Reflection and Its Application to Mechanized Metareasoning about Programming Languages

Thesis by

Xin Yu

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

2007

(Defended June 12, 2006)

© 2007
Xin Yu
All Rights Reserved

Acknowledgments

Although I am listed as the author of this thesis, many other people should share the credit for its production. First and foremost I thank my advisor, Professor Jason Hickey, who has provided me with a great deal of support, guidance and instruction throughout my time at Caltech. This is joint work with him and my fellow researchers: Aleksey Nogin and Alexei Kopylov, from discussions with whom I benefited a lot.

I would also like to thank the members of our group, Cristian Tapus, Nathan Gray, Jerome White, and David Goblet, for their help and friendship.

I greatly appreciate the efforts of my thesis committee (Professors Jason Hickey, Alain Martin, Steven Low, and Dr. Rajeev Joshi).

Without the always ready help from various Caltech personnel, my life during the past six years would be painful. I specially thank Jeannie and Divina from the Health Center, and Natalie, Edith, and Icy from the Graduate Office.

I also owe much gratitude to my friends, most notably Zhe Wu, Cong Zhang, and Ling Zhou, for the happiness they brought and encouragement they never hesitated to share.

Above all, I thank my family for their unfailing love, care and support. For them, "thanks" is never enough.

Abstract

It is well known that adding reflective reasoning can tremendously increase the power of a proof assistant. In order for this theoretical increase of power to become accessible to users in practice, the proof assistant needs to provide a great deal of infrastructure to support reflective reasoning. In this thesis we explore the problem of creating a practical implementation of such a support layer.

Our implementation takes a specification of a logical theory (which is identical to how it would be specified if we simply intended to reason within this logical theory, instead of reflecting it) and automatically generates the necessary definitions, lemmas, and proofs that are needed to enable the reflected metareasoning in the provided theory.

One of the key features of our approach is that the *structure* of a logic is preserved when it is reflected, including variables, meta variables, and binding structure. This allows the structure of proofs to be preserved as well, and there is a one-to-one map from proof steps in the original logic to proof steps in the reflected logic. The act of reflecting a language is automated; all definitions, theorems, and proofs are preserved by the transformation and all the key lemmas (such as proof and structural induction) are automatically derived.

The principal representation used by the reflected logic is higher-order abstract syntax (HOAS). However, reasoning about terms in HOAS can be awkward in some cases, especially for variables. For this reason, we define a computationally equivalent variable-free de Bruijn representation that is interchangeable with the HOAS in all contexts. The de Bruijn representation inherits the properties of substitution and alpha-equality from the logical framework, and it is not complicated by administrative

issues like variable renumbering.

We further develop the concepts and principles of proofs, provability, and structural and proof induction. This work is fully implemented in the MetaPRL theorem prover. We illustrate with an application to $F_{<:}$ as defined in the POPLmark challenge.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

1.1 What Is Reflection 4

1.2 Programming Language Metatheory 5

1.3 Terminology 7

1.4 Organization 10

2 Previous Models of Reflection **11**

3 A Hybrid HOAS/de Bruijn Representation of the Syntax **18**

3.1 Bound Terms 19

3.2 Terminology 20

3.3 Abstract Operators 21

3.4 Inductively Defining the Type of Well-Formed Bterms 22

3.5 Our Approach 24

4 Formal Implementation of the Syntax in a Theorem Prover **25**

4.1 Computations and Types 25

4.2 HOAS Representation 26

4.3 Vector HOAS Operations 29

4.4 de Bruijn Representation 30

4.5 Operators 37

4.6 The Type of Reflected Terms 37

4.7 Exotic Terms and Decidability 39

4.8 A Small Example 41

4.9 Related Work 42

5 The HOAS Representation Function **44**

6 Sequent Representation **47**

6.1 Sequent Context Induction 47

6.2 Computation on Sequent Terms 50

6.3 Computing Canonical Sequent Representations 51

7 Proof Reflection **54**

7.1 Proof Checking 55

7.2 Derivations and Provability 56

7.3 Proof Reflection and Automation 58

7.4 Proof Induction 60

7.5 Preliminary Discussion 62

8 An Example: $F_{<:}$ **65**

8.1 Defining the Syntax of $F_{<:}$ 65

8.2 Reflected Syntax 67

8.3 The $F_{<:}$ Logic 68

8.4 Structural Induction 70

9 Discussion and Future Work **73**

Bibliography **76**

A Architecture and Summary of the Implementation in MetaPRL **84**

B Partial Listing of Relevant MetaPRL Theories **86**

B.1 Itt_hoas_base module 87

B.1.1 Parents 87

B.1.2 Terms 87

B.1.3 Rewrites 89

B.2 Itt_hoas_vector module 91

B.2.1 Parents 91

B.2.2 Terms 91

B.2.3 Rewrites 93

B.3 Itt_hoas_debruijn module 96

B.3.1 Parents 96

B.3.2 Terms 97

B.3.2.1 A de Bruijn-like representation of syntax 97

B.3.2.2 Basic operations on syntax 97

B.3.3 Rewrites 99

B.4 Itt_hoas_operator module 106

B.4.1 Parents 106

B.4.2 Terms 106

B.4.3 Rules 107

B.4.4 Concrete Operators 109

B.5 Itt_hoas_destterm module 110

B.5.1 Parents 111

B.5.2 Terms 111

B.5.3 Rules 112

B.5.4 Rewrites 112

B.6 Itt_hoas_bterm module 114

B.6.1 Parents 114

B.6.2 Terms 114

B.6.3 Rewrites 116

B.6.4 Rules 117

Chapter 1

Introduction

Very generally, reflection is the ability of a system to be "self-aware" in some way. More specifically, by reflection we mean the property of a computational or formal system to be able to access and internalize some of its own properties.

It is well known that reflection can tremendously increase the power of a formal reasoning system. Surprisingly, formal systems tend to avoid the subject, or provide poor tools for reflective work. The fundamental goal of this work is to provide a method for implementing a practical reflection of syntax, computation, and proof in a logical framework.

The implementation of a reflection system has two core parts: representing the syntax, and mechanizing the reasoning. The issue of representation is central, and far from trivial: dealing with formal syntax means dealing with structures that involve bindings, and in a logical context it seems natural to use the same formal tools to describe syntax—often limiting the usability of such formalizations to specific theories and toy examples. Representing languages with bindings has been recognized as crucial by the theorem proving community; it requires a treatment of α-equivalence and capture-avoiding substitution. Many different solutions to the binding problem have been proposed. For example, one may use nameless or de Bruijn encodings for variables [23], nominal representations [53, 17], or higher-order abstract syntax [52] (HOAS), with various trade-offs. The de Bruijn encoding provides a very concrete representation with a clear induction principle, but reasoning is cluttered by superfluous artifacts like the need to perform name shifting, and one gets very little built-in

help from the prover for these issues. At the other extreme, HOAS provides a clean abstract representation with excellent support from the prover, but variable names are inaccessible and the induction scheme can be hard to formulate. Nominal approaches are in between; names are accessible, the representation is mildly cluttered by explicit renamings, but frequently the existing framework logic or metalogic must be extended to include explicit naming contexts.

In this work we present a hybrid approach using a combined HOAS/de Bruijn representation for terms. To address the issue of computation and induction over terms, each reflected term has two equivalent forms. One is a HOAS representation, where variables in the object logic are represented by variables in the reflected logic, and binding is preserved. On top of it there is a de Bruijn representation layer, where binders are specified by arity, and variables are denoted with numerical indices. These two representations are formally and computationally indistinguishable, which allows the appropriate representation to be selected at the appropriate time. For example, the HOAS representation is normally the preferred form for users because of its clarity, but the de Bruijn representation is more appropriate for computations that involve induction or computation on variables (for example, computing the free variables of a term).

The fundamental reason that our approach is practical is that it preserves structure *exactly* in this sense: all variables, including both object and meta variables, are preserved by the representation. In the manner of HOAS, binding structure is also preserved by the transformation, both for formulas and for (sequent) judgments. One might call this meta-higher-order abstract syntax.

The benefit of preserving the term structure is that mechanized reasoning works transparently. That is, there is a one-to-one correspondence from proof steps in the original logic to reflected proof steps in the metalogic. In fact the translation is direct and mechanical, which means that proof automation in the original logic also applies in the reflected logic.

The reflection framework we implement provides a context that is convenient for talking about representations and specifying properties of proofs in a uniform

way that applies to all logical theories that can be specified as a logic in the logical framework. Furthermore, the act of reflecting a logic is automated, and the principles for structural and proof induction are automatically derived for every reflected theory.

Naturally, our work can be applied to develop mechanized methods for a programming language metatheory, with which we can reason not only about individual languages, but also about *classes* of languages, language *schemas*, and so on. The reason is that a programming language can be defined as a particular logic in the logical framework, using the same mechanisms for definition, reasoning, and automation that are available to other logics. There are at least three logics in consideration here—$\mathbb{P}$: the programming language (also called the "object logic"); $\mathbb{M}$: the metalogic in which reasoning about the programming language is to be performed; and $\mathbb{F}$: the meta-metalogic, or framework logic, in which the metalogic $\mathbb{M}$ is defined. The choice of framework logic $\mathbb{F}$ and metalogic $\mathbb{M}$ are based mainly on the choice of the prover, which then restricts the set of choices for the framework logic and the metalogic (in many cases $\mathbb{F}$ and $\mathbb{M}$ are one and the same). The main issue here is to define a *representation* of programs and judgments in $\mathbb{P}$ in terms of formulas, propositions, and sentences in $\mathbb{M}$. We use *reflection* to provide a uniform representation map, where a programming language $\mathbb{P}$ is reflected en masse to form a subtheory of the metalogic $\mathbb{M}$. As mentioned earlier, the act of reflecting a language is automated; all definitions, theorems, and proofs are preserved by the transformation and all the key lemmas (such as proof and structural induction) are automatically derived. We will illustrate with an application to the $F_{<:}$ language as defined in the POPLmark challenge [12].

To summarize, the contributions of this thesis include:

- a new approach to representing languages with bindings;
- a theory of implementing a practical reflection of syntax, computation, and proof in a logical framework;
- an implementation that can be used for mechanized metareasoning about programming languages.

We have carried out a complete formal account of this work in the MetaPRL logical framework [41, 43]. In the appendices of this thesis, we provide an architectural overview of the formal implementation, and also list several most important modules of this work. All formalizations are available online [43, 42]. An important feature of this work is that no extensions to the metalogic or framework logic are needed.

In the rest of this chapter, we provide some background knowledge. We first give a more detailed account of what reflection is in Section 1.1, then in Section 1.2 give an introduction on the current state of the art in automated programming language metatheory. In Section 1.3 we develop the syntax and language of logics. At the end of the chapter, we outline the structure of this thesis.

1.1 What Is Reflection

As mentioned earlier, reflection is the ability of some entity to refer to itself.

There are many areas of computer science where reflection plays or should play a major role. When exploring properties of programming languages (and other languages) one often realizes that languages have at least two kinds of properties—*semantic* properties that have to do with the *meaning* of what the language's constructs express, and *syntactic* properties of the language itself.

Suppose for example that we are exploring some language that contains arithmetic operations. And in particular, in this language one can write polynomials like $x^2 + 2x + 1$. In this case the number of roots of a polynomial is a semantic property since it has to do with the *valuation* of the polynomial. On the other hand, the degree of a polynomial could be considered an example of a syntactic property since the most natural way to define it is as a property of the *expression* that *represents* that polynomial. Of course, syntactic properties often have semantic consequences, which is what makes them especially important. In this example, the number of roots of a polynomial is bounded by its degree.

Another area where reflection plays an important role is run-time code generation—in most cases, a language that supports run-time code generation is essentially re-

flective, as it is capable of manipulating its own syntax. In order to reason about run-time code generation and to express its semantics and properties, it is natural to use a reasoning system that is reflective as well.

There are many different flavors of reflection. The *syntactic reflection* we have seen in the examples above, which is the ability of a system to internalize its own syntax, is just one of these many flavors. Another very important kind of reflection is *logical reflection*, which is the ability of a reasoning system or logic to internalize and reason about its own logical properties. A good example of a logical reflection is reasoning about knowledge—since the result of reasoning about knowledge is knowledge itself, the logic of knowledge is naturally reflective [11].

In most cases it is natural for reflection to be iterated. In the case of syntactic reflection we might care not only about the syntax of our language, but also about the syntax used for expressing the syntax, the syntax for expressing the syntax for expressing the syntax and so forth. In the case of the logic of knowledge it is natural to have iterations of the form "I know that he knows that I know ..."

Reflection can add a lot of additional power to a formal reasoning system [34,10]. In particular, it is well known [36,44,29,48] that reflection allows a superexponential reduction in the size of certain proofs. In addition, reflection could be a very useful mechanism for implementing proof search algorithms [3,33,21]. See also Harrison [40] for a survey of reflection in theorem proving.

1.2 Programming Language Metatheory

It has been well established that better programming technology is needed to assure the safety and reliability of our software infrastructure. As found in the PITAC reports [55,56], our software infrastructure is fragile, technologies to build reliable software are inadequate, and the demand for reliable software exceeds our ability to produce it.

The development of new programming languages is a crucial part of addressing the software development problem. New languages, including domain-specific languages,

can simplify development and maintenance by making programs more concise and clear, and they can improve reliability in the form of *guarantees* that hold for all programs written in the language. For example, strongly-typed languages guarantee memory and code safety, though at some loss in expressivity. Domain-specific languages can often go further, guaranteeing additional properties relevant to a domain.

Programming language properties and guarantees are more than just properties of specific programs—they are properties of *any* program written in the language. They ensure that, no matter how a program is constructed or modified, the guarantees will continue to hold. They can be a tremendous benefit during software development because no design effort is needed, the guarantees simply hold.

The study of programming language properties is called programming language *metatheory*. Given the benefits of metatheoretical guarantees, one might wonder why there are not more such languages in widespread use. There are at least two answers. First, while the properties of any individual language may be precise, the properties of the composition of languages is not nearly as clear, and there has been little research on this topic. For this reason (and others) programmers prefer to keep their repertoire, and their risk, limited. Second, programming language metatheory can be exceedingly tedious and error prone, using highly technical, yet routine, proof methods. This leads to a reluctance on the part of the designer to perform the proofs, and a reluctance on the part of the community to read the proofs. The consequence is that even in the rare case when a new language is accompanied by proofs, these proofs are given less scrutiny than we might wish. This leaves us with two central problems: (1) the metatheoretical properties of systems constructed using multiple languages are poorly studied, and (2) new languages are rarely accompanied by a strong metatheory.

Automated methods promise a solution to part of the problem. Since proofs tend to be routine, why not have a machine perform the tedious, technical work? There has been some progress in this direction recently, notably presented as the POPLmark challenge [12], which poses the specific problem of automated reasoning for the kernel System $F_{<:}$ language (the polymorphic λ calculus with subtyping).

Several researchers are performing ongoing work related to the challenge [20, 59, 30], and it seems likely that acceptable solutions will be found, at least for $F_{<:}$. In these approaches, the $F_{<:}$ language is formal, and the proofs are mechanized. However, the proof methods are ad hoc.

Stated more generally, the current state of the art in automated metatheory addresses the problem of formal automated reasoning about specific *concrete* languages using custom methods for each language. What is missing is the ability to reason about properties that apply to multiple languages, or relations between languages, or formal functions that map the properties of one language to another. That is, what is missing is (1) reuse of formal knowledge, and (2) reasoning about systems with components written in several languages.

Our reflection framework can provide mechanized methods for a programming language metatheory with the following abilities.

- The ability to reason about *classes* of languages, so that results may be reused to cover multiple concrete languages
- The ability to perform higher-order reasoning, establishing relationships between languages.

1.3 Terminology

We assume that the language of the logical framework $\mathbb{F}$ contains sequents, second-order meta variables, and terms, as shown in Figure 1.1. A term t is a formula containing variables, concrete terms, or sequents. A concrete term $op\{b_1; \cdots; b_n\}$ has a name op, as well as some subterms $b_1, \ldots, b_n$ that have possible binding occurrences of variables. For example, a term for representing the sum $i + j$ might be defined as $\texttt{add}\{.i; .j\}$ (normally, we will omit the leading "." if there are no binders, writing it as $\texttt{add}\{i; j\}$). A term for lambda-abstraction $\lambda x. t$ would include a binding occurrence

[1] Strictly speaking, context variables are *bindings* and meta-variables have context arguments in addition to term arguments. This does not affect the presentation until we get to context induction (Chapter 6), and we omit context arguments for now.

t	$::=$	x	object (first-order) variables
	$\mid$	$z[t_1; \cdots ; t_n]$	second-order meta-variables
	$\mid$	$\Gamma \vdash t$	sequents
	$\mid$	$op\{b_1; \cdots ; b_n\}$	concrete terms
b	$::=$	$x_1, \ldots, x_n. t$	bound terms
Γ	$::=$	$h_1; \cdots ; h_n$	sequent contexts
h	$::=$	$X[t_1; \cdots ; t_n]$	context meta-variables[1]
	$\mid$	$x \colon t$	hypothesis bindings and terms
$\mathbb{P}$	$::=$	$R_1; R_2; \cdots ; R_n$	a logic
R	$::=$	$t_1 \longrightarrow \cdots \longrightarrow t_n$	inference rules
		(t_i are closed w.r.t. object variables)	

Figure 1.1: Syntax of formulas and logics

`lambda`$\{x. t\}$. Note that in this case, the primitive binding construct is the bound term b, and λ-binders are a defined term. An alternate choice would be to use a single primitive λ binder (for example, as is done in LF [39]).

A sequent $\Gamma \vdash t$ includes a sequent context Γ, which is a sequence of dependent hypotheses $h_1; \cdots ; h_m$, where each hypothesis is a binding $x \colon t$ or a context variable $X[t_1; \cdots ; t_n]$ (x and X bind to the right). Note that sequents can be arbitrarily nested inside other terms and are not necessarily associated with judgments.

Second-order meta variables $z[t_1; \cdots ; t_n]$ and context variables $X[t_1; \cdots ; t_n]$ include zero or more term arguments $t_1, \ldots, t_n$. These meta variables represent closed *substitution functions*, and are implicitly universally quantified for each rule in which they appear [45]. For example, a second-order variable $z[]$ represents all closed terms (normally we will write simply z, omitting empty brackets[2]). The second-order variable $z[x]$ represents all terms with zero or more occurrences of the variable x (that is, any term where x is the only free variable).

To illustrate, consider the "substitution lemma" that is valid in many logics. In textbook notation, it might be written as follows, where $t_1[x \leftarrow s]$ represents the

[2]This will not cause confusion with first-order variables, since, when in use, a first-order variable cannot occur free in a judgment and a second-order variable cannot be bound; we can always distinguish them.

substitution of s for x in t_1, and $x \notin \mathsf{fv}(\Delta)$ is a side condition of the rule.

$$\frac{\Gamma, x\colon t_3, \Delta \vdash t_1 \in t_2 \quad \Gamma, \Delta \vdash s \in t_3 \quad (x \notin \mathsf{fv}(\Delta))}{\Gamma, \Delta \vdash t_1[x \leftarrow s] \in t_2}$$

In our more concrete notation, s, t_1, t_2, t_3 are all represented with second-order variables, and Γ, Δ with context variables. Substitutions are defined using the term arguments; rules are defined using the meta implication $\cdot \longrightarrow \cdot$, and we consider all meta variables to be universally quantified in a rule. The concrete version is written as follows (where we use $s \in t$ as a pretty form for a term $\texttt{member}\{s; t\}$, and z_i are second-order meta variables).

$$\begin{aligned} &(X; x\colon z_3; Y \vdash z_1[x] \in z_2) \longrightarrow \\ &(X; Y \vdash z_0 \in z_3) \longrightarrow \\ &(X; Y \vdash z_1[z_0] \in z_2) \end{aligned} \tag{1.1}$$

In the final sequent, the term $z_1[z_0]$ implicitly specifies substitution of z_0 for x in z_1.

