Uniqueness Proof

We call the type `[γ ⊢hastype M T[] ]` a contextual type and the object inhabiting it a contextual object. The term `M` can depend on the variables declared in the context γ. Implicitely, all meta-variables occurring inside `[ ]` are associated with a post-poned substitution which can be omitted, if the substitution is the identity substitution (which can be written as …). Hence, writing simply `M` in the context γ, is equivalent to writing `M[…]`. Why are meta-variables such as `M` associated with post-poned substitutions? - Intuitively, `M` itself is a contextual object of type `[γ ⊢term]` and … is the identity substitution which α-renames the bound variables. On the other hand, `T` and `S` stand for closed objects of type `tp` and they cannot refer to declarations from the context γ. Their declared type is `[⊢ tp]`. To use an object of type `[⊢tp]` in a context γ, we need to weaken it. This is what we express in the statement by writing `T[]` and `S[]`. Here `[]` denotes a weakening substitution from the empty context to γ. Note that these subtleties were not captured in our original informal statement of the type uniqueness theorem.

We subsequently omit writing weakening substitutions as they clutter the explanation and we should be in principle able to infer them. We consider each case individually. Each case in the proof on page 59 will correspond to one case in the case-expression.

Application case: If the first derivation `d` concludes with `t_app`, it matches the pattern `[γ ⊢t_app D1 D2]`, and is a contextual object of type `hastype (app M N) S` in the context γ. `D1` corresponds to the first premise of the typing rule for applications and has the contextual type `[γ ⊢hastype M (arr T S)]`.

Using a let-binding, we invert the second argument, the derivation `f` which must have type `[γ ⊢hastype (app M N)  S']`. `F1` corresponds to the first premise of the typing rule for applications and has type `[γ ⊢hastype M (arr T' S')]`. The appeal to the induction hypothesis using `D1` and `F1` in the on-paper proof corresponds to the recursive call `unique [γ ⊢D1] [γ ⊢F1]`. Note that while `unique`'s type says it takes a context variable `{γ:tctx}`, we do not pass it explicitly; Beluga infers it from the context in the first argument passed. The result of the recursive call is a contextual object of type `[ ⊢eq (arr T S) (arr T' S')]`. The only rule that could derive such an object is `ref`, and pattern matching establishes that `arr T S`=`arr T' S'` and hence `T` = `T'` and `S` = `S'`. Therefore, there is a proof of `[ ⊢eq S S']` using the rule `ref`.

Abstraction case: If the first derivation `d` concludes with `t_lam`, it matches the pattern `[γ ⊢t_lam λx.λu.D]`, and is a contextual object in the context γ of type `hastype (lam T (λx.M)) (arr T S)`. Pattern matching—through a let-binding—serves to invert the second derivation `f`, which must have been by `t_lam` with a subderivation `F1` deriving `hastype M S'` that can use `x`, `u:hastype x T`, and assumptions from γ[1].

The use of the induction hypothesis on `D` and `F` in a paper proof corresponds to the recursive call to `unique`. To appeal to the induction hypothesis, we need to extend the context by pairing up `x` and the typing assumption `hastype x T`. This is accomplished by creating the declaration `b:`**block** `x:term,u:hastype x T`. In the code, we wrote an underscore `_` instead of `T`, which tells Beluga to reconstruct it. (We cannot write `T` there without binding it by explicitly giving the type of `D`, so it is easier to write `_`.) To retrieve `x` we take the first projection `b.1`, and to retrieve `x`'s typing assumption we take the second projection `b.2`.

Now we can appeal to the induction hypothesis using `D1[…, b.1, b.2]` and `F1[… , b.1, b.2]` in the context `g,b:`**block** `x:term,u:hastype x T1`. Note that we apply explicitly the substitution `…, b.1, b.2` which allows us to transport a derivation `D1` in the context `γ, x:term, u:hastype x T1` to a derivation in the context `γ, b:` **block**`(x:term, u:hastype x T1)`. From the i.h. we get a contextual object, a closed derivation of `[⊢equal (arr T S) (arr T S')]`. The only rule that could derive this is `ref`, and pattern matching establishes that `S` must equal `S'`, since we must have `arr T S`= `arr T1 S'`. Therefore, there is a proof of `[ ⊢equal S S']`, and we can finish with the reflexivity rule `ref`.

[1] More precisely, `F1` has type `hastype M[…,x] S'[]`.

Assumption case: Here, we must have used an assumption from the context γ to construct the derivation d. Parameter variables allow a generic case that matches a declaration **block** x:term, u:hastype x T for any T in γ. Since our pattern match proceeds on typing derivations, we want the second component of the parameter #q, written as #q.2 or #q.u. The pattern match on d also establishes that M = #q.1 (or M = #q.x). Next, we pattern match on f, which has type hastype #q.1 S in the context γ. Clearly, the only possible way to derive f is by using an assumption from γ. We call this assumption #r, standing for a declaration **block** y:term,u:hastype y S, so #r.2 refers to the second component hastype #r.1 S. Pattern matching between #r.2 and f also establishes that #r.1 = #q.1. Finally, we observe that #r.1 = #q.1 only if #r is equal to #q. We can only instantiate the parameter variables #r and #q with bound variables from the context or other parameter variables. Consequently, the only solution to establish that #r.1 = #q.1 is the one where both the parameter variable #r and the parameter variable #q refer to the same bound variable in the context g. Therefore, we must have #r = #q, and both parameters must have equal types, and S = S' = T = T'. (In general, unification in the presence of Σ-types does not yield a unique unifier, but in Beluga only parameter variables and variables from the context can be of Σ type, yielding a unique solution.)

Chapter 6

Program Transformations

6.1 Translation of de Bruijn Terms to HOAS Terms

We return here to the beginning of Chapter 4 where we discussed two different representations for lambda-terms, namely using de Bruijn representation and using higher-order abstract syntax (HOAS). In Section 4.1, we defined de Bruijn terms and also showed how to translate lambda-terms in HOAS to their corresponding de Bruijn representation. Here, we implement de Bruijn terms and write a total function for the translation of lambda-terms in HOAS to their corresponding de Bruijn representation. To contrast we repeat our definition of lambda-terms using HOAS on the left and define de Bruijn terms on the right.

```
LF term   : type =                       LF dBruijn   : type =
| app   : term  → term  → term           | one    : dBruijn
| lam   : (term → term) → term;          | shift  : dBruijn  → dBruijn
                                         | lam'   : dBruijn  → dBruijn
                                         | app'   : dBruijn  → dBruijn  →
                                            dBruijn;
```

The translation from term to deBruijn is naturally recursive and as we travers a term, we will go under binders. Hence, our translation must translate a term in the context γ to deBruijn (see also Sec. 4.1). We hence first define the shape and structure of our context γ. This is simply done by : **schema** ctx = term ;

Here ctx is the name of a context schema and we declare contexts to be containing only declarations of type term. We can now turn our inference rules defining how to translate lambda-terms to de Bruijn terms into a recursive program:

```
rec vhoas2db : {γ:ctx}{#p:[γ ⊢term]}  [ ⊢dBruijn] =
 / total γ (vhoas2db γ  ) /
mlam γ ⇒ mlam #p ⇒  case [γ] of
```

```
| [] ⇒ impossible [ ⊢#p ]
| [γ', x:term] ⇒ (case [γ', x:term ⊢#p ] of
 | [γ',x:term ⊢x] ⇒ [ ⊢one ]
 | [γ',x:term ⊢#q[…] ] ⇒
   let [ ⊢Db] = vhoas2db [γ'] [γ' ⊢#q] in
     [ ⊢shift Db])
;

rec hoas2db : (γ:ctx) [γ ⊢term] → [ ⊢dBruijn ] = / total e ( hoas2db _ e) /
 fn e ⇒  case e of
  | [γ ⊢#p ] ⇒ vhoas2db [γ] [γ ⊢#p]

 | [γ ⊢lam λx.M] ⇒
   let [ ⊢F] =  hoas2db  [γ,x:term ⊢ M] in
     [ ⊢lam' F ]

 | [γ ⊢app M1 M2] ⇒
   let [ ⊢F1] = hoas2db  [γ ⊢ M1]  in
   let [ ⊢F2] = hoas2db  [γ ⊢ M2]  in
     [ ⊢app' F1 F2]
;
```

The type of the program reads as follows: Given a context γ of schema `ctx`, and given an object of type `term` in the context γ, we return an object `dBruijn` which is closed.

The type system will ensure we never work with variables outside their scope; it will also ensure that we produce a closed de Bruijn term.

Let us look at the easy cases first, for example the case `[γ ⊢app M1 M2]`. We note again that by default `M1` and `M2` can depend on the variables declared in the context γ.

So, to translate `[γ ⊢ app M1 M2]` we recursively translate `[γ ⊢ M1]` and `[γ ⊢ M2]`. Each recursive call will produce a closed de Bruijn term, namely `[⊢F1]` and `[⊢F2]`, which we can use to re-assemble our proper de Bruijn term `[⊢app' F1 F2]`.

When we translate a lambda-term, we must extend the context.