Note how the term arguments are used to specify binding precisely—the variable x is allowed to occur free in z_1, but in no other term. The reason we adopt this second-order notation is for the precision that can be obtained without the need for side conditions. All rule schemas representable with substitution notation are also representable as second-order schemas, but not vice versa, except with a sufficiently rich set of side conditions.

For the final part, a logic $\mathbb{P}$ is an ordered sequence of rules. Each rule may be an axiom, or it may be derived from the previous rules in the logic.

In the particular framework logic $\mathbb{F}_{\mathsf{MetaPRL}}$ that we use, logics are considered to be open ended. That is, the models of the logic include models with possibly more formulas, more rules, and more sentences that are true. In particular, $\mathbb{F}$ does not provide any proof induction principle for its theories. This is of little consequence; once reflected, a theory can be closed, and we will obtain an induction principle.

1.4 Organization

The general structure of this text roughly follows the research path that it describes. Chapter 2 begins with a brief survey of existing techniques for formal reasoning about syntax. Next in Chapter 3 we outline our approach to reasoning about syntax and in Chapter 4 we present a formal account of our syntax theory for simple terms based on a Martin-Löf style computational type theory and the implementation of that account in the MetaPRL theorem prover, and in Chapter 6 extend it to sequent judgments, where we rely on the use of "teleportation" to perform sequent context induction. Once the representation for terms is defined, we proceed to develop a representation of proofs and the corresponding principle for proof induction (Chapter 7). We proceed with some examples relating to the POPLmark challenge [12] (Chapter 8) where we also develop the principle for structural induction (Section 8.4). We finish with discussion and future work (Chapter 9).

Chapter 2

Previous Models of Reflection

Any form of logical reflection is preconditioned by the ability to reflect the syntax. Reflecting syntax in a logical system entails writing proof rules that express that reflection, i.e., establishing an inferential connection between the actual syntax used in the object logic $\mathbb{P}$ and the meta terms in the metalogic $\mathbb{M}$ supposedly referring to it. In this chapter, we take a look at previous methods that have been used to reflect the syntax. We also discuss more related work in Section 4.9.

In 1931 Gödel used reflection to prove his famous incompleteness theorem [35]. To express arithmetic in arithmetic itself, he assigned a unique number (a *Gödel number*) to each arithmetic formula. A Gödel number of a formula is essentially a numeric code of a string of symbols used to represent that formula.

A modern version of the Gödel's approach was used by Aitken et al. [7, 2, 3, 18] to implement reflection in the NuPRL theorem prover [19, 4]. A large part of this effort was essentially a reimplementation of the core of the NuPRL prover inside NuPRL's logical theory.

In Gödel's approach and its variations (including Aitken's one), a general mechanism that could be used for formalizing one logical theory in another is applied to formalizing a logical theory in itself. This can be very convenient for reasoning *about* reflection, but for our purposes it turns out to be extremely impractical. First, when formalizing a theory in itself using generic means, the identities between the theory being formalized and the one in which the formalization happens become very obfuscated, which makes it almost impossible to relate the reflected theory back to the

original one. Second, when one has a theorem proving system that already implements the logical theory in question, creating a completely new implementation of this logical theory inside itself is a very tedious redundant effort. Another practical disadvantage of the Gödel numbers approach is that it tends to blow up the size of the formulas; and iterated reflection would cause the blow-up to be iterated as well, making it exponential or worse.

A much more practical approach is being used in some programming languages, such as Lisp and Scheme. There, the common solution is for the implementation to *expose* its internal syntax representation to user-level code by the `quote` constructor (where `quote`(t) prevents the evaluation of the expression t). The problems outlined above are solved instantly by this approach: there is no blow-up, there is no repetition of structure definitions, there is even no need for verifying that the reflected part is equivalent to the original implementation since they are *identical*. Most Scheme implementations take this even further: the `eval` function is the internal function for evaluating a Scheme expression, which is exposed to the user level; Smith [57] showed how this approach can achieve an infinite tower of processors. A similar language with the quotation and antiquotation operators was introduced in [38].

This approach, however, violates the *congruence property* with respect to computation: if two terms are computationally equal then one can be substituted for the other in any context. For instance, although $2 * 2$ is equal to 4, the expressions "`2*2`" and "`4`" are syntactically different, thus we cannot substitute 4 for `2*2` in the expression `quote(2*2)`. The congruence property is essential in many logical reasoning systems, including NuPRL and MetaPRL.

A possible way to expose the internal syntax without violating the congruence property is to use the so-called quoted or shifted operators [1, 13, 14] rather than quoting the whole expression at once. For any operator *op* in the original language, we add the *quoted operator* (denoted as $\ulcorner op \urcorner$) to represent a term built with the operator *op*. For example, if the original language contains the constant "0" (which, presumably, represents the number 0), then in the reflected language, $\ulcorner 0 \urcorner$ would stand for the term that denotes the expression "0". Generally, the quoted operator has the

same arity as the original operator, but it is defined on syntactic terms rather than on semantic objects. For instance, while $*$ is a binary operator on numbers, $\ulcorner * \urcorner$ is a binary operator on terms. Namely, if t_1 and t_2 are syntactic terms that stand for expressions e_1 and e_2 respectively, then $t_1 \ulcorner * \urcorner t_2$ is a new syntactic term that stands for the expression $e_1 * e_2$. Thus, the quotation of the expression $1*2$ would be $\ulcorner 1 \urcorner \ulcorner * \urcorner \ulcorner 2 \urcorner$.

In general, the well-formedness (typing) rule for a quoted operator is the following:

$$\frac{t_1 \in \mathsf{Term} \qquad \cdots \qquad t_n \in \mathsf{Term}}{\ulcorner op \urcorner \{t_1; \cdots; t_n\} \in \mathsf{Term}} \tag{2.1}$$

where Term is a type of quoted terms. The requirement that the arity of op be n is implicit—since a quoted operator has the same arity as the original operator, the assumption is that the system will simply disallow typing something of the wrong arity.

Note that quotations can be iterated arbitrarily many times, allowing us to quote quoted terms. For instance, $\ulcorner\ulcorner 1 \urcorner\urcorner$ stands for the term that denotes the term that denotes the numeral 1.

Formalizing Languages with Bindings

Problems arise when quoting expressions that contain binding variables. For example, what is the quotation of $\lambda x. x$? There are several possible ways of answering this question. A commonly used approach [52, 25, 24, 8, 9] in logical frameworks such as Elf [51], LF [39], and Isabelle [49, 50] is to construct an object logic with a concrete $\ulcorner \lambda \urcorner$ operator that has a type like

$$(\mathsf{Term} \to \mathsf{Term}) \to \mathsf{Term} \quad \text{or} \quad (\mathsf{Var} \to \mathsf{Term}) \to \mathsf{Term}.$$

In this approach, the quoted $\lambda x. x$ might look like $\ulcorner \lambda \urcorner(\lambda x. x)$ and the quoted $\lambda x. 1$ might look like $\ulcorner \lambda \urcorner(\lambda x. \ulcorner 1 \urcorner)$. Note that in these examples the quoted terms have to make use of both the syntactic (i.e., quoted) operator $\ulcorner \lambda \urcorner$ and the semantic operator λ.

Exotic Terms. Naïve implementations of the above approach suffer from the well-known problem of exotic terms [22, 24]. The issue is that in general we cannot allow applying the $\ulcorner\lambda\urcorner$ operator to an arbitrary function that maps terms to terms (or variables to terms) and expect the result of such an application to be a "proper" reflected term.

Consider for example the following term

$$\ulcorner\lambda\urcorner(\lambda x.\,\mathbf{if}\ x = \ulcorner 1\urcorner\ \mathbf{then}\ \ulcorner 1\urcorner\ \mathbf{else}\ \ulcorner 2\urcorner).$$

It is relatively easy to see that it is not a real syntactic term and cannot be obtained by quoting an actual term. (For comparison, consider $\ulcorner\lambda\urcorner(\lambda x.\,\ulcorner\mathbf{if}\urcorner\ x\ \ulcorner{=}\urcorner\ \ulcorner 1\urcorner\ \ulcorner\mathbf{then}\urcorner\ \ulcorner 1\urcorner\ \ulcorner\mathbf{else}\urcorner\ \ulcorner 2\urcorner)$, which is a quotation of $\lambda x.\,\mathbf{if}\ x = 1\ \mathbf{then}\ 1\ \mathbf{else}\ 2)$.

How can one ensure that $\ulcorner\lambda\urcorner e$ denotes a "real" term and not an "exotic" one? That is, is it equal to the result of quoting an actual term of the object language? One possibility is to require e to be a *substitution function*; in other words it has to be equal to an expression of the form $\lambda x.\,t[x]$ where t is composed entirely of term constructors (i.e., quoted operators) and x, while using *destructors* (such as case analysis, the **if** operator used in the example above, etc.) is prohibited.

There are a number of approaches for enforcing the above restriction. One of them is the usage of logical frameworks with restricted function spaces [52,39], where λ-terms may contain only constructors. Another is to first formalize the larger type that does include exotic terms and then to define recursively a predicate describing the "validity" or "well-formedness" of a term [25, 24] thus removing the exotic terms from consideration. Yet another approach is to create a specialized type theory that combines the idea of restricted function spaces with a modal type operator [28,26,27]. There the case analysis is disallowed on objects of "pure" type T, but is allowed on objects of a special type $\diamond T$. This allows expressing both the restricted function space "$T_1 \rightarrow T_2$" and the unrestricted one "$(\diamond T_1) \rightarrow T_2$" within a single type theory.

Another way of regarding the problem of exotic terms is that it is caused by the attempt to give a semantic definition to a primarily syntactic property. A more

syntax-oriented approach was used by Barzilay et al. [15, 16, 14]. In Barzilay's approach, the quoted version of an operator that introduces a binding has the same *shape* (i.e., the number of subterms and the binding structure) as the original one and the variables (both the binding and the bound occurrences) are unaffected by the quotation. For instance, the quotation of $\lambda x.\, x$ is just $\ulcorner\lambda\urcorner x.\, x$.

The advantages of this approach include

- This approach is simple and clear.
- Quoted terms have the same structure as original ones, inheriting a lot of properties of the object syntax.
- In all the above approaches, the α-equivalence relation for quoted terms is inherited "for free." For example, $\ulcorner\lambda\urcorner x.\, x$ and $\ulcorner\lambda\urcorner y.\, y$ are automatically considered to be the same term.
- Substitution is also easy: we do not need to reimplement the substitution that renames binding variables to avoid the capture of free variables; we can use the substitution of the original language instead.

To prune exotic terms, Barzilay says that $\ulcorner\lambda\urcorner x.\, t[x]$ is a valid term when $\lambda x.\, t[x]$ is a *substitution function*. He demonstrates that it is possible to formalize this notion in a *purely syntactical* fashion. In this setting, the general well-formedness rule for quoted terms with bindings is the following:

$$\frac{\mathsf{is_subst}_k\,\{x_1,\cdots,x_k.\, t[\vec{x}]\} \qquad \cdots \qquad \mathsf{is_subst}_l\,\{z_1,\cdots,z_l.\, s[\vec{z}]\}}{\ulcorner op\urcorner\{x_1,\cdots,x_k.\, t[\vec{x}];\ \cdots;\ z_1,\cdots,z_l.\, s[\vec{z}]\} \in \mathsf{Term}} \tag{2.2}$$

where $\mathsf{is_subst}_n\,\{x_1,\cdots,x_n.\, t[\vec{x}]\}$ is the proposition that t is a substitution function over variables $x_1,\ldots,x_n$ (in other words, it is a syntactic version of the `Valid` predicate of [25, 24]). This proposition is defined syntactically by the following two rules:

$$\frac{}{\mathsf{is_subst}_n\,\{x_1,\cdots,x_n.\, x_i\}}$$

and

$$\frac{\mathsf{is_subst}_{n+k}\,\{x_1, \cdots, x_n, y_1, \cdots, y_k.\, t[\vec{x}; \vec{y}]\} \quad \vdots \quad \mathsf{is_subst}_{n+l}\,\{x_1, \cdots, x_n, z_1, \cdots, z_l.\, s[\vec{x}; \vec{z}]\}}{\mathsf{is_subst}_n\,\{x_1, \cdots, x_n.\, \ulcorner op \urcorner\{y_1, \cdots, y_k.\, t[\vec{x}; \vec{y}];\ \cdots;\ z_1, \cdots, z_l.\, s[\vec{x}; \vec{z}]\}\}}$$

In this approach the $\mathsf{is_subst}_n\,\{\}$ and $\ulcorner\lambda\urcorner$ operators are essentially *untyped* (in NuPRL type theory, the computational properties of untyped terms are at the core of the semantics; types are added on top of the untyped computational system).

Recursive Definition and Structural Induction Principle. A difficulty shared by both the straightforward implementations of the $(\mathsf{Term} \to \mathsf{Term}) \to \mathsf{Term}$ approach and by the Barzilay's one is the problem of recursively defining the Term type. We want to define the Term type as the smallest set satisfying rules (2.1) and (2.2). Note, however, that unlike rule (2.1), rule (2.2) is not monotonic in the sense that $\mathsf{is_subst}_k\,\{x_1, \cdots, x_k.\, t[\vec{x}]\}$ depends nonmonotonically on the Term type. For example, to say whether $\ulcorner\lambda\urcorner x.\, t[x]$ is a term, we should check whether t is a substitution function over x. It means at least that *for every* x in Term, $t[x]$ should be in Term as well. Thus we need to define the whole type Term before using (2.2), which produces a logical circle. Moreover, since $\ulcorner\lambda\urcorner$ has type $(\mathsf{Term} \to \mathsf{Term}) \to \mathsf{Term}$, it is hard to formulate the structural induction principle for terms built with the $\ulcorner\lambda\urcorner$ term constructor.

Variable-Length Lists of Binders. In Barzilay's approach, for each number n, $\mathsf{is_subst}_n\,\{\}$ is considered to be a separate operator—there is no way to quantify over n, and there is no way to express variable-length lists of binders. This issue of expressing the unbounded-length lists of binders is common to some of the other approaches as well.

Metareasoning. Another difficulty that is especially apparent in Barzilay's approach is that it only allows reasoning about *concrete* operators in concrete languages. This approach does not provide the ability to reason about operators *abstractly*; in particular, there is no way to state and prove metatheorems that quantify over oper-

ators or languages, much less *classes* of languages.

Although it is possible to solve the problems outlined above (and we will return to the discussion of some of those solutions in Section 4.9), our desire is to avoid these difficulties from the start. We propose a natural model of reflection that manages to work around these difficulties. In the next two chapters, we will show how to give a simple *recursive definition* of terms with binding variables, which *does not allow* the construction of exotic terms and *does allow* structural induction on terms.

Chapter 3

A Hybrid HOAS/de Bruijn Representation of the Syntax

As discussed in the previous chapter, representing syntax with bindings is hard. In the following two chapters, we propose a new approach to reflective metareasoning about languages with bindings. We present a theory of syntax that:

- in a natural way provides both a higher-order abstract syntax (HOAS) approach to bindings and a de Bruijn-style approach to bindings, with easy and natural translation between the two;
- provides a uniform HOAS-style approach to both bound and free variables that extends naturally to variable-length "vectors" of binders;
- permits metareasoning about languages—in particular, the operators, languages, open-ended languages, classes of languages etc. are all first-class objects that can be reasoned about both abstractly and concretely;
- comes with a natural induction principle for syntax that can be parameterized by the language being used;
- provides a natural mapping between the object syntax and metasyntax that is free of exotic terms, and allows mapping the object-level substitution operation directly to the metalevel one (i.e., β-reduction);
- is fully derived in a preexisting type theory in a theorem prover;

- is designed to serve as a foundation for a general reflective reasoning framework in a theorem prover;
- is designed to serve as a foundation for a programming language experimentation framework.

In this chapter we provide a conceptual overview of our approach; details are given in the next two chapters. We will show how we deal with concrete operators and abstract operators, and how we inductively define the type of all well-formed reflected terms.

3.1 Bound Terms

One of the key ideas of our approach is how we deal with terms containing free variables. We extend to free variables the principle that *variable names do not really matter*. In fact, we model free variables as *bindings* that can be arbitrarily α-renamed. Namely, we will write $\mathsf{bterm}\{x_1, \cdots, x_n.\, t[\vec{x}]\}$ for a term t over variables $x_1, \ldots, x_n$. For example, instead of term $x \ulcorner * \urcorner y$ we will use the term $\mathsf{bterm}\{x, y.\, x \ulcorner * \urcorner y\}$ when it is considered over variables x and y and $\mathsf{bterm}\{x, y, z.\, x \ulcorner * \urcorner y\}$ when it is considered over variables x, y and z. Free occurrences of x_i in $t[\vec{x}]$ are considered bound in $\mathsf{bterm}\{x_1, \cdots, x_n.\, t[\vec{x}]\}$ and two α-equal $\mathsf{bterm}\{\}$ expressions ("bterms") are considered to be *identical*.

Not every bterm is necessarily well-formed. We will define the type of terms in such a way as to eliminate exotic terms. Consider for example a definition of lambda-terms.

Example 3.1: *Define a set of reflected lambda-terms as the smallest set such that*

- $\mathsf{bterm}\{x_1, \cdots, x_n.\, x_i\}$, where $1 \leq i \leq n$, is a lambda-term (a variable);
- if $\mathsf{bterm}\{x_1, \cdots, x_n, x_{n+1}.\, t[\vec{x}]\}$ is a lambda-term, then

  $$\mathsf{bterm}\{x_1, \cdots, x_n.\, \ulcorner\lambda\urcorner x_{n+1}.\, t[\vec{x}]\}$$

  is also a lambda-term (an abstraction);

- if $\mathsf{bterm}\{x_1, \cdots, x_n.\, t_1[\vec{x}]\}$ and $\mathsf{bterm}\{x_1, \cdots, x_n.\, t_2[\vec{x}]\}$ are lambda-terms, then

$$\mathsf{bterm}\{x_1, \cdots, x_n.\, \ulcorner\mathsf{apply}\urcorner\{t_1[\vec{x}]; t_2[\vec{x}]\}\}$$

  is also a lambda-term (an application).

In a way, bterms could be understood as an explicit coding for Barzilay's substitution functions. And indeed, some of the basic definitions are quite similar. The notion of bterms is also very similar to that of *local variable contexts* [32].

3.2 Terminology

Before we proceed further, we need to define some terminology.

Definition 3.1: *We change the notion of* subterm *so that the subterms of a bterm are also bterms. For example, the immediate subterms of* $\mathsf{bterm}\{x, y.\, x \ulcorner * \urcorner y\}$ *are* $\mathsf{bterm}\{x, y.\, x\}$ *and* $\mathsf{bterm}\{x, y.\, y\}$*; the immediate subterm of* $\mathsf{bterm}\{x.\, \ulcorner\lambda\urcorner y.\, x\}$ *is* $\mathsf{bterm}\{x, y.\, x\}$*.*

Definition 3.2: *We call the number of outer binders in a bterm expression its* binding depth. *Namely, the* binding depth *of the bterm* $\mathsf{bterm}\{x_1, \cdots, x_n.\, t[\vec{x}]\}$ *is* n*.*

Definition 3.3: *Throughout the rest of the thesis we use the notion of operator* shape. *The* shape *of an operator is a list of natural numbers each stating how many new binders the operator introduces on the corresponding subterm. The length of the shape list is therefore the arity of the operator. For example, the shape of the* $+$ *operator is* $[0; 0]$ *and the shape of the* λ *operator is* $[1]$*.*

The mapping from operators to shapes is also sometimes called a *binding signature* of a language [32, 54].

Definition 3.4: *Let* h *be a shape of* $[d_1; \cdots; d_N]$*, and let* l *be a list of bterms* $[b_1; \cdots; b_M]$*. We say that* l *is* compatible *with* h *at depth* n *when*

1. $N = M$;

2. the binding depth of bterm b_j is $n + d_j$ for each $1 \leq j \leq N$.

3.3 Abstract Operators

Expressions of the form $\mathsf{bterm}\{\vec{x}.\ulcorner op \urcorner\{\cdots\}\}$ can only be used to express syntax with *concrete* operators. In other words, each expression of this form contains a specific constant operator $\ulcorner op \urcorner$. However, we would like to reason about operators abstractly; in particular, we want to make it possible to have variables of some type "$\mathcal{O}$" that can be quantified over and used in the same manner as operator constants. In order to address this we use explicit term constructors in addition to $\mathsf{bterm}\{\vec{x}.\ulcorner op \urcorner\{\cdots\}\}$ constants.

The expression $\mathsf{B}\{n; ``\ulcorner op \urcorner"; btl\}$, where "$\ulcorner op \urcorner$" is some encoding of the quoted operator $\ulcorner op \urcorner$, stands for a bterm with binding depth n, operator $\ulcorner op \urcorner$ and subterms *btl*. Namely,

$$\begin{aligned} \mathsf{B}\{n; \ulcorner op \urcorner;\ & \mathsf{bterm}\{x_1, \cdots, x_n, \vec{y_1}.\, t_1[\vec{x}; \vec{y_1}]\} :: \cdots :: \\ & \mathsf{bterm}\{x_1, \cdots, x_n, \vec{y_k}.\, t_k[\vec{x}; \vec{y_k}]\} :: \mathtt{nil}\} \end{aligned}$$

is

$$\mathsf{bterm}\{x_1, \cdots, x_n.\ulcorner op \urcorner\{\vec{y_1}.\, t_1[\vec{x}; \vec{y_1}]; \cdots; \vec{y_k}.\, t_k[\vec{x}; \vec{y_k}]\}\}\,.$$

Here, `nil` is the empty list and :: is the right-associative list `cons` operator and therefore the expression $b_1 :: \cdots :: b_n :: \mathtt{nil}$ represents the concrete list $[b_1; \cdots; b_n]$.

Note that if we know the shape of the operator *op* and we know that the B expression is well-formed (or, more specifically, if we know that *btl* is compatible with the shape of *op* at depth n), then it would normally be possible to deduce the value of n (since n is the difference between the binding depth of any element of the list *btl* and the corresponding element of the $\mathsf{shape}(op)$ list). There are two reasons, however, for supplying n explicitly:

- When *btl* is empty (in other words, when the arity of *op* is 0), the value of n cannot be deduced this way and still needs to be supplied somehow. One could

consider 0-arity operators to be a special case, but this results in a significant loss of uniformity.

- When we do *not* know whether a B expression is necessarily well-formed (and as we will see it is often useful to allow this to happen), then a lot of definitions and proofs are greatly simplified when the binding depth of B expressions is explicitly specified.

Using the B constructor and a few other similar constructors that will be introduced later, it becomes easy to reason abstractly about operators. Indeed, the second argument to B can now be an arbitrary expression, not just a constant. This has a cost of making certain definitions slightly more complicated. For example, the notion of "compatible with *sh* at depth n" now becomes an important part of the theory and will need to be explicitly formalized. However, this is a small price to pay for the ability to reason abstractly about operators, which easily extends to reasoning abstractly about languages, classes of languages and so forth.

3.4 Inductively Defining the Type of Well-Formed Bterms

There are two equivalent approaches to inductively defining the general type (set) of all well-formed bterms. The first one follows the same idea as in Example 3.1:

- $\mathsf{bterm}\{x_1, \cdots, x_n.\, x_i\}$ is a well-formed bterm for $1 \leq i \leq n$;
- $\mathsf{B}\{n; op; btl\}$ is a well-formed bterm when op is a well-formed quoted operator and btl is a list of well-formed bterms that is compatible with the shape of op at some depth n.

If we denote $\mathsf{bterm}\{x_1, \cdots, x_l, y, z_1, \cdots, z_r.\, y\}$ as $\mathsf{V}\{l; r\}$, we can restate the base case of the above definition as "$\mathsf{V}\{l; r\}$, where l and r are arbitrary natural numbers, is a well-formed bterm." Once we do this it becomes apparent that the above definition

has a lot of similarities with de Bruijn-style indexing of variables [23]. Indeed, one might call the numbers l and r the *left and right indices of the variable* $\mathsf{V}\{l; r\}$.

It is possible to provide an alternate definition that is equivalent to the definition above, but is closer to pure HOAS. We define the following terms.

- $\lambda_b x.\, t[x]$, where t is a well-formed substitution function, is a well-formed bterm (the λ_b operation increases the binding depth of t by one by adding x to the beginning of the list of t's outer binders).

- $\mathsf{T}\{op; btl\}$, where op is a well-formed quoted operator, and btl is a list of well-formed bterms that is compatible with the shape of op at depth 0, is a well-formed bterm (of binding depth 0).

Other than better capturing the idea of HOAS, the latter definition also makes it easier to express the reflective correspondence between the metasyntax (the syntax used to express the theory of syntax, namely the one that includes the operators B, λ_b, etc.) and the meta-metasyntax (the syntax that is used to express the theory of syntax and the underlying theory, in other words, the syntax that includes the second-order notations.) Namely, provided that we define the $\mathsf{subst}\{bt; t\}$ operation to compute the result of substituting a closed term t for the first outer binder of the bterm bt, we can state that

$$\mathsf{subst}\{\lambda_b x.\, t_1[x]; t_2\} \quad \leftrightarrow \quad t_1[t_2] \tag{3.1}$$

where t_1 and t_2 are literal second-order variables, and $\leftrightarrow$ is the computational equality relation[1]. In other words, we can state that the substitution operator subst and the implicit second-order substitution in the "meta-metalanguage" are equivalent.

The downside of the alternate definition is that it requires defining the notion of "being a substitution function."

[1]Computational equality means that when $a \leftrightarrow b$ is true, a may be replaced with b in an arbitrary context. Examples include beta-reduction $\lambda x.\, t[x]\ y \leftrightarrow t[y]$, arithmetical equalities ($1 + 2 \leftrightarrow 3$), and definitional equality (an abstraction is considered to be computationally equal to its definition).

3.5 Our Approach

In our work we combine the advantages of both approaches outlined above. In the next chapter we present a theory that includes both the HOAS-style operations (λ_b, T) and the de Bruijn-style ones (V, B). Our theory also allows deriving the equivalence (3.1). In our theory the definition of the basic syntactic operations is based on the HOAS-style operators; however, the recursive definition of the type of well-formed syntax is based on the de Bruijn-style operations. Our theory also includes support for variable-length lists of binders.

Chapter 4

Formal Implementation of the Syntax in a Theorem Prover

In the following two chapters we describe how the syntax theory is formally defined and derived in the NuPRL-style Computational Type Theory in the MetaPRL Theorem Prover. In this chapter we develop the representation for simple terms, and then we extend it to sequent judgments in the next chapter. For brevity, we will present a slightly simplified version of our implementation; full details are available in [42].

4.1 Computations and Types

In our work we make heavy usage of the fact that our type theory allows us to define computations *without* stating upfront (or even knowing) what the relevant types are. In NuPRL-style type theories (which some even dubbed "untyped type theory"), one may define arbitrary recursive functions (even potentially nonterminating ones). Only when proving that such function belongs to a particular type, one may have to prove termination. See Allen [5, 6] for a semantics that justifies this approach.

The formal definition of the syntax of terms consists of two parts:

- The definition of untyped term constructors and term operations, which includes both HOAS-style operations and de Bruijn-style operations. We can establish most of the reduction properties without explicitly giving types to all the operations.

- The definition of the type of terms. We will define the type of terms as the type that contains all terms that can be legitimately constructed by the term constructors.

4.2 HOAS Representation

At the core of our term syntax definition are two basic HOAS-style constructors.

- $\lambda_b x.\, t[x]$ is meant to represent a quoted term with a free variable x. The intended semantics (which will not become explicit until later) is that $\lambda_b x.\, t[x]$ will only be considered well-formed when t is a substitution function.

  Internally, $\lambda_b x.\, t[x]$ is implemented simply as the left injection[1] of a lambda abstraction.

  $$\lambda_b x.\, t[x] \quad := \quad \mathsf{inl}\{\lambda x.\, t[x]\}$$

  This definition is abstract and is used only to prove the properties of the two destructors presented below; it is never used outside of this section (Section 4.2).

- $\mathsf{T}\{op; ts\}$ represents a closed quoted term (i.e., a bterm of binding depth 0) with operator op and the list of subterms ts. It will be considered well-formed when op is an operator and ts is a list of terms that is *compatible* with the shape of op at depth 0. For example, $\mathsf{T}\{\ulcorner\lambda\urcorner; [\lambda_b x.\, x]\}$ is $\ulcorner\lambda x.\, x\urcorner$.

  Internally, $\mathsf{T}\{op; ts\}$ is implemented as the right injection of a pair of the operator and the list of subterms.

  $$\mathsf{T}\{op; ts\} \quad := \quad \mathsf{inr}\{\langle op, ts\rangle\}$$

  Again, this definition is never used outside of this section.

---

[1] The left injection of a, $\mathsf{inl}\{a\}$, is an element of some union type $A \cup B$, if a has type A. Similarly, the right injection of b, $\mathsf{inr}\{b\}$, is also an element of $A \cup B$, if b has type B. The $\mathsf{decide}\,\{x; y.\, t_1[y]; z.\, t_2[z]\}$ term decides the handedness of the term x in $A \cup B$, for which we usually use its pretty form "**match** x **with** $\mathsf{inl}\{y\} \rightarrow t_1[y] \mid \mathsf{inr}\{z\} \rightarrow t_2[z]$."

We also implement two destructors:

- $\mathsf{subst}\{bt; t\}$ (with shorthand $bt@t$) is meant to represent the result of substituting term t for the first variable of the bterm bt. Internally, $\mathsf{subst}\{bt; t\}$ is defined as

$$\begin{array}{rcl} \mathsf{subst}\{bt; t\} & := & \textbf{match}\, bt\, \textbf{with} \\ & & \mid \mathsf{inl}\{f\} \to f\ t \\ & & \mid \mathsf{inr}\{\langle o, s\rangle\} \to \bullet \end{array}$$

  where the symbol $\bullet$ is used to denote a dummy or error term. That is, a substitution is defined only for binders.

  We derive the following property of this substitution operation:

$$(\lambda_b x.\, t_1[x])@t_2 \quad \leftrightarrow \quad t_1[t_2]$$

  where $\leftrightarrow$ is the computational equality relation and t_1 and t_2 are completely arbitrary, perhaps even ill-typed. This derivation is the only place where the internal definition of $\mathsf{subst}\{bt; t\}$ is used.

  Note that the above equality is exactly the "reflective property of substitution" (3.1) that was one of the design goals for our theory.

- $\mathsf{weak_dest}\,\{bt; bcase; op, ts.\ mkt_case[op; ts]\}$ is the destructor designed to provide a way to find out whether bt is a $\lambda_b_$ term or a $\mathsf{T}\{op; ts\}$ term and to "extract" the op and ts in the latter case. In the rest of this thesis we will use the "pretty-printed" form for $\mathsf{weak_dest}$—"$\textbf{match}\, bt\ \textbf{with}\ \lambda_b_ \ \to\ bcase\ \mid\ \mathsf{T}\{op; ts\}\ \to$

$$
\begin{array}{rcll}
r & ::= & x & \text{first-order variables} \\
 & | & z[r_1; \cdots ; r_n] & \text{second-order meta-variables} \\
 & | & \mathsf{T}\{\ulcorner op \urcorner; [r_1; \cdots ; r_n]\} & \text{reflected terms} \\
 & | & \lambda_b x.\, r & \text{binding} \\
 & | & \Gamma_r \ulcorner\vdash\urcorner r & \text{reflected sequents} \\
 \\
\Gamma_r & ::= & q_1; \cdots ; q_n & \text{reflected sequent contexts} \\
q & ::= & X[r_1; \cdots ; r_n] & \text{context meta-variables} \\
 & | & x : r & \text{hypothesis binding}
\end{array}
$$

Figure 4.1: Reflected terms in HOAS form

$mkt_case[op; ts]$." We also derive the following properties of weak_dest:

$$
\begin{pmatrix}
\textbf{match}\, \lambda_b x.\, t[x]\ \textbf{with} \\
\quad \lambda_{b_} \rightarrow bcase \\
\mid \mathsf{T}\{op; ts\} \rightarrow mkt_case[op; ts]
\end{pmatrix} \leftrightarrow bcase
$$

$$
\begin{pmatrix}
\textbf{match}\, \mathsf{T}\{op; ts\}\ \textbf{with} \\
\quad \lambda_{b_} \rightarrow bcase \\
\mid \mathsf{T}\{o; t\} \rightarrow mkt_case[o; t]
\end{pmatrix} \leftrightarrow mkt_case[op; ts]
$$

Figure 4.1 shows the set of term representatives that are used to represent reflected terms in HOAS form. This includes the usual first-order, second-order, and context variables. It also includes the two new term representatives we introduced: $\mathsf{T}\{\ulcorner op \urcorner; [r_1; \cdots ; r_n]\}$ which represents a reflected term with operator op and subterms $r_1, \ldots, r_n$, and $\lambda_b x.\, r$ which introduces bindings.

Sequents are shown in Figure 4.1 for reference. However, quoted sequents are treated as noncanonical forms. That is, sequents are eliminated from the representation through an appropriate coding, as will be discussed in Chapter 6.

4.3 Vector HOAS Operations

As we have mentioned at the end of Chapter 2, some approaches to reasoning about syntax make it hard or even impossible to express arbitrary-length lists of binders. In our approach, we address this challenge by allowing operators where a single binding in the metalanguage stands for a list of object-level bindings. In particular, we allow representing $\lambda_b x_1.\, \lambda_b x_2.\, \cdots .\, \lambda_b x_n.\, t[x_1; \ldots; x_n]$ as $\mathsf{vbnd}\{n; x.\, t[\mathsf{nth}\{x;1\}; \ldots; \mathsf{nth}\{x;n\}]\}$, where "$\mathsf{nth}\{l;i\}$" is the "$i$th element of the list l" function.

We define the following vector-style operations:

- $\mathsf{vbnd}\{n; x.\, t[x]\}$ represents a "telescope" of nested λ_b operations. It is defined by induction[2] on the natural number n as follows:

$$\begin{array}{rcl} \mathsf{vbnd}\{0; x.\, t[x]\} & := & t[\mathtt{nil}] \\ \mathsf{vbnd}\{n+1; x.\, t[x]\} & := & \lambda_b v.\, \mathsf{vbnd}\{n; x.\, t[v :: x]\} \end{array}$$

  We also introduce $\mathsf{vbnd}\{n; t\}$ as a simplified notation for $\mathsf{vbnd}\{n; x.\, t\}$ when t does not have free occurrences of x.

- $\mathsf{vsubst}\{bt; ts\}$ is a "vector" substitution operation that is meant to represent the result of simultaneous substitution of the terms in the ts list for the first $|ts|$ variables of the bterm bt (here $|l|$ is the length of the list l). It is defined by induction on the list ts as follows:

$$\begin{array}{rcl} \mathsf{vsubst}\{bt; \mathtt{nil}\} & := & bt \\ \mathsf{vsubst}\{bt; t :: ts\} & := & \mathsf{vsubst}\{bt@t; ts\} \end{array}$$

[2] Our presentation of the inductive definitions is slightly simplified by omitting some minor technical details. See Appendix B for complete details.

Below are some of the derived properties of these operations:[3]

$$\lambda_b v.\, t[v] \leftrightarrow \mathsf{vbnd}\{1; l.\, \mathtt{hd}(l)\} \tag{4.1}$$

$$\forall m, n \in \mathbb{N}.\ \big(\mathsf{vbnd}\{m+n; x.\, t[x]\} \leftrightarrow \mathsf{vbnd}\{m; y.\, \mathsf{vbnd}\{n; z.t[y \circledcirc z]\}\}\big) \tag{4.2}$$

$$\forall l \in \mathtt{List}.\ (\mathsf{vsubst}\{\mathsf{vbnd}\{|l|; v.\, t[v]\}; l\} \leftrightarrow t[l]) \tag{4.3}$$

$$\begin{aligned} &\forall l \in \mathtt{List}.\ \forall n \in \mathbb{N}.\big(n \geq |l| \Rightarrow \\ &\quad (\mathsf{vsubst}\{\mathsf{vbnd}\{n; v.\, t[v]\}; l\} \leftrightarrow \mathsf{vbnd}\{n - |l|; v.\, t[l \circledcirc v]\})\big) \end{aligned} \tag{4.4}$$

$$\forall n \in \mathbb{N}.\ \big(\mathsf{vbnd}\{n; l.\, \mathsf{vsubst}\{\mathsf{vbnd}\{n; v.\, t[v]\}; l\}\} \leftrightarrow \mathsf{vbnd}\{n; l.\, t[l]\}\big) \tag{4.5}$$

where "hd" is the list "head" operation, "$\circledcirc$" is the list append operation, "List" is the type of arbitrary lists (the elements of a list do not have to belong to any particular type), $\mathbb{N}$ is the type of natural numbers, and all the variables that are not explicitly constrained to a specific type stand for arbitrary expressions.

Equivalence (4.2) allows the merging and splitting of vector bind operations. Equivalence (4.3) is a vector variant of equivalence (3.1). Equivalence (4.5) is very similar to equivalence (4.3) applied in the $\mathsf{vbnd}\{n; l.\ \cdots\}$ context, except that (4.5) does not require l to be a member of any special type.

4.4 de Bruijn Representation

As mentioned in the introduction, defining an induction principle for the HOAS representation can be difficult. The issue is how to deal with bindings. In this section we develop a de Bruijn representation that is computationally equivalent to the HOAS representation. This will then lead us to a type definition and induction principle.

The de Bruijn representation introduces two new constructors based on the HOAS constructors defined in the previous two sections.

- The term $\mathsf{V}\{l; r\}$ represents a variable; the natural number l is called the *left*

[3] Note that all the proofs of these properties are carried out formally in the MetaPRL system. The complete list of theorems we derived can be found online at [42].

binding index, and r the *right binding index*. It is defined as

$$\mathsf{V}\{l; r\} \quad := \quad \mathsf{vbnd}\{l; \lambda_b v.\, \mathsf{vbnd}\{r; v\}\}\,.$$

We can see that this definition indeed corresponds to the informal $\mathsf{bterm}\{x_1, \cdots, x_l, y, z_1, \cdots, z_r.\, y\}$ definition given in Section 3.4. A variable acts as a selector, with the following equivalence.

$$\mathsf{V}\{l; r\}@t_1@\cdots@t_{l+1}@\cdots@t_{l+r+1} \leftrightarrow t_{l+1}$$

We can derive the following equivalences for variables.

$$\begin{aligned}
\lambda_b x.\, \mathsf{V}\{l; r\} &\leftrightarrow \mathsf{V}\{l+1; r\} \\
\mathsf{V}\{0; r\}@t &\leftrightarrow t \\
\mathsf{V}\{l+1; r\}@t &\leftrightarrow \mathsf{V}\{l; r\}
\end{aligned}$$

- The term $\mathsf{B}\{n; op; ts\}$ is called a *bound term*. It is meant to compute a bterm of binding depth n, with operator op, and with ts as its subterms. This operation is defined by induction on natural number n as follows.

$$\begin{aligned}
\mathsf{B}\{0; op; ts\} &:= \mathsf{T}\{op; ts\} \\
\mathsf{B}\{n+1; op; ts\} &:= \lambda_b v.\, \mathsf{B}\{n; op; \mathsf{map}\ \lambda t.\, t@v\ \ ts\}
\end{aligned}$$

Informally,

$$\begin{aligned}
\mathsf{B}\{n; op; [s_1; \cdots; s_m]\} \quad \leftrightarrow \quad \lambda_b x_1. \cdots. \lambda_b x_n.\, \mathsf{T}\{op;\ & [s_1@x_1@\cdots@x_n; \\
& \cdots; \\
& s_m@x_1@\cdots@x_n]\}.
\end{aligned}$$

The two commutativity properties below indicate the computational behavior

of the bound terms.

$$\mathsf{B}\{n+1; op; [s_1; \cdots; s_m]\}@r \leftrightarrow \mathsf{B}\{n; op; [s_1@r; \cdots; s_m@r]\}$$
(subst-commutes)

$$\lambda_b x.\, \mathsf{B}\{n; op; [s_1[x]; \cdots; s_m[x]]\} \leftrightarrow \mathsf{B}\{n+1; op; [\lambda_b x.\, s_1[x]; \cdots; \lambda_b x.\, s_m[x]]\}$$
(bind-commutes)

The subst-commutes property indicates that to substitute into a bound term, the substitution should be applied to each of the subterms. Note that an outer binder is removed as well.