Finally, we must consider the variable cases. We implement the variable case using a separate function `vhoas2db` whose type can be read as: For all γ `:ctx` and for all parameters (variables) `#p:[γ ⊢ term]` there exists a `deBruijn` term. We note that the algorithm in Section 4.1 took advantage of the shape of the context; for example, we really took the context for being ordered. The same thing happens in Beluga. We can match on the shape of contexts.

We implement the function `vhoas2db` by pattern matching on the context γ. If the context is empty, then there are no variables and hence there is no `#p`. If the context is not empty but has the shape `γ', x:term`, we continue pattern matching on `#p`. There are two possible cases for `#p`:

1. `#p` stands for `x`, written as `[γ', x:term ⊢ x]`. In this case we simply return `one`.

2. `#p` stands for a variable from γ', written as `[γ', x:term ⊢ #q[…]]`. Note that we associate `#q` with the weakening substitution … that provides a map from a context γ' to the context `γ', x:term`. Therefore, `#q` can only be instantiated with a variable from `γ'`, but not `x`. In this latter case, we recursively call `vhoas2db` on `γ'` and `[γ' ⊢ #q]` and shift the result.

Part III

Advanced

Chapter 7

Normalization

In Chapter 4, we discussed the Simply Typed Lambda Calculus (STLC), its grammar, operational semantics and typing judgement. In addition, we studied some meta-theoretic properties of this formalism such as type preservation, the fact that type is preserved under evaluation of the operational semantics, and type uniqueness, the fact that each well-typed term has only one type. In this chapter we will explore another meta-theoretical property, normalization –i.e., the evaluation of well-typed terms always terminates. We proved earlier in Section 3.3 termination of evaluation for a simple language containing arithmetic and booleans. This proof was straightforward by structural induction on the typing derivation. Unfortunately, proving this property for the simply typed lambda-calculus is more challenging. A direct proof via structural induction fails. Instead, we fall back to a technique called proofs by logical relations going back to Tait Tait [1967] and was later refined by Girard [Girard et al. 1990]. The central idea of logical relations is to specify relations on well-typed terms via structural induction on the syntax of types instead of directly on the syntax of terms themselves. Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise.

Mechanizing logical relations proofs is challenging: first, specifying logical relations themselves typically requires a logic which allows arbitrary nesting of quantification and implications; second, to establish soundness of a logical relation, one must prove the Fundamental Property which says that any well-typed term under a closing simultaneous substitution is in the relation. This latter part requires some notion of simultaneous substitution together with the appropriate equational theory of composing substitutions. As Altenkirch Altenkirch [1993] remarked,

> “I discovered that the core part of the proof (here proving lemmas about CR) is fairly straightforward and only requires a good understanding of

> the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening."

While logical normalization proofs often are not large, they are conceptually intricate and mechanizing them has become a challenging benchmark for proof environments. In BELUGA, this proof and similar logical relations proofs can be implemented concisely. In developing this kind of proof in BELUGA, we introduce several new ideas:

- We will start by revisiting well-typed lambda-terms together with a simple call-by-name semantics and show an intrinsically.
- Using inductive definitions to define reducibility candidates and other properties of derivations and contexts
- Using first-class substitution variables

7.1 Representing Well-Typed Terms

Let's revisit the definition of STLC, starting with the grammar of terms, values and types:

$$\begin{array}{lll} \text{Terms} & M, N & ::= x \mid \lambda x.M \mid M\ N \mid c \\ \text{Types} & T, S & ::= B \mid T \to S \\ \text{Values} & V & ::= \lambda x.M \mid c \end{array}$$

Please note that in this presentation, we added to the language base type and a constant term.

Then we define the small-step operational semantics. In this particular presentation we use rules are different to Chapter 4. This presentation corresponds to what is typically referred to as a Call-By-Name operational semantics.

$\boxed{M \longrightarrow M'}$ Term M steps to term M'

$$\frac{}{(\lambda x.M)\ N \longrightarrow [N/x]M}\ \text{E-APP-ABS} \qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}\ \text{E-APP2}$$

$\boxed{M \longrightarrow^* M'}$ Term M steps in multiple steps to term M'

$$\frac{}{M \longrightarrow^* M}\ \text{M-REF} \qquad \frac{M \longrightarrow N \quad N \longrightarrow^* M'}{M \longrightarrow^* M'}\ \text{M-ONE-STEP}$$

Finally, we define the typing judgement. Note that we added the rule T-BASE for constants of base type.

$\boxed{M : T}$ Term M has type T

$$\dfrac{\begin{array}{c}\overline{x : T}^{\ u} \\ \vdots \\ M : S\end{array}}{\lambda x.M : T \to S}\ \text{T-ABS}^{x,u} \qquad \dfrac{M : T \to S \quad N : T}{M\ N : S}\ \text{T-APP} \qquad \dfrac{}{c : B}\ \text{T-BASE}$$

In Chapter 4, the formalization in Beluga followed very closely the paper presentation, in this case we will take a slightly different approach in which we combine the syntax of terms and the typing rules to obtain and *intrinsically typed representation* of the language.

```
LF tp : type =
| b :  tp
| arr : tp → tp → tp
;

LF tm : tp → type =
| app : tm (arr T S) → tm T → tm S
| lam : (tm T → tm S) → tm (arr T S)
| c : tm b
;
```

The type family `tm` defines simply-typed lambda terms by indexing `tm` by the type `tp`, the type of a term. In typical higher-order abstract syntax (HOAS) fashion, lambda abstraction takes a function representing the abstraction of a term over a variable. There is no case for variables, as they are treated implicitly in HOAS.

We now encode the step relation of the operational semantics. In particular, we create the `step` type indexed by two well-typed terms that represent each step of the computation. Hence our stepping relation is intrinsically type-preserving.

```
LF step : tm A → tm A → type =
| beta    : step (app (lam M) N) (M N)
| stepapp : step M M' → step (app M N) (app M' N)
;

LF mstep : tm A → tm A → type =
| refl   : mstep M M
| onestep: step M N → mstep N M' → mstep M M'
;
```

Notice again how the `beta` rule re-uses the LF notion of substitution by computing the application of `M` to `N`.

Finally, we define values using the type family `val` and a predicate `halts` to encode that a term halts if it steps into a value.

```
LF val : tm A → type =
| val_c : val c
| val_lam : val (lam M)
;
```

To characterize values, instead of directly implementing the grammar of values, we define a predicate on well-typed terms that selects the values.

```
LF halts : tm A → type =
| halts_m : mstep M M' → val M' → halts M
;
```

Finally, a term `M` halts (i.e. `halts M`), if it eventually step to a value. Thhe normalization proof will establish that for all terms `M`, `M halts`. At this point, one might be tempted to prove this property by induction on the structure of terms as we did in Section 3.3 for a language containing booleans and numbers. However, the naïf direct approach fails when we consider functions and function application. As we reduce well-typed terms, we cannot guarantee that their size is decreasing. We hence take a different approach.

7.2 Inductive and Stratified Definitions

Instead of proving directly that the evaluation of well-typed terms terminates on the size of terms, we characterize abstractly terms that are reducible at a given type (written as $\mathcal{R}_A(M)$). We then prove that evaluation of well-typed terms halts in two steps:

1. All well-typed terms are reducible.

2. If a term is reducible, it will halt.

We will be careful to define reducibility such that the second part of this argument becomes trivial; proving the first part is more challenging.

The key insight in defining what it means for a term to be reducible is to define it not on the size of the term which may in fact grow, but on the type of the term. This will give us a strong enough induction hypothesis for proving part 1.

We define the *Reducibility Candidates* predicate as:

- $\mathcal{R}_B(M)$ iff M halts

- $\mathcal{R}_{T\to S}(M)$ iff M halts and for all N, if $\mathcal{R}_{T}(N)$ then $\mathcal{R}_{S}(M\ N)$

The definitions states that a term of base type is in the relation if it halts, and a term of function type is reducible when it halts and it is well behaved under application. The general idea of the proof is to show that the terms in $\mathcal{R}_{T}(M)$ have the desired property. In our case the desired property is that the evaluation of the term halts, and it straight forward to show that if a term is reducible then its evaluation halts.

Reducibility cannot be directly encoded at the LF layer, since the universal quantifier and implication in the definition of $\mathcal{R}_{T\to S}(M)$ are not merely describing a hypothetical and parametric derivation. The definition is a statement in first-order logic! It states an inductive property about terms which we should be able to prove (possibly by induction). We often say it involves a strong implication or strong computational function space vs the weak implication (function space) LF provides. Hence we revisit to the computation layer of BELUGA. So far, the power of BELUGA's language corresponds to first-order logic. Here we now exploit the power of defining inductive properties about our domain. This corresponds to adding fix-points to our logic. From a programmers point of view, it corresponds to adding the power of indexed recursive data types which can be used to define properties and relations about contexts and contextual objects. Here we define a relation `Reduce` between types `A` and terms `M` that corresponds to our reducibility candidates $\mathcal{R}_{A}(M)$. We declare the kind of `Reduce` as using the keyword **ctype** which distinguishes this definition from LF types families. Computation-level type families such as `Reduce` are indexed by contexts and contextual objects which are embedded into computation-level types and programs are written inside `[ ]` (see Cave and Pientka [2012]).

```
stratified Reduce : {A:[⊢ tp]}{M:[⊢ tm A]} ctype =
| Rb : [⊢ halts M] → Reduce [⊢ b ] [⊢ M]
| Rarr :  [⊢ halts M] →
    ({N:[⊢ tm A]} Reduce [⊢ A ] [⊢ N] → Reduce [⊢ B ] [⊢ app M N])
    → Reduce [⊢ arr A B ] [⊢ M]
;
```