The bind-commutes property defines the conversion to de Bruijn representation, which is obtained by "pushing" all binds inward as far as possible. For example, consider the following term fragment, shown first in informal HOAS notation.

$$\ulcorner \lambda z.\, \mathbf{match}\, z\ \mathbf{with}\ \mathsf{inl}\{x\} \to x \mid \mathsf{inr}\{y\} \to z \urcorner$$

To write it formally, we choose the operator lambda for λ, and decide for **match**, which gives the following concrete representation.

$$\ulcorner \mathsf{lambda}\,\{z.\, \mathsf{decide}\{z; x.\, x; y.\, z\}\} \urcorner$$

By definition of $\mathsf{T}\{;\,,\}$ we obtain the following term in HOAS form, with binders in the expected positions.

$$\mathsf{T}\{\ulcorner\mathsf{lambda}\urcorner; [\lambda_b z.\, \mathsf{T}\{\ulcorner\mathsf{decide}\urcorner; [z; \lambda_b x.\, x; \lambda_b y.\, z]\}]\}$$

This term is equivalent to the following term in de Bruijn form by definitions of $\mathsf{B}\{;\ \mathrm{a};\ \mathrm{n}\}\mathrm{d}\ \mathsf{V}\{;,\}$ with the binder pushed inwards.

$$\mathsf{B}\{0; \ulcorner\mathsf{lambda}\urcorner; [\mathsf{B}\{1; \ulcorner\mathsf{decide}\urcorner; [\mathsf{V}\{0;0\}; \mathsf{V}\{1;0\}; \mathsf{V}\{0;1\}]\}]\}$$

The above HOAS and de Bruijn forms are computationally equivalent. In a metalogic like computational type theory, where computational equivalence is a congruence, the two forms are formally equivalent in all contexts and can be interchanged at will. The HOAS form corresponds closely to the original expression, while the de Bruijn form is variable free.

de Bruijn-style Destructors

We also define a number of de Bruijn-style destructors, i.e., operations that compute various de Bruijn-style characteristics of a bterm. Since the V and B constructors are defined in terms of the HOAS constructors, the destructors have to be defined in terms of HOAS operations as well. Because of this, these definitions are often far from straightforward.

It is important to emphasize that the tricky definitions that we use here are only needed to establish the basic properties of the operations we defined. Once the basic theory is complete, we can raise the level of abstraction and no usage of this theory will ever require using any of these definitions, being aware of these definitions, or performing similar tricks again.

- $\mathsf{bdepth}\{t\}$ computes the binding depth of term t. It is defined recursively using the Y combinator.

$$Y \left(\begin{array}{l} \lambda f.\, \lambda b.\ \textbf{match}\, b\, \textbf{with} \\ \qquad\qquad \lambda_{b_} \to 1 + f\,(b @ \mathsf{T}\{\ulcorner \mathsf{true} \urcorner; \mathtt{nil}\}) \\ \qquad\quad\; \mid \mathsf{T}\{_;_\} \to 0 \end{array} \right) t$$

In effect, this recursive function strips the outer binders from a bterm one by one using substitution (note that here we can use an arbitrary T expression as a second argument for the substitution function; the arguments to T do not have to have the "correct" type) and counts the number of times it needs to do this before the outermost T is exposed.

Informally,

$$\begin{aligned}\mathsf{bdepth}\{t\} \quad := \quad & \textbf{match}\, t\, \textbf{with} \\ & \mid \mathsf{inl}\{f\} \rightarrow \mathsf{bdepth}\{f\ \bullet\} + 1 \\ & \mid \mathsf{inr}\{\langle o, s\rangle\} \rightarrow 0.\end{aligned}$$

We derive the following properties of bdepth.

$$\forall l, r \in \mathbb{N}. \left(\mathsf{bdepth}\{\mathsf{V}\{l; r\}\} \leftrightarrow (l + r + 1)\right)$$

$$\forall n \in \mathbb{N}. \left(\mathsf{bdepth}\{\mathsf{B}\{n; op; ts\}\} \leftrightarrow n\right)$$

Note that the latter equivalence only requires n to have the "correct" type, while op and ts may be arbitrary. Since the bdepth operator is needed for defining the type of well-formed bterms, at this point we are not yet able to express what the "correct" type for ts would be.

- $\mathsf{left}\{t\}$ is designed to compute the "left index" of a V expression. It is defined as

$$Y \left(\begin{array}{l} \lambda f.\, \lambda b.\, \lambda n.\ \textbf{match}\, b\, \textbf{with} \\ \qquad\qquad \lambda_{b_} \rightarrow 1 + f\,(b@\mathsf{T}\{n; \mathtt{nil}\})\,(n+1) \\ \qquad\qquad \mid \mathsf{T}\{n'; _\} \rightarrow n' \end{array} \right) t\ 0.$$

In effect, this recursive function substitutes $\mathsf{T}\{0; \mathtt{nil}\}$ for the first binding of t, $\mathsf{T}\{1; \mathtt{nil}\}$ for the second one, $\mathsf{T}\{2; \mathtt{nil}\}$ for the next one, and so forth. Once all the binders are stripped and a $\mathsf{T}\{n; \mathtt{nil}\}$ is exposed, n is the index we were looking for. Note that here we intentionally supply T with an argument of a "wrong" type ($\mathbb{N}$ instead of the operator type); we could have avoided this, but then the definition would have been significantly more complicated.

As expected, we derive that

$$\forall l, r \in \mathbb{N}. \left(\mathsf{left}\{\mathsf{V}\{l; r\}\} \leftrightarrow l\right).$$

- $\mathsf{right}\{t\}$ computes the "right index" of a V expression. It is trivial to define in terms of the previous two operators.

$$\mathsf{right}\{t\} \quad := \quad \mathsf{bdepth}\{t\} - \mathsf{left}\{t\} - 1$$

- $\mathsf{subterms}\{t\}$ is designed to recover the last argument of a B expression. The definition is rather technical and complicated. We first define the number of subterms of a term.

$$\mathtt{num_subterms}\{t\} \quad := \quad Y \begin{pmatrix} \lambda f.\, \lambda b.\ \textbf{match}\, b\, \textbf{with} \\ \qquad \lambda_{b_} \rightarrow f\ (b @ \mathsf{T}\{\ulcorner \mathsf{true} \urcorner; \mathtt{nil}\}) \\ \mid \mathsf{T}\{_; l\} \rightarrow |l| \end{pmatrix} t$$

Then we compute the nth subterm of a term.

$$\mathtt{nth_subterm}\{t; n\} \quad := \quad Y \begin{pmatrix} \lambda f.\, \lambda b.\ \textbf{match}\, b\, \textbf{with} \\ \qquad \lambda_{b_} \rightarrow \lambda_b v.\, f\ (b @ v) \\ \mid \mathsf{T}\{_; l\} \rightarrow \mathsf{nth}\{l; n\} \end{pmatrix} t$$

The definition for subterms of a term follows trivially.

$$\mathsf{subterms}\{t\} \quad := \quad \mathsf{lof}\{i.\, \mathtt{nth_subterm}\{t; i\}; \mathtt{num_subterms}\{t\}\}$$

where

$$\mathsf{lof}\{i.\, f[i]; n\} \quad := \quad [f[0]; \cdots ; f[n-1]]$$

The main property of the $\mathsf{subterms}$ operation that we derive is

$$\begin{array}{l} \forall n \in \mathbb{N}.\ \forall ts \in \mathtt{List}. \\ \quad (\mathsf{subterms}\{\mathsf{B}\{n; op; ts\}\} \leftrightarrow \mathsf{map}\ (\lambda b.\, \mathsf{vbnd}\{n; v.\, \mathsf{vsubst}\{b; v\}\})\ ts)\,. \end{array}$$

The right-hand side of this equivalence is not quite the plain "*ts*" that one might have hoped to see here. However, when *ts* is a list of bterms with binding depths at least n, which is necessarily the case for any well-formed $\mathsf{B}\{n; op; ts\}$, equivalence (4.5) will allow simplifying this right-hand side to the desired *ts*.

- $\mathsf{getop}\{t; op\}$ is an operation such that

$$\begin{aligned} & \forall n \in \mathbb{N}.\, (\mathsf{getop}\{\mathsf{B}\{n; op; ts\}; op'\} \leftrightarrow op) \\ \text{and} \quad & \forall l, r \in \mathbb{N}.\, (\mathsf{getop}\{\mathsf{V}\{l; r\}; op\} \leftrightarrow op)\,. \end{aligned}$$

  Its definition is similar to that of $\mathsf{left}\{\}$.

- $\mathsf{isvar}\{t\}$ decides whether a bterm is a V or a B. It is defined as

$$\mathsf{isvar}\{t\} \quad := \quad \mathsf{getop}\{t; \ulcorner\mathsf{true}\urcorner\} \neq \mathsf{getop}\{t; \ulcorner\mathsf{false}\urcorner\}.$$

- $\mathsf{dest_bterm}\,\{t; l, r.\; vcase[l; r];\; d, op, ts.\; op_case[d; op; ts]\}$ is designed to extract all the components of the de Bruijn-like representation of a bterm. In the rest of this thesis we will use the "pretty-printed" form for $\mathsf{dest_bterm}$—"**match** t **with** $\mathsf{V}\{l; r\} \rightarrow vcase[l; r] \mid \mathsf{B}\{d; op; ts\} \rightarrow op_case[d; op; ts]$". It is defined as

$$\begin{aligned} \textbf{if } \mathsf{isvar}\{t\} \quad & \textbf{then } vcase[\mathsf{left}\{t\}\,; \mathsf{right}\{t\}] \\ & \textbf{else } op_case[\mathsf{bdepth}\{t\}; \mathsf{getop}\{t; \bullet\}; \mathsf{subterms}\{t\}]. \end{aligned}$$

It should be noted that the de Bruijn representatives $\mathsf{V}\{n; m\}$ and $\mathsf{B}\{n; o; s\}$ and the corresponding destructors are defined in terms of the HOAS operations. They do not add any additional properties that were not already provable in the metalogic—in other words, the extension is conservative. In our implementation, all equivalences have been formally verified.

4.5 Operators

The type $\mathcal{O}$ of operators is defined abstractly. We only require that operators have decidable equality and that there exist a function of type $\mathcal{O} \to \mathbb{N}$ `list` that computes operators' *shapes*. The term $\mathsf{shape}\{op\}$ represents a list of natural numbers specifying the binding arities of the corresponding subterms. The following are some examples.

$$\begin{aligned} \mathsf{shape}\{\ulcorner\mathsf{lambda}\urcorner\} &= [1] \\ \mathsf{shape}\{\ulcorner\mathsf{let}\urcorner\} &= [0;1] \\ \mathsf{shape}\{\ulcorner\mathsf{add}\urcorner\} &= [0;0] \\ \mathsf{shape}\{\ulcorner\mathsf{decide}\urcorner\} &= [0;1;1] \end{aligned}$$

4.6 The Type of Reflected Terms

Naturally, since we wish to reason about programs in type theory, we should give a type that specifies the collection of reflected terms. For this we rely on the *function image* type, originated by Nogin and Kopylov [46]. The image type has the form $\mathsf{Img}\{A; x.\, f[x]\}$ where $f[x]$ is an arbitrary function, A is its domain, and the type includes exactly those terms $f[a]$ for each $a \in A$.

While it would be straightforward to define a type containing all de Bruijn-style terms, we plan to be a bit more careful and include only those where the binding depths are "sensible." For example, suppose our language contains a term called let with the expected form.

$$\ulcorner \mathbf{let}\, x = t_1 \;\mathbf{in}\; t_2[x] \urcorner$$

$$\leftrightarrow$$

$$\ulcorner \mathbf{let}\, \{t_1; x.\, t_2[x]\} \urcorner$$

$$\leftrightarrow$$

$$\mathsf{T}\{\ulcorner\mathsf{let}\urcorner; [\ulcorner t_1 \urcorner; \lambda_b x.\, \ulcorner t_2[x] \urcorner]\}$$

Type definitions

$$
\begin{array}{rcl}
\mathsf{dom}\{T\} & := & \mathbb{N} * \mathbb{N} + (\Sigma n\colon \mathbb{N}.\ \Sigma o\colon \mathcal{O}.\ \{s\colon T\ \mathtt{list} \mid \mathsf{compatible}\{n;\mathsf{shape}\{o\};s\}\})^4 \\
f(t) & := & \mathbf{match}\, t\ \mathbf{with}\ \ \mathsf{inl}\{\langle n,m\rangle\} \to \mathsf{V}\{n;m\} \mid \mathsf{inr}\{\langle n,o,s\rangle\} \to \mathsf{B}\{n;o;s\} \\
 & & \\
T_0 & := & \mathsf{void} \\
T_{i+1} & := & \mathsf{Img}\{\mathsf{dom}\{T_i\}; x.\, f(x)\} \\
 & & \\
\mathsf{BTerm} & := & \bigcup_{i\in\mathbb{N}} T_i
\end{array}
$$

Depth compatibility

$\mathsf{compatible}\{n;s;t\} := \forall i \in_L \{0..|t|-1\}\ .\ \mathsf{bdepth}\{\mathsf{nth}\{t;i\}\} = n + \mathsf{nth}\{s;i\}^5$

Figure 4.2: Definition of the type BTerm

The first equivalence holds because "$\mathbf{let}\, x = t_1\ \mathbf{in}\ t_2[x]$" is just the pretty-printed form of "$\mathbf{let}\,\{t_1; x.\, t_2[x]\}$." In this case, the term $\mathsf{T}\{\ulcorner\mathsf{let}\urcorner; [\ulcorner t_1\urcorner; \lambda_b x.\ulcorner t_2[x]\urcorner]\}$ makes sense only if the subterm $\lambda_b x.\ulcorner t_2[x]\urcorner$ contains exactly one more binder than the subterm $\ulcorner t_1\urcorner$. In other words, for any reflected term $\mathsf{B}\{n;\ulcorner\mathsf{let}\urcorner;[r_1;r_2]\}$ we require that the binding depths are well-formed.

$$
\begin{array}{rcl}
\mathsf{bdepth}\{r_1\} & = & n \\
\mathsf{bdepth}\{r_2\} & = & n+1
\end{array}
$$

We define the type BTerm of reflected terms in Figure 4.2. In this formulation, the function $f(t)$ produces a variable or bound term in de Bruijn form. The type T_i represents a quoted term tree with maximal subterm depth i. Finally, the desired type BTerm is the type of all term trees with finite depth; it contains all expressions of the forms:

- $\mathsf{V}\{i;j\}$ for all natural numbers i, j; and

[4] The dependent product $\Sigma x\colon A.\ B[x]$ is a space of *pairs* where A is a type, and $B[x]$ is a family of types indexed by $x \in A$. The elements of the product space are the pairs $\langle a,b\rangle$, where $a \in A$ and $b \in B[a]$.

[5] $\forall x \in_L l.\ T[x]$ represents that for each element x of the list l, $T[x]$ holds. Similarly, $\exists x \in_L l.\ T[x]$ defines the existential quantifier for lists.

- $\mathsf{B}\{n; op; ts\}$ for any natural number n, operator op, and list of terms ts that is compatible with $\mathsf{shape}\{op\}$ at depth n.

We derive the intended introduction rules for the BTerm type:

$$\frac{i \in \mathbb{N} \qquad j \in \mathbb{N}}{\mathsf{V}\{i; j\} \in \mathsf{BTerm}},$$

$$\frac{n \in \mathbb{N} \quad op \in \mathcal{O} \quad ts \in \mathsf{BTerm}\ \mathtt{list} \quad \mathsf{compatible}\{n; \mathsf{shape}\{op\}; ts\}}{\mathsf{B}\{n; op; ts\} \in \mathsf{BTerm}}.$$

Also, the structural induction principle is derived for the BTerm type. Namely, we show that to prove that some property $P[t]$ holds for any reflected term t, it is sufficient to prove

- (base case) P holds for all variables, that is, $P[\mathsf{V}\{i; j\}]$ holds for all natural numbers i and j;

- (induction step) $P[\mathsf{B}\{n; op; ts\}]$ is true for any natural number n, any operator op, and any list of reflected terms ts that is compatible with $\mathsf{shape}\{op\}$ at depth n, provided $P[t]$ is true for all elements t of the list ts.

Note that the type of "terms over n variables" (where $n = 0$ corresponds to closed terms) may be trivially defined using the BTerm type and the "subset" type constructor—$\{t\colon \mathsf{BTerm} \mid \mathsf{bdepth}\{t\} = n\}$.

4.7 Exotic Terms and Decidability

Further discussion is in order here. First, we should note that what we call the "HOAS" representation is in fact a very restricted form of HOAS, where the $\mathsf{Img}\{\cdot; \cdot\}$ type constructor is used to restrict the function spaces. For instance, consider the standard HOAS representation of a universal quantifier $\forall x\colon t_1.\ t_2[x]$. In traditional HOAS, this might be represented with a term constructor taking two arguments.

$$\forall : \text{Type} \rightarrow (\mathsf{Term} \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}$$

There are two potential problems with this representation. First, how should alpha-equality of terms be defined? Strictly speaking, the second argument to a $\forall$ term is a function. Since function equality is undecidable in general, if alpha-equality is to be computable, this cannot be an arbitrary function as we normally think of it! Second, if the function is unrestricted, the issue of *exotic terms* arises as we have mentioned in Chapter 2, where the function analyzes the structure of its argument inappropriately.

The approach taken in LF to the first problem is to syntactically restrict the function space by disallowing term destructors. In LF the λ terms are strongly normalizable and alpha-equality is therefore decidable. In LF, exotic terms are simply not expressible.

Our approach is similar, but also somewhat different. Our HOAS allows us to express exotic terms, however since the BTerm type is formalized as the image of the de Bruijn representation (and the formalization does not refer to function spaces), the exotic terms are left out and the BTerm type is restricted to only those HOAS-style expressions that are not exotic.

Theorem 4.1: *The type BTerm contains only those terms that have a de Bruijn representation.*

PROOF: The induction principle (Section 4.6), which was mechanically checked, establishes that the only terms in BTerm are $\mathsf{V}\{n; m\}$ and $\mathsf{B}\{n; o; s\}$. ∎

Corollary 4.1: *The type BTerm does not contain exotic terms.*

For example, consider the following exotic term, for some arbitrary fixed term t.

$$\lambda_b x.\, \lambda_b y.\ \textbf{if}\ x = t\ \textbf{then}\ x\ \textbf{else}\ y$$

This term is in canonical form, but it does not have type BTerm because it is not computationally equivalent to any $\mathsf{V}\{n; m\}$ or $\mathsf{B}\{n; o; s\}$.

Similarly, the mechanically-checked introduction rules for the BTerm type [42] establish that all the terms that have a de Bruijn representation belong to this type. For

those HOAS-style expressions that are members of the BTerm type, the corresponding de Bruijn representation is computable and therefore the alpha-equality is decidable (as it can be reduced to straightforward comparisons of de Bruijn representations).

4.8 A Small Example

To illustrate the utility of the de Bruijn representation, let us write a simple function to compute the free variables of a term (as might be used in closure conversion for example). That is, suppose we have a term t with binding depth n. In the HOAS representation, this term is computationally equivalent to a term of the form $t = \lambda_b x_1. \cdots. \lambda_b x_n. r$, and we wish to determine which of the variables $x_1, \ldots, x_n$ are free in r.

Now suppose we work directly with the HOAS representation. Calculating the free variables seems simple enough. First, we choose two distinct terms $r_1 \neq r_2$. Then, x_i occurs free in r *iff* the substitutions are not equal $r[r_1/x_i] \neq_\alpha r[r_2/x_i]$.[6] This leads to the following definition.

$$fv_{HOAS}(t) \quad = \quad \left\{ i \mid r \underbrace{@r_1@\cdots@r_1}_{i} \neq_\alpha r \underbrace{@r_1@\cdots@r_1}_{i-1} @r_2 \right\}$$

However, this definition is somewhat unsatisfying. It involves n substitutions and n equality tests. Most of all, it does not conform to the kind of naïve model we would expect, where the term is traversed in the usual way. In contrast, the de Bruijn destructor representation allows the more naïve algorithm.

In the following program, S is the set of free variables, t is the term being analyzed, and **fold** is the list fold function that calls fv_{dB} for each of the terms in the list l. The key part is the variable case, where the variable is added if its index i is less than n,

[6] We write $=_\alpha$ to emphasize that this is the expected alpha-equality. Since this is HOAS, this is just term equality $=$. As mentioned in the previous chapter, the decidability of term equality in general depends on the definition of the HOAS representation.

which indicates that the variable is free.

$$\begin{aligned} fv_{dB}(S,t) &= \textbf{match}\, t \,\textbf{with} \\ &\quad \mid \mathsf{V}\{i;j\} \rightarrow \textbf{if}\, i < n \,\textbf{then}\, S \cup \{i\} \,\textbf{else}\, S \\ &\quad \mid \mathsf{B}\{n;o;l\} \rightarrow \textbf{fold}\, fv_{dB}\ S\ l \end{aligned}$$

We claim that the de Bruijn-style computation fv_{dB} is clearer than the corresponding HOAS computation fv_{HOAS}. It has a more straightforward computational flavor, and is perhaps somewhat easier to reason about. Of course, there is no magic here. The two functions are extensionally equivalent. Furthermore, since the de Bruijn representation is defined in terms of HOAS, it merely codifies a style of computation that is reducible to operations on the HOAS.

4.9 Related Work

In Chapter 2 we have already discussed a number of approaches that we consider ourselves inheriting from. Here we would like to revisit some of them and mention a few other related efforts.

Our work has a lot in common with the HOAS implemented in Coq by Despeyroux and Hirschowitz [25]. In both cases, the more general space of terms (which include the exotic ones) is later restricted in a recursive manner. In both cases, the higher-order analogs of first-order de Bruijn operators are defined and used as a part of the "well-formedness" specification for the terms. Despeyroux and Hirschowitz use functions over infinite lists of variables to define open terms, which is similar to our vector bindings.

There are a number of significant differences as well. Our approach is sufficiently syntactical, which allows eliminating all exotic terms, even those that are extensionally equal to the well-formed ones, while the more semantic approach of [25, 24] has to accept such exotic terms (their solution to this problem is to consider an object term to be represented by the whole *equivalence class* of extensionally equal terms);

more generally while [25] states that "this problem of extensionality is recurrent all over our work," most of our lemmas establish identity and not just equality, thus avoiding most of the issues of extensional equality. In our implementation, the substitution on object terms is mapped directly to β-reduction, while Despeyroux, Felty, and Hirschowitz [24] have to define it recursively. In addition, we provide a *uniform* approach to both free and bound variables that naturally extends to variable-length "vector" bindings.

While our approach is quite different from the modal λ-calculus one [28, 26, 27], there are some similarities in the intuition behind it. Despeyroux, Pfenning, and Schürmann [28] says "Intuitively, we interpret $\Box B$ as the type of *closed* objects of type B. We can iterate or distinguish cases over closed objects, since all constructors are statically known and can be provided for." The intuition behind our approach is in part based on the canonical model of the NuPRL type theory [5, 6], where *each* type is mapped to an equivalence relations over the closed terms of that type.

Gordon and Melham [37] define the type of λ-terms as a quotient of the type of terms with concrete binding variables over α-equivalence. Michael Norrish [47] builds upon this work by replacing certain variable "freshness" requirements with variable "swapping." This approach has a number of attractive properties; however, we believe that the level of abstraction provided by the HOAS-style approaches makes the HOAS style more convenient and accessible.

Ambler, Crole, and Momigliano [8] have combined the HOAS with the induction principle using an approach which in some sense is opposite to ours. Namely, they define the HOAS operators on top of the de Bruijn definition of terms using *higher order pattern matching*. In a later work [9] they have described the notion of "*terms-in-infinite-context*" which is quite similar to our approach to vector binding. While our vector bindings presented in Section 4.3 are finite length, the exact same approach would work for the infinite-length "vectors" as well.

Chapter 5

The HOAS Representation Function

The representation function $\ulcorner\cdot\urcorner$ defines a *quotation* translation that produces a quoted form of its argument. To be clear, the quotation $\ulcorner\cdot\urcorner$ is terminology we use in this paper to describe the translation; it is not a part of any of the formal logics ($\mathbb{F}$, $\mathbb{M}$, $\mathbb{P}$). It is important that $\ulcorner\cdot\urcorner$ be total, where $\cdot$ can range over all the constructs of a formal logic, including rules, theorems, terms, etc.

The definition of the representation function is shown in Figure 5.1. The parts of interest are the quotations for concrete terms, sequents, and inference rules. The quoted representation of a concrete term, $\ulcorner op\{b_1; \cdots; b_n\}\urcorner$, produces a new term with a quoted name $\ulcorner op\urcorner$, and the quotation is carried out recursively on the subterms $\ulcorner b_1\urcorner; \cdots; \ulcorner b_n\urcorner$. The quotation of a sequent, $\ulcorner\Gamma \vdash t\urcorner$, is similar. For sequents, the "turnstile operator" is quoted, and the parts are quoted recursively.

The quotation of bound terms introduces a binder, written $\lambda_b x.\, t$, that represents each binding in quoted form. Note that the binding variable itself is unchanged; the variable is preserved as a binding, but each binding is explicitly coded as a λ_b.

As we can see, the representation function preserves the structure of the term, including variables, meta variables, and binding structure. This makes it possible to preserve a one-to-one correspondence between proofs in an original logic $\mathbb{P}$ and its reflected logic $\ulcorner\mathbb{P}\urcorner$.

Finally, the quotation of an inference rule, $\ulcorner t_1 \longrightarrow \cdots \longrightarrow t_n\urcorner$ becomes a state-

Terms			
$\ulcorner t \urcorner$:	$\ulcorner x \urcorner$	$=$	x
	$\ulcorner z[t_1; \cdots ; t_n] \urcorner$	$=$	$z[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$
	$\ulcorner \Gamma \vdash t \urcorner$	$=$	$\ulcorner \Gamma \urcorner \ulcorner \vdash \urcorner \ulcorner t \urcorner$
	$\ulcorner op\{b_1; \cdots ; b_n\} \urcorner$	$=$	$\mathsf{T}\{\ulcorner op \urcorner; [\ulcorner b_1 \urcorner; \cdots ; \ulcorner b_n \urcorner]\}$
$\ulcorner b \urcorner$:	$\ulcorner x_1, \cdots , x_n. t \urcorner$	$=$	$\lambda_b x_1. \cdots . \lambda_b x_n. \ulcorner t \urcorner$
Sequent contexts			
$\ulcorner \Gamma \urcorner$:	$\ulcorner h_1; \cdots ; h_n \urcorner$	$=$	$\ulcorner h_1 \urcorner; \cdots ; \ulcorner h_n \urcorner$
$\ulcorner h \urcorner$:	$\ulcorner x : t \urcorner$	$=$	$x : \ulcorner t \urcorner$
	$\ulcorner X[t_1; \cdots ; t_n] \urcorner$	$=$	$X[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$
Rules and logics			
$\ulcorner \mathbb{P} \urcorner$:	$\ulcorner R_1; \cdots ; R_n \urcorner$	$=$	$\ulcorner R_1 \urcorner; \cdots ; \ulcorner R_n \urcorner$
$\ulcorner R \urcorner$:	$\ulcorner t_1 \longrightarrow \cdots \longrightarrow t_n \urcorner$	$=$	$(Z \vdash \Box_{\mathbb{P}} \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box_{\mathbb{P}} \ulcorner t_n \urcorner)$

Figure 5.1: The definition of the representation function

ment about provability $(Z \vdash \Box_{\mathbb{P}} \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box_{\mathbb{P}} \ulcorner t_n \urcorner)$. The context variable Z is fresh, and each sequent $(Z \vdash \Box_{\mathbb{P}} \ulcorner t_i \urcorner)$ is a judgment in the metalogic $\mathbb{M}$ about provability in $\mathbb{P}$. As we will see in Chapter 7, $\Box_\star(\cdot)$ is a specific binary predicate defined in metalogic $\mathbb{M}$ using the standard definitional mechanisms provided by the logical framework $\mathbb{F}$.

Informally, the reflected form of the rule states that if each premise $t_1, \ldots, t_{n-1}$ is provable in logic $\mathbb{P}$, then so is t_n. A key goal is that the reflected rule $\ulcorner R \urcorner$ must be automatically derivable from the definition of $\mathbb{P}$. For clarity, when reasoning about a single logic we will normally omit the subscript $\Box_{\mathbb{P}}$ and simply write $\Box$.

Returning to our example, the quoted form of the substitution lemma (1.1) is as follows, where we write $s \ulcorner \in \urcorner t$ for $\ulcorner \mathsf{member} \urcorner\{s; t\}$.

$$\begin{aligned}
&Z \vdash \Box(X; x\colon z_3; Y \ulcorner \vdash \urcorner z_1[x] \ulcorner \in \urcorner z_2) \longrightarrow \\
&Z \vdash \Box(X; Y \ulcorner \vdash \urcorner z_0 \ulcorner \in \urcorner z_3) \longrightarrow \\
&Z \vdash \Box(X; Y \ulcorner \vdash \urcorner z_1[z_0] \ulcorner \in \urcorner z_2)
\end{aligned} \tag{5.1}$$

The operators have been quoted (in this case $\ulcorner \vdash \urcorner$ and $\ulcorner \in \urcorner$), and the theorem is

now a statement about provability expressed in the metalogic as $Z \vdash \Box \cdots$. Only the operator names have been changed, otherwise the structure, including variables and binding, has not changed.

For an example with binding, consider the rule for universal-introduction, shown below with the translated version. In this case, the binder x is translated to a meta binder with λ_b.

$$\ulcorner X; x\colon z_1 \vdash z_2[x] \longrightarrow X \vdash \forall x\colon z_1.z_2[x] \urcorner$$

$$=$$

$$Z \vdash \Box(X; x\colon z_1 \ulcorner\vdash\urcorner\ z_2[x]) \longrightarrow Z \vdash \Box(X \ulcorner\vdash\urcorner \ulcorner\forall\urcorner\{z_1; \lambda_b x.\, z_2[x]\})$$

In the next two chapters, we will show how to represent rules in detail and discuss about proof reflection.

Chapter 6

Sequent Representation

So far, we have delayed the discussion of the sequent representation. Other than formalizing the syntax, our goal is to mechanize the reasoning, which means we also need to represent inference rules and proofs, and for this purpose we must be able to represent sequents first. We want to reduce the sequent representation to the terms that already exist.

6.1 Sequent Context Induction

The issue is that arbitrary sequents, especially those that are used to define judgments in the programming language $\mathbb{P}$, usually include context variables. For example, consider a rule that might be used to define typing of a lambda abstraction.

$$\ulcorner \frac{X; x\colon A \vdash z[x] \in B}{X \vdash \lambda x\colon A.\, z[x] \in A \to B} \quad \text{lambda-intro}\urcorner$$

$$= \quad \frac{Z \vdash \Box(X; x\colon A \ulcorner\vdash\urcorner\, z[x] \ulcorner\in\urcorner\, B)}{Z \vdash \Box(X \ulcorner\vdash\urcorner\, (\ulcorner\lambda\urcorner\, x\colon A.\, z[x])\, \ulcorner\in\urcorner\, (A \ulcorner\to\urcorner B))} \quad \ulcorner\text{lambda-intro}\urcorner$$

Here, X and Z are sequent context variables, A, B, and $z[x]$ are second-order variables, and x is a first-order variable. In general, sequent context variables are bindings, sequent contexts are not terms, and they cannot be modeled directly in the object logic. How can we reduce a sequent with a context variable to canonical form?

Since the framework metalogic we are using (the MetaPRL metalogic) does not

include context quantifiers, one option is to add them and use them in the proof-checking predicate. However, this is undesirable in part because the framework's metalogic would become extremely expressive and powerful, but also because the extension is perilous and difficult to get right.

Instead, Hickey extends the framework's metalogic with a weak theory of sequent context induction called *teleportation*. The central logical property is that contexts are finite and inductively defined. Note that this represents a strengthening of the metalogic by effectively including Peano arithmetic.

Teleportation. The concept behind teleportation is fairly simple. Since contexts are inductively defined, contexts can be "migrated" from one point in a judgment to another. Scoping must be preserved, including context variable scoping, but beyond that the migration locations are unconstrained. The teleportation rule simply states that in order to be able to teleport the whole context at once, we only need to be able to migrate it one hypothesis at a time.

To formalize this more precisely, we introduce the notion of teleportation contexts, written $R[\![\Gamma]\!]$, which represents a term or a rule with exactly one occurrence of the context Γ. We will use the symbol ϵ to denote the empty context. These definitions are for presentation purposes; they are not part of the metalogic. Teleportation is specified using a pair of nested teleportation contexts, which we will write as $F[\![\cdot; G[\![\cdot]\!]]\!]$. Here $F[\![\Gamma; G[\![\Delta]\!]]\!]$ must be a rule that has exactly one occurrence of each of the Γ, Δ and G; in addition G must be in the scope of Γ.

The simplest teleportation rule hoists the context from G to F.

$$\frac{\begin{array}{ll}\text{(base)} & F[\![\epsilon; G[\![X]\!]]\!] \\ \text{(step)} & F[\![X; G[\![x\colon t; Y[x]]\!]]\!] \longrightarrow F[\![X; x\colon t; G[\![Y[x]]\!]]\!]\end{array}}{F[\![X; G[\![\epsilon]\!]]\!]}$$

The teleportation rules are added as new primitive rules in our system. The conservativity theorem for sequent schema [45], which states that the language of meta variables is a conservative extension of the meta-theory, can be extended to

include teleportation rules. The central observation here is that for any particular finite concrete context Γ, any proof using the teleportation rules can be transformed into a proof without teleportation by posing a finite sequence of lemmas, one for each of the intermediate steps.

A simple example. The best way to understand teleportation is through examples. For a fairly natural one, consider the problem of context exchange. Suppose a logic has the structural exchange rule for hypotheses.

$$\frac{X_1; y\colon t_2; x\colon t_1; X_2[x;y] \vdash t_3[x;y]}{X_1; x\colon t_1; y\colon t_2; X_2[x;y] \vdash t_3[x;y]}$$

We wish to derive the context-exchange property.

$$\frac{X_1; Z_2; Z_1; X_2 \vdash t}{X_1; Z_1; Z_2; X_2 \vdash t}$$

The proof in this case can be posed as a nested induction. To begin, we propose to migrate Z_1 right. To apply the teleportation rule, for each occurrence of a context variable Γ on which we wish to perform induction, we specify a teleportation target, written as $\bullet_\Gamma$. In the example of context exchange, the annotated rule is written as follows.

$$\frac{X_1; Z_2; Z_1; \bullet_{Z_1}; X_2 \vdash t}{X_1; Z_1; Z_2; \bullet_{Z_1}; X_2 \vdash t}$$

In the goal clause, the target $\bullet_{Z_1}$ specifies that Z_1 is to be shifted right across Z_2; in the premises, Z_1 is shifted in place. The induction produces a base case

$$\frac{X_1; Z_2; Z_1; X_2 \vdash t}{X_1; Z_2; Z_1; X_2 \vdash t},$$

which is trivial, as well as a step case

$$\frac{X_1; Z_2; Z_1; X_2 \vdash t \qquad X_1; S; Z_2; x\colon A; T[x]; X_2[x] \vdash t[x]}{X_1; S; x\colon A; Z_2; T[x]; X_2[x] \vdash t[x]}.$$

The step case follows from the hypothesis migration lemma below.

$$\frac{X_1; Z; x\colon A; X_2[x] \vdash t[x]}{X_1; x\colon A; Z; X_2[x] \vdash t[x]}$$

We prove it by migrating Z past the hypothesis $x\colon A$. Here is the annotated rule for the lemma.

$$\frac{X_1; \bullet_Z; Z; x\colon A; X_2[x] \vdash t[x]}{X_1; \bullet_Z; x\colon A; Z; X_2[x] \vdash t[x]}$$

The base case is trivial. The step case,

$$\frac{X_1; Z; x\colon A; X_2[x] \vdash t[x] \qquad X_1; S; y\colon B; x\colon A; T[y]; X_2[y;x] \vdash t[y;x]}{X_1; S; x\colon A; y\colon B; T[y]; X_2[y;x] \vdash t[y;x]},$$

follows directly from the hypothesis exchange rule.

6.2 Computation on Sequent Terms

The sequent induction scheme also introduces a sequent induction combinator for computation over a sequent context. The $\texttt{sequent_ind}\{x, y.\mathit{step}[x;y]; s\}$ performs computation over a sequent term s. The reduction rules for sequent computation are as follows.

$$\texttt{sequent_ind}\{x, y.\, \mathit{step}[x;y]; (\vdash t)\} \;\rightarrow\; t \qquad \text{(base)}$$

$$\texttt{sequent_ind}\{x, y.\, \mathit{step}[x;y]; (z\colon t_1; X[z] \vdash t_2[z])\} \;\rightarrow$$
$$\mathit{step}[t_1; \lambda z.\, \texttt{sequent_ind}\{x, y.\, \mathit{step}[x;y]; (X[z] \vdash t_2[z])\}] \qquad \text{(left-reduction)}$$

$$\texttt{sequent_ind}\{x, y.\, \mathit{step}[x;y]; (X; z\colon t_1 \vdash t_2[z])\} \;\rightarrow$$
$$\texttt{sequent_ind}\{x, y.\, \mathit{step}[x;y]; (X \vdash \mathit{step}[t_1; \lambda z.\, t_2[z]])\} \qquad \text{(right-reduction)}$$

To illustrate, suppose we wish to develop a "vector" universal quantifier. That is, a sequent with the following definition, given that the logic has a "scalar" quantifier $\forall x\colon t_1.\ t_2[x]$.

$$x_1\colon t_1; \cdots; x_n\colon t_n \vdash_\forall t_{n+1} \;:=\; \forall x_1\colon t_1, \cdots, x_n\colon t_n.\ t_{n+1}$$

The definition is implemented in terms of sequent induction.

$$\Gamma \vdash_\forall t \quad := \quad \texttt{sequent_ind}\{x, y.\, \forall z\colon x.\ (y\ z); (\Gamma \vdash t)\}$$

We get the following reductions.

$$\begin{aligned} \vdash_\forall t \quad &\rightarrow \quad t \\ x\colon t_1; X[x] \vdash_\forall t_2[x] \quad &\rightarrow \quad \forall x\colon t_1.\ (X[x] \vdash_\forall t_2[x]) \\ X; x\colon t_1 \vdash_\forall t_2[x] \quad &\rightarrow \quad X \vdash_\forall (\forall x\colon t_1.\ t_2[x]) \end{aligned}$$

The simple introduction rule can be derived directly.

$$\frac{Z; x\colon t_1 \vdash (X[x] \vdash_\forall t_2[x])}{Z \vdash (x\colon t_1; X[x] \vdash_\forall t_2[x])} \qquad \text{(vall-intro-single)}$$

Furthermore, a general introduction rule is also derivable using the teleportation rules.

$$\frac{Z; X \vdash t}{Z \vdash (X \vdash_\forall t)} \qquad \text{(vall-intro)}$$

Using similar methods, it is possible to define a logic of vector operators, quantifiers, and a vector lambda calculus.