A term of base type `b` is reducible if it halts, and a `term M` of type `arr A B` is reducible if it halts, and moreover for every reducible `N` of type `A`, the application `app M N` is reducible. We write `{N:[⊢tm A]}` for explicit Π-quantification over `N`, a closed term of type `A`. To the left of the turnstile in `[⊢ tm A]` is where one writes the context the term is defined in – in this case, it is empty.

Adding inductive definitions to a language is non-trivial and may jeopardize is consistency. In BELUGA there are two forms: stratified definitions, as the definition of `Reduce`, and inductive definitions. What is this difference between LF types on the one hand and inductive and stratified types on the other? - To understand these differences it is worthwhile looking at some simpler examples.

We might be tempted to give an inductive definition of `Tm` as follows

```
inductive Tm : ctype =
| Lam : (Tm → Tm) → Tm;
```

On the surface, the function space `Tm -> Tm` is a strong function space; we cannot match and inspect the structure of this function as we are able with the LF function space that we used to define abstractions. We can only observe the behavior of computation. However, there is a deeper problem which is best illustrated with a slightly simpler example:

```
inductive D : ctype =
| Inj : (D → D) → D;

rec out : D → (D → D) =
fn x ⇒ case x of Inj f ⇒  f;

let omega  : D = Inj (fn x → (out x) x);
let omega' : D = (out omega) omega ;
```

The definition of `D` seems sensible at first; but we can cleverly create a non-terminating computation `omega'` that will keep on reproducing itself. The problem is the negative occurrence in the definition `D`.

To avoid such issues we commonly state that inductive definitions must satisfy the positivity restriction, i.e. the type we are defining cannot be used the the left hand side of an arrow.

What is then a stratified type? - Clearly our definition of `Reduce` is not inductive as we have an occurrence in a negative position in

```
({N:[⊢ tm A]} Reduce [⊢ A ] [⊢ N] → Reduce [⊢ B ] [⊢ app M N])
```

However, we observe that every computation we would be able to construct of this type has the property that the input to that computation is smaller when we concentrate on the first index describing the type, i.e. `[⊢ A]` is smaller than `[⊢ arr A B]`. While we still can never pattern match on this function to inspect its shape and we can never recurse on this definition directly, we can recurse on the index `A` instead.

7.3 Normalization Proof

7.3.1 Auxiliary Lemma

With the main definitions in place we now prove some more or less trivial lemmas that are sometimes omitted in paper presentations.First, we state and prove a simple lemma which stats that halts is closed under single step expansion.

Lemma 7.3.1. *If $\mathcal{S} :: \mathsf{M} \longrightarrow \mathsf{M}'$ and M' halts then M halts.*

The proof is straightforward. As M' halts, we have a value V and a derivation $\mathcal{S}' :: \mathsf{M}' \longrightarrow^* \mathsf{V}$. By using the rule M-ONE-STEP together with $\mathcal{S}$ and $\mathcal{S}'$ we have a witness for $\mathsf{M} \longrightarrow^* \mathsf{V}$ which shows that M halts. This can be encoded directly in BELUGA.

```
rec halts_step : {S:[⊢ step M M’]} [⊢ halts M’] → [⊢ halts M] =
mlam S ⇒ fn h ⇒
let [⊢ halts_m MS’ V] = h in
 [⊢ halts_m (onestep S MS’) V]
;
```

Next we prove closure of reducibility candidates under expansion.