Note that in these rules, the variable X is a context variable, and the rules are valid for any instance of X.

6.3 Computing Canonical Sequent Representations

With this new tool in hand, let us return to the topic of reflection, where the issue was that we need to reduce the reflected sequents (with context variables) to canonical representations.

At this point, the plan is conceptually easy. There are two parts. First, we develop a canonical representation of concrete sequents without context variables. For the second part, we define a (formal) function that computes the canonical representation

from the noncanonical form that includes context variables.

The first part is an issue of coding, where the goal is to define a representation that preserves the structure of concrete sequents. We choose the following representation, where $\ulcorner\lambda_H\urcorner x\colon t_1.\, t_2$ is a quoted term that represents a hypothesis, its binding, and the rest of the sequent; and $\ulcorner\texttt{concl}\urcorner\{t\}$ represents the conclusion of the sequent.

$$
\begin{array}{rcl}
x_1\colon t_1; \cdots x_n\colon t_n \ulcorner\vdash\urcorner t_{n+1} & := & \ulcorner\lambda_H\urcorner x_1\colon t_1. \cdots .\ulcorner\lambda_H\urcorner x_n\colon t_n.\ulcorner\texttt{concl}\urcorner\{t_{n+1}\} \\
\textit{where} & & \\
\ulcorner\lambda_H\urcorner x_1\colon t_1.\, t_2 & := & \mathsf{T}\{\ulcorner\mathsf{hlambda}\urcorner; [t_1; \lambda_b x.\, t_2]\} \\
\ulcorner\texttt{concl}\urcorner\{t\} & := & \mathsf{T}\{\ulcorner\texttt{concl}\urcorner; [t]\}
\end{array}
$$

For the second part, we define a function over arbitrary quoted sequents using `sequent_ind` that computes the canonical representation from its noncanonical form. This function, written $\vdash_B$, is defined as follows.

$$
X \vdash_B t \;\; := \;\; \texttt{sequent_ind}\{x, y. \ulcorner\lambda_H\urcorner z\colon x.\, (y\ z); (X \vdash \ulcorner\texttt{concl}\urcorner\{t\})\}
$$

Using this definition, the representation of a reflected rule is computed using $\vdash_B$ in place of $\ulcorner\vdash\urcorner$. The original reflected form of a rule

$$
R \;\; = \;\; (\Gamma_1 \vdash t_1) \longrightarrow \cdots \longrightarrow (\Gamma_n \vdash t_n)
$$

is

$$
\ulcorner R\urcorner \;\; = \;\; (Z \vdash \Box(\ulcorner\Gamma_1\urcorner \ulcorner\vdash\urcorner \ulcorner t_1\urcorner)) \longrightarrow \cdots \longrightarrow (Z \vdash \Box(\ulcorner\Gamma_n\urcorner \ulcorner\vdash\urcorner \ulcorner t_n\urcorner)).
$$

Using the noncanonical forms, the new representation is

$$
\ulcorner R\urcorner \;\; = \;\; (Z \vdash \Box(\ulcorner\Gamma_1\urcorner \vdash_B \ulcorner t_1\urcorner)) \longrightarrow \cdots \longrightarrow (Z \vdash \Box(\ulcorner\Gamma_n\urcorner \vdash_B \ulcorner t_n\urcorner)).
$$

For example, the reflection form of the lambda-typing rule is as follows.

$$\frac{Z \vdash \Box(X; x\colon A \vdash_B z[x] \ulcorner\in\urcorner B)}{Z \vdash \Box(X \vdash_B (\ulcorner\lambda\urcorner\, x\colon A.\, z[x]) \ulcorner\in\urcorner (A \ulcorner\rightarrow\urcorner B))} \quad \ulcorner\text{lambda-intro}\urcorner$$

The right-hand side will be proved by first reducing the $\vdash_B$ sequents to canonical form. Note that contexts and context variables are not terms, and so it remains impossible to quantify over them directly. However, the reduced form of a noncanonical $\vdash_B$ sequent with context variables does contain sequent subterms with context variables. With teleportation it is possible to show that these embedded terms are well-defined.

Note that occurrences of $\vdash_B$ represent actual sequents, and a sequent $X \vdash_B t$ has the type BTerm when $X \vdash t$ is a proper sequent. Since the reflection translation $\ulcorner\cdot\urcorner$ preserves variables, including context variables and bindings, the reflected form of the rule has the same binding structure as the original rule in the programming language $\mathbb{P}$. In fact, the reflected rule is in a form that can be used directly as a proof step in the reflected logic $\ulcorner\mathbb{P}\urcorner$, and there is a one-to-one correspondence from proof steps in $\mathbb{P}$ to proof steps in $\ulcorner\mathbb{P}\urcorner$. Furthermore, as we discuss in the next chapter, the reflected rules (such as $\ulcorner$lambda-intro$\urcorner$) are automatically derivable in the metalogic $\mathbb{M}$.

Chapter 7

Proof Reflection

As mentioned earlier, reflection contains two main parts: representing the syntax, and mechanizing the reasoning. Heretofore we have postponed the treatment of the provability predicate $\Box_{\mathbb{P}}\, t$, in which t is the quotation of a formula provable in logic $\mathbb{P}$. In fact, a concept we have glossed over is the truth of reflected rules—if logic $\mathbb{P}$ defines a rule R, the truth of $\ulcorner R \urcorner$ does not follow axiomatically. It must be proved, and the proof relies on the definition of the provability predicate. For the most part, this definition is a *coding* problem where the objective is to construct a formal data structure that corresponds exactly to the legal derivations/proof trees. To define provability properly, we take the following steps.

- First, for each rule $R \in \mathbb{P}$, we define a *proof checking* predicate that specifies whether a proof step is a valid application of rule R.
- Next, we define the (legal) *derivations* to be the proof trees where each proof step in the tree is validated by some rule $R \in \mathbb{P}$.
- A formula t is *provable* in logic $\mathbb{P}$ if, and only if, there is a derivation with root t.

The usual properties hold: proof checking is decidable, provability is not decidable in general.

7.1 Proof Checking

A logic $\mathbb{P}$ is an ordered list of inference rules $R_1, \ldots, R_m$. A proof is a tree of inferences, and it is legal only if each proof step corresponds to an inference using some rule R_i. A *proof step* is a node in the proof tree that corresponds to a concrete inference $t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_n$. We call the terms $t_1, \ldots, t_{n-1}$ the *premises*, and the term t_n the *goal*.

In general, a rule R defines a *schema*, where each second-order meta variable stands for a term, and each context meta variable stands for a context. A concrete proof step is a valid inference of a rule R *iff* for each second-order meta variable in R there is an actual term, and for each context meta variable in R there is an actual context, such that the concrete inference is an instance of the rule.

Let us state this more formally. The *arity* of a meta variable is the number of its arguments, so a variable $z[t_1; \cdots; t_n]$ has arity n. Let $\texttt{BTerm}\{i\}$ be the type of quoted terms of arity i, corresponding to the space of substitution functions $\texttt{BTerm}^i \rightarrow \texttt{BTerm}$. Similarly, let $\texttt{Context}\{i\}$ be the type of contexts of arity i (the contexts correspond to lists of quoted terms).

Consider a rule R with free second-order variables $x_1^{j_1}, \cdots, x_n^{j_n}$ and free context variables $X_1^{i_1}, \cdots, X_m^{i_m}$, where the superscripts i_k and j_k indicate the arities of the variables.[1] Then a concrete inference r is a valid instance of rule R *iff* the following holds.

$$\begin{aligned} &\exists x_1^{j_1} : \texttt{BTerm}\{j_1\}, \cdots, x_n^{j_n} : \texttt{BTerm}\{j_n\}. \\ &\exists X_1^{i_1} : \texttt{Context}\{i_1\}, \cdots, X_m^{i_m} : \texttt{Context}\{i_m\}. \\ &\qquad r = R \in \texttt{ProofStep} \end{aligned} \tag{7.1}$$

That is, the concrete inference r is equal to an instance of rule R. The type $\texttt{ProofStep}$ is the type of proof steps, defined as $\texttt{BTerm list} \times \texttt{BTerm}$, containing the pairs $\langle \textit{premises}, \textit{goal} \rangle$. In our MetaPRL implementation, each premise/goal is a reflected sequent; as stated in Chapter 6, it has type BTerm.

For the purposes of proof checking, the existential witnesses are assembled into

[1] In a setting where context variables are treated as binders, the variable arities are expressions that depend on the lengths $|X_k|$.

a proof witness term, and passed as explicit arguments to the checker. A proof witness is defined to be an element of the `Witness` type, which in turn is defined as `BTerm list` $\times$ `Context list`.

Returning to the example of the substitution lemma (5.1), the corresponding proof checker is defined as follows, where r is the concrete proof step to be checked.

$$\texttt{checks}\{\texttt{subst_lemma}; r; \langle [z_1; z_2; z_3; z_0], [X; Y] \rangle\} = \\ r = \left\langle \begin{array}{c} [(X; x\colon z_3; Y \vdash_B z_1[x] \ulcorner\in\urcorner z_2); (X; Y \vdash_B z_0 \ulcorner\in\urcorner z_3)], \\ (X; Y \vdash_B z_1[z_0] \ulcorner\in\urcorner z_2) \end{array} \right\rangle \in \texttt{ProofStep} \tag{7.2}$$

Note that we use $\vdash_B$ instead of $\ulcorner\vdash\urcorner$ in the `subst_lemma` above; as mentioned in Chapter 6, this is the canonical representation of sequents.

In general, the "rule checker" predicate $\texttt{checks}\{R; r; w\}$ takes three arguments, where R is a rule, $r \in$ `ProofStep` is a concrete inference, and $w \in$ `Witness` is the witness for the rule instantiation. Given a logic $\mathbb{P}$ with rules $R_1, \ldots, R_n$, a proof step is valid *iff* it is an instance of one of the rules in the logic.

$$\texttt{checks}_{\mathbb{P}}\{r; w\} \quad := \quad \exists R \in_{\mathbb{P}} [R_1; \cdots; R_n].\ \texttt{checks}\{R; r; w\}$$

Since proof step equality is decidable, and each logic has a finite number of rules, the $\texttt{checks}_{\mathbb{P}}\{r; w\}$ predicate is decidable as well.

7.2 Derivations and Provability

Now that we have defined proof step checking, the next part is to define the valid derivations, or proof trees. For this chapter, we will assume we are referring to a specific logic $\mathbb{P}$, using the proof-checking predicate $\texttt{checks}_{\mathbb{P}}$ for logic $\mathbb{P}$.

The type D of all derivations is defined inductively in the usual way. A derivation is a finite tree of proof steps; the derivation is valid *iff* each proof step is valid. The derivations are defined inductively over the length of the derivation. There are no derivations of length 0, and a derivation of length i has a goal term and a list of

derivation premises, in the following form, where D_i is the type of derivations of length at most i, and each premise d_j^{i-1} has length at most $i-1$.

$$D_i \ni \frac{d_1^{i-1} \quad \cdots \quad d_n^{i-1}}{t}$$

Given a derivation d, we define $\texttt{goal}\{d\}$ to be the goal term of d. Extending this to lists, we define $\texttt{goal}\{[d_1; \cdots; d_n]\}$ to be the list of goal terms for derivations $d_1, \ldots, d_n$.

The type D_i of derivations of length at most i can be defined formally as follows, where `void` is the empty type. The type D of all derivations is the union of all derivations of finite length.

$$\begin{aligned} D_0 &:= \texttt{void} \\ D_{i+1} &:= \Sigma \mathit{premises}\colon D_i\ \texttt{list}.\ \Sigma \mathit{goal_term}\colon \texttt{BTerm}.\ \Sigma \mathit{witness}\colon \texttt{Witness}. \\ &\qquad \texttt{checks}_{\mathbb{P}}\{\langle \texttt{goal}\{\mathit{premises}\}, \mathit{goal_term}\rangle\,;\mathit{witness}\} \\ D &:= \bigcup_{i\in\mathbb{N}} D_i \end{aligned} \tag{7.3}$$

This definition also allows us to prove an induction principle, which will form the basis for proof induction.

$$\begin{aligned} &\forall P.\ ((\forall \mathit{premises}\colon D\ \texttt{list}.\ \forall g\colon \texttt{BTerm}.\ \forall w\colon \texttt{Witness}. \\ &\qquad (\texttt{checks}_{\mathbb{P}}\,\{\langle \texttt{goal}\{\mathit{premises}\}, g\rangle\,; w\} \\ &\qquad \Rightarrow \forall p \in_L \mathit{premises}.\ P[p] \\ &\qquad \Rightarrow P\,[\langle \mathit{premises}, g, w\rangle])) \\ &\quad \Rightarrow \forall d\colon D.\ P[d]) \end{aligned}$$

At this point, the definition of the *provability* predicate $\Box\, t$ is straightforward. A quoted term t is provable *iff* there is a derivation where t is the goal term.

$$\Box\, t \;:=\; \exists d\colon D.\ (\texttt{goal}\{d\} = t \in \texttt{BTerm})$$

7.3 Proof Reflection and Automation

One important consequence of structure preservation is that proofs can be reflected as well.

Lemma 7.1: *The refinement relation for rules [45] is preserved when reflected. Specifically, let R be a rule, and σ be a refinement of R (by second-order and context substitution), producing R'. The following diagram is commutative for an appropriate refinement σ'.*

$$\begin{array}{ccc} R & \xrightarrow{\sigma} & R' \\ \downarrow & & \downarrow \\ \ulcorner R \urcorner & \xrightarrow{\sigma'} & \ulcorner R' \urcorner \end{array}$$

PROOF: Define σ' to be the reflected form of σ. Namely, for each substitution in σ, define a corresponding substitution in σ', where each term operator is reflected. Because the $\ulcorner \cdot \urcorner$ mapping preserves binding and free variable structure, σ' will be a valid refinement function [45]. By construction, $\sigma'(\ulcorner R \urcorner)$ is $\ulcorner R' \urcorner$. ∎

Lemma 7.2: *If $\mathbb{P}$ defines an axiom R, the reflected form of R, $\ulcorner R \urcorner$, is a theorem in the metalogic $\mathbb{M}$.*

PROOF: Suppose R is an axiom with premises $t_1, \ldots, t_{n-1}$, and goal t_n.

$$R \;=\; t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_n$$

As mentioned in chapters 5 and 6, its reflected form is as below.

$$\ulcorner R \urcorner \;=\; (Z \vdash \Box(\ulcorner \Gamma_1 \urcorner \vdash_B \ulcorner t_1 \urcorner)) \longrightarrow \cdots \longrightarrow (Z \vdash \Box(\ulcorner \Gamma_n \urcorner \vdash_B \ulcorner t_n \urcorner))$$

The definition of the provability predicate says that there is a derivation with root $\ulcorner \Gamma_i \urcorner \vdash_B \ulcorner t_i \urcorner$ for all $i = 1..n$, which means there is a finite tree of valid proof steps. Each valid proof step is an instance of one of the inference rules in the logic. As

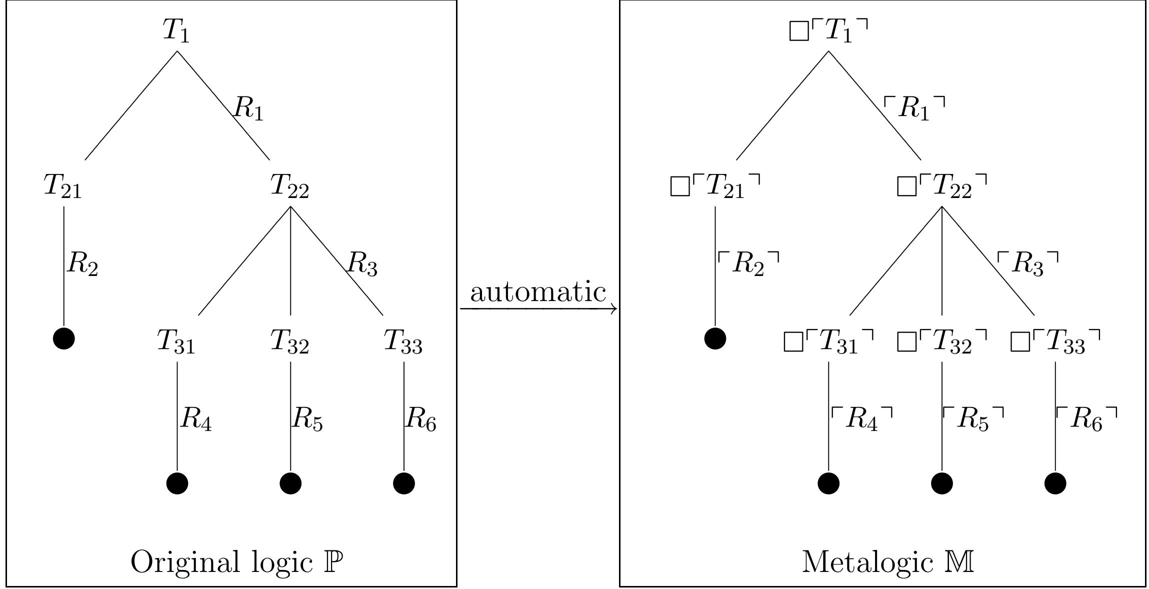

Figure 7.1: An example proof tree and its reflected form: $\Box\ulcorner T_1 \urcorner$ is proved automatically

pointed out by Nogin and Hickey [45], this implies that $\Gamma_i \vdash t_i$ is provable in $\mathbb{P}$. This holds for all $i = 1..n$, so $\ulcorner R \urcorner$ is true in $\mathbb{M}$. ■

Theorem 7.1: *When reflected, the derived rules in* $\mathbb{P}$ *are automatically derivable in the metalogic* $\mathbb{M}$*.*

PROOF: Consider a proof in the original logic $\mathbb{P}$ of some theorem T_1. In a foundational prover, the proof is expressed as a tree of inferences that can be linearized to a finite sequence of rule applications $R_1, R_2, \ldots, R_m$, which means after applying rule R_1, T_1 produces subgoals $T_{21}, T_{22}, \ldots, T_{2j}$ and the subgoals are then proved sequentially by rule applications $R_2, \ldots, R_m$. See Figure 7.1 for an example proof tree of T_1, where after applying R_1 we get T_{21} and T_{22}. T_{21} is proved by applying R_2, and T_{22} by $R_3, \ldots, R_6$.

By definition, the reflected rule $\ulcorner T_1 \urcorner$ has the same structure as T_1. Lemma 7.2 states that $\ulcorner R_1 \urcorner$ is a derived theorem in $\mathbb{M}$. If we apply $\ulcorner R_1 \urcorner$ on $\ulcorner T_1 \urcorner$, by Lemma 7.1, we will get a series of subgoals $\ulcorner T_{21} \urcorner$, $\ulcorner T_{22} \urcorner$, $\ldots$, $\ulcorner T_{2j} \urcorner$, which are the structure-preserved reflected forms of $T_{21}, T_{22}, \ldots, T_{2j}$ (see Figure 7.1). The proof of this theorem follows immediately by induction on the depth of the proof tree. The proof

of $\ulcorner T_1 \urcorner$ is a one-to-one map of the original theorem, using reflected justifications in place of the original. That is, the reflected proof is $\ulcorner R_1 \urcorner, \ulcorner R_2 \urcorner, \ldots, \ulcorner R_m \urcorner$. ∎

While this might seem quite straightforward, the important property here is that the prover internals do not need to be reflected. It is not necessary to formalize the inference mechanics of the theorem prover, because the original mechanism works without change in the reflected theory.

Proof automation is similar. Again, in a foundational prover,[2] each run of heuristic or decision procedure is justified by a sequence of inferences $R_1, R_2, \ldots$ The existing automation may be used for reasoning in the reflected logic, *provided* that rule selection for reflected proofs uses the reflected rules rather than the original ones.

7.4 Proof Induction

Up to this point, we have presented a structure-preserving representation function, a mechanism for formalizing reflected logics, and a procedure for deriving reflected provability rules. This system is already powerful enough to express and prove metaproperties over reflected systems. However, it remains impractical. There is a piece missing, namely induction on the provability predicate $\Box\, t$.

What exactly is the induction principle for provability? Suppose we wish to prove a theorem of the form $\Box\, x \Rightarrow P[x]$, where x is a variable, and P is a predicate on quoted terms. Since x is provable, that means that there is a derivation with root x, and we can apply induction on the length of the derivation.

Now, for illustration, assume that the logic $\mathbb{P}$ contains three rules, $\mathbb{P} = [t_{11};\ t_{21} \longrightarrow$

[2]It is not clear to us whether a similar mechanism might work for non-foundational provers (those with "trusted" decision procedures).

$t_{22};\ t_{31} \longrightarrow t_{32} \longrightarrow t_{33}]$. Then the induction form has the following shape.

$$\frac{\begin{array}{l}\text{(rule sketch)}\\ \Gamma; \Box\, t_{11} \vdash P[t_{11}]\\ \Gamma; \Box\, t_{21}; \Box\, t_{22}; P[t_{21}] \vdash P[t_{22}]\\ \Gamma; \Box\, t_{31}; \Box\, t_{32}; \Box\, t_{33}; P[t_{31}]; P[t_{32}] \vdash P[t_{33}]\end{array}}{\Gamma; \Box\, x \vdash P[x]}$$

However, this rule is not quite right. The issue is that the terms t_{ij} will, in general, contain meta variables, and the meta variables must be separately universally quantified for each induction case.

As mentioned in Chapter 6, explicit quantification of meta variables is not expressible in our metalogic $\mathbb{F}_{\mathsf{MetaPRL}}$. However, here it is acceptable to use object-quantifiers provided by the metatheory $\mathbb{M}_{\mathsf{CTT}}$. There is no appreciable effect on proof automation as long as the first-order form is compatible with the automatically generated reflected rules. The correct form of the rule explicitly quantifies over the meta variables, reusing the mechanism for generating the proof-checking rules. For the current example, we introduce explicit quantifiers. In this case we write $t_{ij}[\vec{X}]$ to represent a term that may contain any of the variable $\vec{X}$ but is otherwise free of context variables.

$$\frac{\begin{array}{l}\Gamma; \vec{X}\colon \texttt{Context}; \Box\, t_{11}[\vec{X}] \vdash P[t_{11}[\vec{X}]]\\ \Gamma; \vec{X}\colon \texttt{Context}; \Box\, t_{21}[\vec{X}]; \Box\, t_{22}[\vec{X}]; P[t_{21}[\vec{X}]] \vdash P[t_{22}[\vec{X}]]\\ \Gamma; \vec{X}\colon \texttt{Context}; \Box\, t_{31}[\vec{X}]; \Box\, t_{32}[\vec{X}]; \Box\, t_{33}[\vec{X}]; P[t_{31}[\vec{X}]]; P[t_{32}[\vec{X}]] \vdash P[t_{33}[\vec{X}]]\end{array}}{\Gamma; \Box\, x \vdash P[x]}$$

In our implementation, we generate a variant of this rule that allows for induction over terms, not just variables. This is done by introducing a "shared" term u that establishes a connection between the provable term t and the predicate P. The actual theorem has the form $\Gamma; u\colon t_1; \Box\, t_2[u] \vdash P[t_3[u]]$, where u is the shared part. The new form is derivable from the previous case for provability on variables, and is stated

below.

$$
\begin{aligned}
\forall P.\ \forall t_2.\ ((\forall v\colon t_1.\ \forall \mathit{premises}\colon \mathsf{BTerm}\ \mathtt{list}.\ \forall w\colon \mathtt{Witness}.\\
&(\mathtt{checks}_{\mathbb{P}}\{(\mathit{premises}, t_2[v]); w\}\\
&\Rightarrow \forall p \in_L \mathit{premises}.\ \Box\, p\\
&\Rightarrow \forall p \in_L \mathit{premises}.\ \forall w\colon t_1.\ (t_2[w] = p \Rightarrow P[w])\\
&\Rightarrow P[v]))\\
\Rightarrow \forall u\colon t_1.\ (\Box\, t_2[u] \Rightarrow P[u]))
\end{aligned}
$$

This mechanism establishes the principle of proof induction. The principle of structural induction is reducible to proof induction by specifying the syntax of a language as a logic of type checking, as will be discussed in Section 8.4.

7.5 Preliminary Discussion

At this point, all the pieces are in place. We have defined

1. the type BTerm of reflected terms,
2. the reflection function $\ulcorner\cdot\urcorner$,
3. the provability predicate $\Box\, r$ (for $r \in \mathsf{BTerm}$), and
4. we have derived a proof induction principle.

The following are the key steps when using this methodology.

- To reason about a specific programming language $\mathbb{P}$:
    1. Formalize the syntax and axioms/typing rules $R_1, \ldots, R_n$ as a primitive logic $\mathbb{P}$ in the framework $\mathbb{F}$.
    2. State any relevant theorems $R'_1, \ldots, R'_m$ of interest in $\mathbb{P}$. Prove those that do not require metatheoretical reasoning.

3. Instruct the framework to reflect the logic $\mathbb{P}$. The framework will create a subtheory $\ulcorner\mathbb{P}\urcorner$ of the metalogic $\mathbb{M}$ having the following parts,
   - a definition of $\Box_{\mathbb{P}}$,
   - inference rules $\ulcorner R_1 \urcorner, \ldots, \ulcorner R_n \urcorner$,
   - theorem statements $\ulcorner R'_1 \urcorner, \ldots, \ulcorner R'_m \urcorner$,
   - a proof induction principle for $\Box_{\mathbb{P}}\, r$, where r is a reflected formula in $\ulcorner\mathbb{P}\urcorner$.

- The following are proved automatically by the framework:
  - Each inference rule $\ulcorner R_i \urcorner$ is derived as a theorem of $\mathbb{M}$.
  - The induction principle is derived as theorem of $\mathbb{M}$.
  - Any proofs of the theorems $R'_1, \ldots, R'_m$ are reflected to form the proofs of $\ulcorner R'_1 \urcorner, \ldots, \ulcorner R'_m \urcorner$.[3]
- At no point are new axioms added to $\mathbb{F}$ and $\mathbb{M}$; the subtheory $\ulcorner\mathbb{P}\urcorner$ is fully derived.

As we have mentioned, the *structure* of terms and rules is preserved by the transformation $\ulcorner\mathbb{P}\urcorner$. The practical consequence is that reasoning in the original logic $\mathbb{P}$ maps directly to reasoning in the reflected logic $\ulcorner\mathbb{P}\urcorner$; any approach to constructing a proof of the theorem R'_i in $\mathbb{P}$ equally applies to proving $\ulcorner R'_i \urcorner$ in $\ulcorner\mathbb{P}\urcorner$.

If a theorem R'_i is not proved in $\mathbb{P}$, the framework will state the corresponding $\ulcorner R'_i \urcorner$ as a theorem to be proved in the reflected logic $\ulcorner\mathbb{P}\urcorner$ by the user. The reflected logic $\ulcorner\mathbb{P}\urcorner$ affords an induction principle, and in addition permits the use of quantifiers from $\mathbb{M}$ that are not available in $\mathbb{P}$. These two properties together enable the statement (and proof) of metatheoretical principles that could not be proved in $\mathbb{P}$ itself. Such metatheoretical principles can then be used in the proof of $\ulcorner R'_i \urcorner$.

It should be noted that this is only a fragment of reflection—we do not add a reflection rule $\Box \ulcorner t \urcorner \Rightarrow t$. Any metatheorem that is proved in $\ulcorner\mathbb{P}\urcorner$ remains a fact solely

[3] This step is mechanical, but in our implementation it is not yet complete.

of $\ulcorner \mathbb{P} \urcorner$; the original theory $\mathbb{P}$ remains open and unconstrained. While a reflection rule could be added as a new axiom to $\mathbb{F}$, we believe it is unnecessary and that the consequent strengthening of the framework logic $\mathbb{F}$ is undesirable and dangerous.

Chapter 8

An Example: $F_{<:}$

To illustrate reflection in practice, we have formalized an initial part of the programming language/type system $F_{<:}$, as defined by the POPLmark challenge. This work (as well as the reflection mechanism that has been discussed), is implemented in the MetaPRL logical framework. In this system, the framework logic $\mathbb{F}$ is the logic of sequent-schema [45], which is essentially a logic of second-order Horn formulas; the metalogic $\mathbb{M}$ is Computational Type Theory [42], a variant of Martin-Löf type theory.

Following the methodology described in the previous chapter, the first step is to define $\mathbb{P}_{F_{<:}}$ as a primitive logic in $\mathbb{F}$. This immediately brings us to the issue of syntax and representation. In most textbook accounts, the syntax of a logic receives little discussion; the syntax is frequently described with a context-free grammar, and then dismissed so that the discussion can proceed to more interesting issues. In a formal account, the process cannot be dismissed, and in fact is quite important, as it provides the basis for structural induction.

8.1 Defining the Syntax of $F_{<:}$

The syntax of $F_{<:}$ is shown in Figure 8.1 in the usual textbook form as well as in the concrete syntax for defining the terms of a logic in MetaPRL. The textbook form is standard. The MetaPRL form introduces each term as a declaration. The "typeclass" declarations, such as declare typeclass Ty, define classes of syntax (this should not be confused with the word typeclass as it is used in Haskell or other languages; the

Syntax in textbook form				**Raw syntax for $\mathbb{P}_{F_{<:}}$ (in MetaPRL)**
t	$::=$		Types	declare typeclass Ty
	$\mid$	Top	the maximal type	declare Top: Ty
	$\mid$	$t_1 \rightarrow t_2$	function space	declare TyFun{t1: Ty; t2: Ty}: Ty
	$\mid$	$\forall X <: t_1 .\, t_2[X]$	universal type	declare TyAll{t1: Ty; X: Ty. t2[X]: Ty}: Ty
e	$::=$		Expresssions	declare typeclass Exp
	$\mid$	$\lambda x : t .\, e[x]$	functions	declare lambda{t: Ty; x: Exp. e[x]: Exp}: Exp
	$\mid$	$e_1\ e_2$	applications	declare apply{e1: Exp; e2: Exp}: Exp
	$\mid$	$\Lambda X <: t .\, e[X]$	type abstraction	declare Lambda{t: Ty; X: Ty. e[X]: Exp}: Exp
	$\mid$	$e[\![t]\!]$	type application	declare Apply{e: Exp; t: Ty}: Exp
P	$::=$		Propositions	declare typeclass Prop
	$\mid$	$e \in t$	type membership	declare member{e: Exp; t: Ty}: Prop
	$\mid$	$t_1 <: t_2$	subtyping	declare subtype{t1: Ty; t2: Ty}: Prop
J	$::=$	$\Gamma \vdash P$	Judgments	declare typeclass Judgment
Γ	$::=$	$h_1; \cdots ; h_n$	sequent contexts	declare sequent {
h	$::=$	$x : t$	value assumption	$\mid$ Exp: Ty
	$\mid$	$X <: t$	type assumption	$\mid$ Ty: TyBound
				>- Prop}: Judgment

Figure 8.1: The syntax of $F_{<:}$, in textbook form and in the concrete syntax for $\mathbb{P}$

meaning here is simply that the term Ty denotes a syntactic collection of terms). The term declarations are in a restricted form of HOAS. For example, the declaration TyFun{t1: Ty; t2: Ty}: Ty specifies that the term TyFun represents a type (Ty) with two subterms, both of which must be types. The declaration TyAll{t1: Ty; X: Ty. t2[X]: Ty}: Ty is similar, but it introduces a binding X, and the subterm t2[X] must be a type if X is a type.

Sequent declarations introduce a new issue. In $F_{<:}$, sequent hypotheses have two forms, one that introduces an expression binder, and another that introduces a type binder. For example, in the $F_{<:}$ sequent $X <: \mathrm{Top}; x \colon X \vdash x \in X$, the variable X has type Ty, and x has type Exp. The corresponding MetaPRL declaration includes the two cases. As a technicality, the MetaPRL framework logic $\mathbb{F}$ includes only a single binding form, so we introduce a new term BoundedBy{t: Ty}: TyBound. The $F_{<:}$ hypothesis $X <: \mathrm{Top}$ takes the form X: BoundedBy{Top}.

8.2 Reflected Syntax

When reflected, a formula $t \in \mathbb{P}_{F_{<:}}$ becomes a formula $\ulcorner t \urcorner \in \mathsf{BTerm}$. However, we should be careful here, because the transformation $\ulcorner t \urcorner$ is defined over many more terms than those in $\mathbb{P}_{F_{<:}}$. For example, we have $\ulcorner \mathsf{lambda}\{\mathrm{Top}; x.\mathrm{Top}\} \urcorner \in \mathsf{BTerm}$ even though $\mathsf{lambda}\{\mathrm{Top}; x.\mathrm{Top}\}$ is not a well-formed formula of $F_{<:}$. The type BTerm includes all the reflected formulas; what is needed is to define subtypes $\mathsf{BTerm}_{\mathcal{C}}$ where $\mathcal{C}$ is a syntactic class (for example $\mathsf{BTerm}_{\mathsf{Ty}}$).

However, naïve definitions of $\mathsf{BTerm}_{\mathcal{C}}$ quickly run aground. For example, suppose we wish to define the type $\mathsf{BTerm}_{\mathsf{Ty}}$ using a type-theoretic form of set comprehension

$$\mathsf{BTerm}_{\mathsf{Ty}} \quad := \quad \{t \colon \mathsf{BTerm} \mid \mathsf{isty}\{t\}\},$$

where $\mathsf{isty} : \mathsf{BTerm} \to \mathbb{B}$ is a predicate and $\mathsf{isty}\{t\}$ is true *iff* t is a term that represents

a type in $F_{<:}$. The pseudocode is as follows.

$$
\begin{aligned}
\mathsf{isty}\{t\} \quad = \quad & \textbf{match}\, t \,\textbf{with} \\
& \mid \mathrm{Top} \rightarrow \textbf{true} \\
& \mid \mathsf{TyFun}\{t_1; t_2\} \rightarrow \mathsf{isty}\{t_1\} \wedge \mathsf{isty}\{t_2\} \\
& \mid \mathsf{TyAll}\{t_1; x.t_2[x]\} \rightarrow \mathsf{isty}\{t_1\} \wedge \forall x \colon \mathsf{BTerm}_{\mathsf{Ty}}.\ \mathsf{isty}\{t_2[x]\}
\end{aligned}
$$

However, the final clause for TyAll includes a quantification over $\mathsf{BTerm}_{\mathsf{Ty}}$, so the type definition must at least be recursive. Furthermore the occurrence is negative; this approach is unlikely to work. In fact, this kind of negativity is just an instance of the general problem of naming in formal metatheory.

However, there is a simple and easy way out, which is to define the syntax as a logic. That is, each term declaration is viewed as a syntactical judgment. We introduce a syntactical judgment $\vdash_M$ (we call it a "metatype" judgment) that defines syntactic well-formedness. Some examples are shown in Figure 8.2. The syntax rules are expressed in prereflected form. For the most part, these rules are entirely straightforward. Each declaration defines a corresponding type-checking rule in the logic $\vdash_M$. Sequent declarations result in two or more type-checking rules, where there is one rule for each of the kinds of hypotheses in the sequent.

If we wish, we can now give a formal definition to the type $\mathsf{BTerm}_{\mathsf{Ty}}$ as follows.

$$
\mathsf{BTerm}_{\mathsf{Ty}} \quad := \quad \{t \colon \mathsf{BTerm} \mid \Box\,(\vdash_M t \in_M \mathsf{Ty})\}
$$

Note however that this type includes only the closed terms t. For this reason, in our implementation, we use the predicate $\Box\,(\Gamma_M \vdash_M t \in_M \mathsf{Ty})$ directly.

8.3 The $F_{<:}$ Logic

We now proceed to define the logical (as opposed to syntax) rules of the $F_{<:}$ type system. The rules themselves are standard, we show a small fragment in Figure 8.3.

Syntax judgments		**Rules**
$J_M ::= \Gamma_M \vdash_M t \in_M \mathcal{C}$	syntax judgment	
$\Gamma_M ::= x_1 : \mathcal{C}_1; \cdots ; x_n : \mathcal{C}_n$	syntax contexts	$\dfrac{}{\Gamma_M; x : \mathcal{C}; \Delta_M[x] \vdash_M x \in_M \mathcal{C}}$

Syntax declaration	**Syntax judgments**
TyAll{t1: Ty; X: Ty; t2[X]: Ty}: Ty	$\dfrac{\Gamma_M \vdash_M \mathsf{t1} \in_M \mathsf{Ty} \quad \Gamma_M; X : \mathsf{Ty} \vdash_M \mathsf{t2}[X] \in_M \mathsf{Ty}}{\Gamma_M \vdash_M \mathsf{TyAll}\{\mathsf{t1{:}\ Ty;\ X{:}\ Ty;\ t2[x]{:}\ Ty}\} \in_M \mathsf{Ty}}$
lambda{t: Ty; x: Exp; e[x]: Exp}: Exp	$\dfrac{\Gamma_M \vdash_M \mathsf{t} \in_M \mathsf{Ty} \quad \Gamma_M; x : \mathsf{Exp} \vdash_M \mathsf{e}[x] \in_M \mathsf{Exp}}{\Gamma_M \vdash_M \mathsf{lambda}\{\mathsf{t{:}\ Ty;\ x{:}\ Exp;\ e[x]{:}\ Exp}\} \in_M \mathsf{Exp}}$
sequent{Exp: Ty | Ty: TyBound ⊢ Prop}: Judgment	$\left\{ \begin{array}{l} \dfrac{\Gamma_M \vdash_M P \in_M \mathsf{Prop}}{\Gamma_M \vdash_M (\vdash P) \in_M \mathsf{Judgment}} \\ \\ \dfrac{\Gamma_M \vdash_M t \in_M \mathsf{Ty} \quad \Gamma_M; x : \mathsf{Exp} \vdash_M (Y[x] \vdash P[x]) \in_M \mathsf{Judgment}}{\Gamma_M \vdash_M (x : t; Y[x] \vdash P[x]) \in_M \mathsf{Judgment}} \\ \\ \dfrac{\Gamma_M \vdash_M t \in_M \mathsf{TyBound} \quad \Gamma_M; x : \mathsf{Ty} \vdash_M (Y[x] \vdash P[x]) \in_M \mathsf{Judgment}}{\Gamma_M \vdash_M (x : t; Y[x] \vdash P[x]) \in_M \mathsf{Judgment}} \end{array} \right.$

Figure 8.2: A fragment of the syntax judgments

Here, the letters e, s, S, and T stand for second-order meta variables. As before, hypothesis $x : \mathsf{BoundedBy}\{t\}$ represents the form x <: t.

The remainder of the logic now proceeds much as it did for the syntax. When reflected, the rules define a logic of provability in $\ulcorner\mathbb{P}\urcorner$. The reflected sublogic $\ulcorner\mathbb{P}\urcorner$ enables reasoning both by structural induction (based on the syntax rules) and proof induction (based on the logical rules), and the user can draw on either principle as need be.