Lemma 7.3.2 (Backward closed)**.** *If $\mathsf{M} \longrightarrow \mathsf{M}'$ and $\mathcal{R}_{\mathsf{T}}(\mathsf{M}')$ then $\mathcal{R}_{\mathsf{T}}(\mathsf{M})$.*

The proof is by induction on the type T. In the base case we appeal to `halts_step`, while in the `Rarr` case we must also appeal to the induction hypothesis at the range type, going inside the function position of applications.

```
rec bwd_closed’ : {T:[⊢ tp]}{M:[⊢ tm T]}{M’:[⊢ tm T]}{S:[ ⊢ step M M’]} Reduce
    [⊢ T] [ ⊢ M’] → Reduce [⊢ T] [ ⊢ M] =
/ total a (bwd_closed’ a) /
 mlam T, M, M’ ⇒ mlam SS ⇒ fn r ⇒ case [⊢ T] of
| [⊢ b] ⇒ let I ha = r in I (halts_step [ ⊢ SS] ha)
| [⊢ arr T S] ⇒
  let Arr ha f = r in
    Arr (halts_step [ ⊢ MS] ha)
        (mlam N ⇒ fn rn ⇒
         bwd_closed’ [⊢ S] [⊢ _ ] [⊢ _ ] [ ⊢ stepapp MS] (f [ ⊢ N] rn))
;

rec bwd_closed : {S:[ ⊢ step M M’]} Reduce [⊢ T] [ ⊢ M’] → Reduce [⊢ T] [ ⊢ M]
     =
/ total (bwd_closed) /
mlam S ⇒ fn r ⇒
let [⊢ S] : [⊢ step M M’] = [⊢ S] in
let [⊢ M] : [⊢ tm T]    = [⊢ M] in
  bwd_closed’ [⊢ T] [⊢ M ] [⊢ M’ ]  [⊢ S] r;
```

The trivial fact that reducible terms halt has a corresponding trivial proof, analyzing the construction of the the proof of `Reduce[⊢ T] [⊢ M]`.

Finally we prove that that if a term M is reducible at type T then it halts.

Lemma 7.3.3. *If $\mathcal{R}_{\mathsf{T}}(\mathsf{M})$ then M halts.*

The proof is trivial and follows directly from the definition of $\mathcal{R}_T(M)$. This can be translated direct into BELUGA as follows.

```
rec reduce_halts : Reduce [⊢ T] [⊢ M] → [⊢ halts M] =
fn r ⇒ case r of
| Rb h ⇒ h
| Rarr h f ⇒ h
;
```

It is at this point, that we may start thinking on the proof of the main theorem. The main theorem states that all terms are reducible, so a first approximation to this could be to try to prove this theorem:

```
rec main : {M:[⊢ tm A]} Reduce [⊢ A] [⊢ M] =
  ?
;
```

However, if we try to prove this theorem, we very quickly realize that we need to have appeals to the induction hypothesis for open term. In particular, when trying to build the reducibility candidate for λ-terms. We hence need to generalize this statement.

7.3.2 Fundamental Lemma and Main Theorem

It is worthwhile to revisit the proof of the fundamental lemma on paper. The right generalization is often stated as follows:

Lemma 7.3.4 (Main lemma). *If* $\mathcal{D} : \Gamma \vdash M : T$ *and* $\mathcal{R}_\Gamma(\sigma)$ *then* $\mathcal{R}_T([\sigma]M)$ *where* $\mathcal{R}_\Gamma(\sigma)$ *is defined as:*

- $\mathcal{R}_\cdot(\cdot)$ *(i.e. empty substitutions are reducible at the empty context)*
- $\mathcal{R}_{\Gamma,x:T}(\sigma, N/x)$ *iff* $\mathcal{R}_\Gamma(\sigma)$ *and* $\mathcal{R}_T(N)$

Proof. By structural induction on the typing derivation $\mathcal{D}$.

Case $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Gamma, x{:}T \vdash M : S\end{matrix}}{\Gamma \vdash \mathsf{lam}\ x.M : T \to S}\ \mathit{lam}$

$[\sigma](\mathsf{lam}\ x.M) = \mathsf{lam}\ x.([\sigma, x/x]M)$ by properties of substitution
$\mathsf{lam}\ x.[\sigma, x/x]M$ halts since it is a value
Suppose $\mathcal{R}_T(N)$.

1. $\mathcal{R}_S([\sigma, N/x]M)$ by I.H. on $\mathcal{D}_1$ since $\mathcal{R}_\Gamma(\sigma)$

2. $\mathcal{R}_S([N/x][\sigma, x/x]M)$ by properties of substitution

3. $\mathcal{R}_S(\mathsf{app}\ (\mathsf{lam}\ x.\ [\sigma, x/x]M)\ N)$ by Backwards closure

Hence $\mathcal{R}_{T\to S}([\sigma]\mathsf{lam}\ x.M)$ by definition

□

In the proof we spelled out explicitly where we rely on substitution properites such as composition of substitution and associativity. This highlights the important and often hidden role substitutions play. In BELUGA, where we have first-class simultaneous substitutions together with their equational theory, the uses of such substitution properties will be silently taken care of. This in turn leads to a very elegant and compact encoding of the fundamental lemma as it only reflects the main structure of the proof.

As the statement talks about open terms, i.e. terms in a context Γ, we first define a context schema that classifies contexts containg typing assumptions about variables.

```
schema ctx = tm T;
```

In the statement of the fundamental lemma, we also relied on the concept of *grounding substitutions*. This is again natively supported in BELUGA using first-class substitutions that provide a mapping from Γ to an empty context. Finally, we need to define what it means for a substitution to be *reducible*. In particular, a *a reducible substitution* is a substitution where all the terms that compose it are also reducible. We define that, using an inductive data-type indexed by grounding substitutions. In the code below we pretty print the composition of the substitution σ with the empty substitution (written as ^) by simply writing σ.

```
inductive Red_sub : {γ:ctx}{σ:[⊢ γ]} ctype =
| Nil : Red_sub  [ ] [⊢ ^ ]
| Dot : Red_sub  [γ] [⊢ σ ] → Reduce [⊢ T] [⊢ M]
     → Red_sub [γ, x:tm T[]] [⊢ σ,M ]
;

rec lookup : {γ:ctx}{#p:[γ ⊢ tm T[]]}Red_sub [γ] [⊢ σ] →
             Reduce [⊢ T] [⊢ #p[σ]] =
mlam γ⇒ mlam #p ⇒ fn rs ⇒ case [γ] of
| [] ⇒ impossible [ ⊢ #p]
| [γ’, x:tm T] ⇒ (case [γ’, x:tm T ⊢ #p] of
 | [γ’,x:tm T ⊢  x] ⇒     let (Dot rs’ rN) = rs in rN
 | [γ’,x:tm T ⊢  #q[…]] ⇒ let Dot rs’ rN = rs in
                         lookup [γ’] [γ’ ⊢ #q] rs’)
;
```

Recall all meta-variables are associated with substitutions, but we are able to omit writing that substitution, if it is the identity substitution. The same holds for substitution variables. They are associated with a delayed substitution which intuitively represent a delayed composition of a substitution variable with that substitution. If the substitution variable is associated with the identity substitution or a trivial empty substitution (i.e. from the empty context to another empty context), we omit writing it. In other words we write simply σ for σ[..] (i.e. the composition of σ with the identity substitution) and σ[] (i.e. the composition of σ with the trivial empty substitution)[1].

We also show how to look-up a reducible term in a reducible substitution by using the lookup function. We can read the type of lookup as follows: For a context γ, for all variables #p of closed type T from the context γ =x_1:term T_1, ..., x_n:term T_n , if $\mathcal{R}_\sigma(\gamma)$, i.e. σ is a substitution of the form $N_1/x_1, \ldots, N_n/x_n$ that provides a term N_i for each variable x_i in γ s.t. Reducible T_i N_i , we know Reducible T #p[σ].This property may seem obvious, but it must be shown by recursively checking it for each variable in γ. We therefore pattern macht on the context γ. If the context γ is empty, then there is no variable and hence the case is impossible (written as **impossible** [⊢ #p].

If $\gamma = \gamma', x : \mathtt{tm} T$, then we pattern match on the parameter variable #p. There are two cases: either #p = x. in this case we simply analyze rs which stands for the proof Red_sub [γ', x:term T[]] [σ', N'] and return the witness rN for Reducible [⊢ N'] [⊢ T].

Otherwise #p stands for a variable other than x and must come from γ'. This case is written as [γ', x:term T ⊢ #q[...]] where the parameter variable #q stands for a variable from γ' and we simply recurse on [γ' ⊢ #q].

We make sure to associate the parameter variable with the weakening substitution [...] in the pattern. Note that [...] is not an identity substitution here. The identity substitution in the context γ', x:term T[] is [...,x].

Finally, we can revisit the fundamental lemma and show its implementation in BELUGA.

```
rec main: {γ:ctx}{M:[γ ⊢ tm T[]]} Red_sub [γ] [⊢ σ] → Reduce [⊢ T] [⊢ M[σ]] =
 mlam γ ⇒ mlam M ⇒ fn rs ⇒ case [γ ⊢ M] of
| [γ ⊢ #p] ⇒ lookup [γ] [γ ⊢ #p] rs
| [γ ⊢ lam λx. M] ⇒
 Rarr [⊢ halts_m refl val_lam]
   (mlam N ⇒ fn rN ⇒
    bwd_closed [⊢ beta] (main [γ,x:tm _] [γ,x ⊢ M] (Dot rs rN)))
 | [γ ⊢ app M1 M2] ⇒
  let Rarr ha f = main [γ] [γ ⊢ M1] rs in
  f [⊢ _ ] (main [γ] [γ ⊢ M2] rs)
```

[1] In ASCII we write #S for σ.

```
| [γ' ⊢  c] ⇒ Rb [⊢ halts_m refl val_c]
;
```

In the variable case, we simply use the lookup-function to guarantee that #p[σ] is reducible. The case for abstractions follows our on paper proof. To prove the Reducible [⊢ arr T S] [⊢ lam λx.M], we must prove two things: first, we must show halts (lam λx.M). We use val_lam to justify that abstractions are values and by reflexivity abstrations step to themselves. Therefore their evaluation halts. Note that we do not explicitly refer to properties of substitutions in our implementation, although we highlighted such reasoning in our previous proof. Such reasoning is hidden as part of the equational theory our type checker uses. Second, we show that for all N, if Reduce [⊢T] [⊢N] then Reduce [⊢S] [⊢app (lam λx.M) N].

We first assume N together with rN:Reduce [⊢ T] [⊢ N] by writing **mlam** N ⇒ **fn** rN ⇒ .... . Then we use the IH and the lemma bwd_closed to finish the proof as described earlier. We emphasize again that not explicit substitution reasoning is required in the implementation, although it clearly is used in the proof.

Lastly, consider the case for applications, i.e. [γ ⊢ app M1 M2]. By the IH we have that Reducible [⊢ arr S T] [⊢ M1[σ]]. By definition of Reduce, we have that halts M1[σ] and more importantly, we have a function f whose type says: for all N, if Reduce [⊢ S] [⊢ N] then Reduce [⊢ T] [⊢ app M1[σ] N]. We now simply use f and pass to it [⊢ M2[σ] and a witness for Reduce [⊢ T] [⊢ M2[σ]] which is obtained by making a recursive call on [γ ⊢ M2].

And finally we have all the elements to prove that all well typed terms eventually reduce to a value:

```
rec weakNorm : {M:[⊢ tm A]} [⊢ halts M] =
mlam M ⇒ reduce_halts (main [] [⊢ M] Nil)
;
```

Let's retrace our steps. We set out to prove that all terms reduce to a value. A simple proof by induction would not give us a powerful enough induction hypothesis, so we defined the reducibility candidate relation that was inductive on the type and allowed for a powerful enough induction hypothesis. Then when proving that all terms are reducibl, We needed again a more powerful induction hypothesis to be able to prove that all terms were in the reducibility relation. In particular, we generalized the theorem to all terms and their grounding substitutions, for this we needed a predicate on grounding substitution, to be sure that all the terms in them were reducible. With all this machinery in place the proof went through by appealing to the lemmas we had proven earlier.

Index

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, 1990.

T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. ISBN 3-540-56517-5.

A. W. Appel. Verified software toolchain. In *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

B. Aydemir, S. Zdancewic, and S. Weirich. Abstracting syntax. (CIS) 901, University of Pennsylvania, 1990.

S. Berardi. Girard normalization proof in LEGO. In *Proceedings of the First Workshop on Logical Frameworks*, pages 67–78, 1990.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

C. Coquand. A proof of normalization for simply typed lambda calculus writting in ALF. In *Informal Proceedings of Workshop on Types for Proofs and Programs*, pages 80–87. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.

C. Doczkal and J. Schwinghammer. Formalizing a strong normalization proof for Moggi's computational metalanguage: A case study in Isabelle/HOL-nominal. In

Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09), pages 57–63. ACM, 2009. ISBN 978-1-60558-529-1.

J.-Y. Girard, Y. Lafont, and P. Tayor. *Proofs and types*. Cambridge University Press, 1990.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM. doi: 10.1145/800233.807045. URL `http://doi.acm.org/10.1145/800233.807045`.

J. Narboux and C. Urban. Formalising in Nominal Isabelle Crary's completeness proof for equivalence checking. *Electr. Notes Theor. Comput. Sci.*, 196:3–18, 2008.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. Technical Report 33D.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*, pages 199–208, June 1988.

B. Pientka and A. Cave. Inductive Beluga:Programming Proofs (System description). In *25th International Conference on Automated Deduction (CADE-25)*. Springer, 2015.

B. Pientka and J. Dunfield. Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description). In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer-Verlag, 2010.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32 (2):198–212, 1967.

C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40 (4):327–356, 2008.

C. Urban, J. Cheney, and S. Berghofer. Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.*, 12(2):15, 2011.

J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Symposium on Principles of Programming Languages*, pages 427–440. ACM Press, 2012.