Textbook notation		Concrete representation in MetaPRL
$\dfrac{x \in \mathsf{dom}\{\Gamma\}}{\Gamma \vdash x <: x}$	sa_tvar	**prim** sa_tvar : $X; x : \mathsf{BoundedBy}\{t\}; Y[x] \vdash \mathsf{subtype}\{x; x\}$
$\dfrac{\Gamma; x <: S \vdash e \in T}{\Gamma \vdash \Lambda x <: S.\ e \in \forall x <: S.\ T}$	t_tabs	**prim** t_tabs : $\dfrac{X; x : \mathsf{BoundedBy}\{S\} \vdash \mathsf{member}\{e[x]; T[x]\}}{X \vdash \mathsf{member}\{\mathsf{Lambda}\{S; x.e[x]\}; \mathsf{TyAll}\{S; x.T[x]\}\}}$

Figure 8.3: A fragment of logical rules from $F_{<:}$

One point to note is that some metaproperties are expressible in the original logic, and it is usually desirable to do so in these cases. For example, the property of reflexivity of subtyping is expressible (though not provable) in the original logic, as a theorem of the following form.

$$\Gamma \vdash t \texttt{<:}\, t$$

When reflected in the context of the $F_{\texttt{<:}}$ syntax, we require that the theorem be a valid judgment of the logic $\mathbb{F}_{F_{\texttt{<:}}}$, and the theorem takes the form

$$(\Box \vdash_M (\Gamma \ulcorner\vdash\urcorner\, t \texttt{<:}\, t) \in_M \ulcorner\texttt{Judgment}\urcorner) \Rightarrow (\Box\,(\Gamma \ulcorner\vdash\urcorner\, t \ulcorner\texttt{<:}\urcorner t)),$$

where quotations have been added in the appropriate places by the reflection mechanism.

8.4 Structural Induction

Since the syntax is expressed as a type-checking logic, structural induction reduces to proof induction. That is, a type-checking judgment has the form $\Box\,(\Gamma_M \ulcorner\vdash_M\urcorner \ulcorner t\urcorner \ulcorner\in_M\urcorner \ulcorner\mathcal{C}\urcorner)$, for some term t and syntax class $\mathcal{C}$. Induction on the provability yields the possible cases for the term t.

To illustrate, suppose we wish to prove reflexivity of subtyping. The proof proceeds as follows. For clarity, we have omitted occurrences of the quotation symbol $\ulcorner\cdot\urcorner$; it should be understood that every non-variable term is quoted.

- $\Box\,(\vdash_M (\Gamma \vdash x \texttt{<:}\, x) \in_M \textsf{Judgment}) \Rightarrow \Box\,(\Gamma \vdash x \texttt{<:}\, x)$

  This is the goal to be proved. We forward chain, using the assumption as follows.

  - $\Box\,(\vdash_M (\Gamma \vdash x \texttt{<:}\, x) \in_M \textsf{Judgment})$

    This is the assumption. Proof induction on the Judgment judgment leads to the following fact, where the $\lfloor\Gamma\rfloor$ denotes the reduction of the context Γ to a syntactic context with hypotheses of the form $x\colon \mathcal{C}$.

- $\square(\lfloor\Gamma\rfloor \vdash_M (x <: x) \in_M \mathsf{Prop})$

  From the declaration for $\mathsf{subtype}$, we can infer from $(x <: x)$ that x is a type by proof induction.

- $\square(\lfloor\Gamma\rfloor \vdash_M x \in_M \mathsf{Ty})$

- We now perform structural induction on $\square(\lfloor\Gamma\rfloor \vdash_M x \in_M \mathsf{Ty})$, to obtain the following subgoals. By assumption $(\lfloor\Gamma\rfloor \vdash_M x \in_M \mathsf{Ty})$ has a proof, and that proof must end with one of the Ty rules, so x is either a variable or one of Top, TyFun, or TyAll.

  - $\square(\Gamma; x <: T; \Delta[x] \vdash x <: x)$

  - $\square(\Gamma \vdash \mathrm{Top} <: \mathrm{Top})$

  - $\square(\lfloor\Gamma\rfloor \vdash_M t_1 \in_M \mathsf{Ty}) \Rightarrow \square(\Gamma \vdash t_1 <: t_1) \Rightarrow$
    $\square(\lfloor\Gamma\rfloor \vdash_M t_2 \in_M \mathsf{Ty}) \Rightarrow \square(\Gamma \vdash t_2 <: t_2) \Rightarrow$
    $\square(\Gamma \vdash (t_1 \rightarrow t_2) <: (t_1 \rightarrow t_2))$

  - $\square(\lfloor\Gamma\rfloor \vdash_M t_1 \in_M \mathsf{Ty}) \Rightarrow \square(\Gamma \vdash t_1 <: t_1) \Rightarrow$
    $\square(\lfloor\Gamma\rfloor; x\colon \mathsf{Ty} \vdash_M t_2[x] \in_M \mathsf{Ty}) \Rightarrow$
    $\square(\Gamma; x <: \mathrm{Top} \vdash t_2[x] <: t_2[x]) \Rightarrow$
    $\square(\Gamma \vdash (\forall x <: t_1.\ t_2[x]) <: (\forall x <: t_1.\ t_2[x]))$

  Each of these subgoals is now provable from the rules in $\ulcorner F_{<:} \urcorner$.

In other cases, the metatheorems are not expressible in the original logic. For example, the proof of transitivity of subtyping requires the proof of two properties by simultaneous induction.

1. If $\Gamma \vdash t_1 <: t_2$ and $\Gamma \vdash t_2 <: t_3$, then $\Gamma \vdash t_1 <: t_3$.

2. If $\Gamma, x <: t_2, \Delta[x] \vdash t_4[x] <: t_5[x]$ and $\Gamma \vdash t_6 <: t_2$, then $\Gamma, x <: t_6, \Delta[x] \vdash t_4[x] <: t_5[x]$.

Since the theorem is the simultaneous proof of two rules, it cannot be expressed in $F_{<:}$. In the reflected system it simply becomes a conjunction.

$$\begin{pmatrix} (\Box\,\Gamma \vdash t_1 <: t_2) \Rightarrow \\ (\Box\,\Gamma \vdash t_2 <: t_3) \Rightarrow \\ (\Box\,\Gamma \vdash t_1 <: t_3) \end{pmatrix} \wedge \begin{pmatrix} (\Box\,\Gamma, x <: t_2, \Delta[x] \vdash t_4 <: t_5) \Rightarrow \\ (\Box\,\Gamma \vdash t_6 <: t_2) \Rightarrow \\ (\Box\,\Gamma, x <: t_6, \Delta[x] \vdash t_4 <: t_5) \end{pmatrix}$$

The proof proceeds in the usual way, by structural induction on t_2 [12, Appendix A].

Chapter 9

Discussion and Future Work

The goal of this work is to develop a *practical* theory of logical reflection. While reflection can be defined over many different representations, we believe that a practical approach requires reusing existing automated methods, and doing so requires that the structure of a theory be preserved, including variables, meta variables, and bindings. We presented a structure-preserving representation of syntax and logical terms, using a hybrid approach of a combined HOAS/de Bruijn representation. Based on a weak form of sequent context induction called *teleportation*, we developed a mechanism that allows reasoning and computation over terms that include sequent context variables. This led to a formalization of proofs, proof-checkers, and derivations, together with automated generation of reflected rules and induction forms in the reflected theory.

In some ways, the result seems simple. When a logic is reflected, its presentation changes only slightly, and the existing reasoning methods and proof procedures continue to work. The difference is, of course, that reasoning about metaproperties of the logic becomes possible.

It was important to us that the development of the theory of reflection be accompanied by its implementation. This makes it more useful of course, but an additional reason is that the theory of reflection is rife with paradoxes, and it is easy to fall into false thinking. While we have tried to simplify the account in this thesis, the actual formalization was demanding. In Appendix A we show an architectural overview of the theories we implemented in MetaPRL and in Appendix B we provide a selected listing of the most important implemented theories. The complete theory listing is

online and freely available [42].

Part of the power of our approach comes from the availability of first-order forms (including the de Bruijn representation) that are used to build the formal foundations. The goal, however, is to provide users with a convenient HOAS-based high-level interface.

This formal framework of reflection provides a general and uniform way to define, view, and manipulate programming languages, and can be used to develop programming language metatheories. Currently, we are using reflection to develop an account of the $F_{<:}$ type theory as defined by the POPLmark challenge. Using our environment for the task of formalization is straightforward; the routine proofs are fully automated. In addition, for simple proofs, the first-order forms are entirely hidden. One challenge, however, is to simplify the use of proof induction, which, as discussed in Section 7.4, relies on the first-order representation, including the metalogic quantifiers. The use of metalogical quantifiers differs from the rest of the account, where implicit framework quantifiers are used to quantify over second-order and context variables. The result is that there is a mismatch in representation when induction is performed, and proofs by induction are overly complicated. We believe that this can be solved by reformulating the induction rules, and we are currently working on a solution.

Another area of immediate interest is the issue of evaluation contexts, which are needed for reasoning about preservation and progress. We believe that the syntactical definitions used frequently in the literature provide a natural way to define evaluation contexts, and we are pursuing this line of investigation.

We believe that our results may be generalized to other provers and frameworks. The non-standard properties of the logical framework that we rely upon are the following.

1. Programs may be expressed without first giving them a type; in addition, programs may have more than one type.

2. Computation defines a congruence; any two programs that are computationally

(beta) equivalent can be interchanged in any formal context.

3. For reasoning about sequents, a weak induction principle is needed on sequent context variables.

4. A *function image* type [46].

Bibliography

[1] E. Aaron and S. Allen. Justifying calculational logic by a conventional metalinguistic semantics. Technical Report TR99-1771, Cornell University, Ithaca, New York, Sept. 1999.

[2] W. Aitken and R. L. Constable. Reflecting on NuPRL : Lessons 1–4. Technical report, Cornell University, Computer Science Department, Ithaca, NY, 1992.

[3] W. Aitken, R. L. Constable, and J. Underwood. Metalogical Frameworks II: Using reflected decision procedures. *Journal of Automated Reasoning*, 22(2):171–221, 1993.

[4] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The NuPRL open logical environment. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[5] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, June 1987.

[6] S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[7] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 95–197. IEEE Computer Society Press, June 1990.

[8] S. Ambler, R. L. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, pages 13–30, London, UK, 2002. Springer-Verlag.

[9] S. J. Ambler, R. L. Crole, and A. Momigliano. A definitional approach to primitive recursion over higher order abstract syntax. In *Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–11. ACM Press, 2003.

[10] S. Artemov. On explicit reflection in theorem proving and formal verification. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 267–281, Berlin, July 7–10 1999. Trento, Italy.

[11] S. Artemov. Evidence-based common knowledge. Technical Report TR-2004018, CUNY Ph.D. Program in Computer Science Technical Reports, Nov. 2004.

[12] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. Available from http://www.cis.upenn.edu/group/proj/plclub/mmm/, 2005.

[13] E. Barzilay. Quotation and reflection in NuPRL and Scheme. Technical Report TR2001-1832, Cornell University, Ithaca, New York, Jan. 2001.

[14] E. Barzilay. *Implementing Reflection in NuPRL*. PhD thesis, Cornell University, 2006.

[15] E. Barzilay and S. Allen. Reflecting higher-order abstract syntax in NuPRL. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[16] E. Barzilay, S. Allen, and R. Constable. Practical reflection in NuPRL. Short paper presented at 18th Annual IEEE Symposium on Logic in Computer Science, June 22–25, Ottawa, Canada, 2003.

[17] J. Cheney. Towards a general theory of names, binding and scope. In *Proceedings of the 2005 workshop on Mechanized reasoning about languages with variable binding*, pages 33–40. ACM Press, 2005.

[18] R. L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, NATO Series F*, pages 65–100. Springer, Berlin, 1994.

[19] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

[20] K. Crary, R. Harper, and M. Ashley-Rollman. POPLmark submission. Available at `http://www.cis.upenn.edu/group/proj/plclub/mmm/submissions.shtml`, 2005.

[21] L. Crus-Filipe and F. Weidijk. Hierarchical reflection. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 2004.

[22] J. Davis and D. Huttenlocher. Shared annotations for cooperative learning. In *Proceedings of the ACM Conference on Computer Supported Cooperative Learning*, Sept. 1995.

[23] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972. This also appeared in

the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.

[24] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculus and its Applications (TLCA'95)*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, Apr. 1995. Also appears as INRIA research report RR-2556.

[25] J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in Coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, 1994. Also appears as INRIA research report RR-2292.

[26] J. Despeyroux and P. Leleu. A modal lambda calculus with iteration and case constructs. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs: International Workshop, TYPES '98, Kloster Irsee, Germany, March 1998*, volume 1657 of *Lecture Notes in Computer Science*, pages 47–61, 1999.

[27] J. Despeyroux and P. Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11(4):555–572, 2001.

[28] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher–order abstract syntax. In R. Hindley, editor, *Proceedings of the International Conference on Typed Lambda Calculus and its Applications (TLCA'97)*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163, Nancy, France, Apr. 1997. Springer-Verlag. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[29] A. Ehrenfeucht and J. Mycielski. Abbreviating proofs by adding new axioms. *Bulletin of the American Mathematical Society*, 77:366–367, 1971.

[30] M. Fairbairn. POPLmark submission. Available at http://www.cis.upenn.edu/group/proj/plclub/mmm/submissions.shtml, 2005.

[31] S. Feferman, J. W. Dawson, S. C. Kleene, G. H. Moore, R. M. Solovay, and J. van Heijenoort, editors. *Kurt Gödel Collected Works*, volume 1. Oxford University Press, Oxford, Clarendon Press, New York, 1986.

[32] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.

[33] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 162–178. Springer-Verlag, 2000.

[34] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, Cambridge, Mass., 1989.

[35] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English version in [58].

[36] K. Gödel. Über die Länge von beweisen. *Ergebnisse eines mathematischen Kolloquiums*, 7:23–24, 1936. English translation in [31], pages 397–399.

[37] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.

[38] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Oxford Univerity, Computing Laboratory, 2003.

[39] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993. A revised and expanded version of the 1987 paper.

[40] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-53, SRI International, Cambridge Computer Science Research Centre, Millers Yard, Cambridge, UK, Feb. 1995.

[41] J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. MetaPRL — A modular logical environment. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[42] J. J. Hickey, B. Aydemir, Y. Bryukhov, A. Kopylov, A. Nogin, and X. Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.

[43] J. J. Hickey, A. Nogin, A. Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[44] A. Mostowski. *Sentences undecidable in formalized arithmetic: An exposition of the theory of Kurt Gödel*. Amsterdam: North-Holland, 1952.

[45] A. Nogin and J. Hickey. Sequent schema for derived rules. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.

[46] A. Nogin and A. Kopylov. Formalizing type operations using the "Image" type constructor. Submitted to Workshop on Logic, Language, Information and Computation (WoLLIC), 2006.

[47] M. Norrish. Recursive function definition for types with binders. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 241–256. Springer-Verlag, 2004.

[48] R. Parikh. Existence and feasibility in arithmetic. *The Journal of Symbolic Logic*, 36:494–508, 1971.

[49] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.

[50] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1994.

[51] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 313–322, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[52] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[53] A. M. Pitts and M. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[54] G. Plotkin. An illative theory of relations. In R. Cooper, K. Mukai, and J. Perry, editors, *Situation Theory and Its Applications, Volume 1*, number 22 in CSLI Lecture Notes, pages 133–146. Centre for the Study of Language and Information, 1990.

[55] President's Information Technology Advisory Committee. Report to the President—information technology research: Investing in our future, Feb. 1999.

[56] President's Information Technology Advisory Committee. Report to the President—Cyber security: A crisis of prioritization, Feb. 2005.

[57] B. Smith. Reflection and semantics in Lisp. *Principles of Programming Languages*, pages 23–35, 1984.

[58] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, MA, 1967.

[59] J. Vouillon. POPLmark submission. Available at `http://www.cis.upenn.edu/group/proj/plclub/mmm/submissions.shtml`, 2005.

Appendix A

Architecture and Summary of the Implementation in MetaPRL

As mentioned earlier, this work is fully implemented in the MetaPRL theorem prover. Here we show the architecture of the formal implementation of our reflection framework in MetaPRL (Figure A.1). In the parentheses next to each module name, we give orderly the number of rules and rewrites in that module, the total number of proof tactics we used to prove them, as well as the total number of actual proof steps in the system.

Corresponding to the presentation in this thesis, we divide the architecture into three parts: syntax representation, sequent representation, and proof reflection, each of which consists of a number of theories/modules that define the corresponding terms and operations, and include axioms and/or fully derived theorems.

The five modules surrounded by the broken line is the "core" of our theory, which defines a hybrid HOAS/de Bruijn representation of syntax as mentioned in Chapter 4, and provides a foundation for our whole reflection work.

To help with better understanding of our work, we include a selected listing of the modules we implemented in Appendix B, as indicated by the blue shades in Figure A.1. For a complete theory listing, please refer to the nightly updated online resource [42].

Syntax representation

itt_hoas_base (6,6,64)
itt_hoas_vbind (8,13,139)
itt_hoas_vector (29,55,4185)
itt_hoas_operator (29,25,447)
itt_hoas_debruijn (39,105,15283)
itt_hoas_destterm (10,8,283)
itt_hoas_lof (47,159,15892)
itt_hoas_bterm (97,283,14390)
itt_hoas_lof_vec (14,30,427)
itt_hoas_relax (54,343,33312)
itt_hoas_normalize (16,16,8695)
itt_hoas_util (2,10,270)
itt_hoas_bterm_wf (20,148,33917)

Sequent representation

itt_hoas_sequent (63,198,5916)
itt_hoas_sequent_bterm (70,408,14639)
itt_hoas_sequent_term (52,118,1945)
itt_hoas_sequent_term_wf (30,184,10070)
itt_hoas_sequent_normalize (20,44,1127)

Proof reflection

itt_hoas_proof (56,253,10304)
itt_hoas_meta_types (0,0,0)
itt_hoas_sequent_proof_step (6,24,993)
itt_hoas_sequent_proof (15,49,1961)
itt_hoas_proof_ind (7,41,1708)
itt_hoas_sequent_elim (1,1,15)

Figure A.1: Architecture of the reflection formalization in MetaPRL

Appendix B

Partial Listing of Relevant MetaPRL Theories

This appendix is a printout of some of the relevant MetaPRL theories and was generated automatically by the MetaPRL system. The complete implementation is available online [43, 42]. The MetaPRL notation used in this appendix is partially explained in [45, 42, 43]. Rules and rewrites marked with a "*[n_1, n_2]" are derived (n_1 and n_2 provide a measure of the proof size) and the "![$\cdots$]" marker means that the rule/rewrite is an axiom. Most of the operators are presented in their pretty-printed forms.

B.1 Itt_hoas_base module

The `Itt_hoas_base` module defines the basic operations of the Higher Order Abstract Syntax (HOAS).

B.1.1 Parents

Extends `Base_theory`

Extends `Itt_dfun`

Extends `Itt_union`

Extends `Itt_prod`

Extends `Itt_list2`

B.1.2 Terms

The expression $B\ x.\ t[x]$ represents a "bound" term ("bterm") with a potentially free variable x. In order for it to be well-formed, t must be a "substitution function".

The $T(op;\ subterms)$ expression represents a term with the operator *op* and subterms *subterms*. In order for it to be well-formed, the length of *subterms* must equal the arity of *op* and each subterm must have the "binding depth" (i.e. the number of outer binds) equal to the corresponding number in the shape of *op* (remember, the shape of an operator is a list of natural numbers and the length of the list is the operator's arity).

The expression $bt@t$ represents the result of substituting t for the first binding in bt.

Finally, the `weak_dest_bterm` operator allows testing whether a term is a `bind` or a `mk_term` and to get the *op* and *subterms* in the latter case.

define `unfold_bind` :

```
Itt_hoas_base!bind{x. 't<|!!|>['x]}
```

(displayed as "$B\ x.\ t[x]$"**)** $\longleftrightarrow$

$inl\ (\lambda x.t[x])$

define iform `bind_list` :

`Itt_hoas_base!bind_list{'terms<|!!|>}`

(displayed as "$bind_list\{terms\}$"**)** $\longleftrightarrow$

$B\ terms$

define `unfold_mk_term` :

`Itt_hoas_base!mk_term{'op<|!!|>; 'subterms<|!!|>}`

(displayed as "$T(op;\ subterms)$"**)** $\longleftrightarrow$

$inr\ (op, subterms)$

declare `Itt_hoas_base!illegal_subst`

(displayed as "$illegal_subst$"**)**

define `unfold_subst` :

`Itt_hoas_base!subst{'bt<|!!|>; 't<|!!|>}`

(displayed as "$bt@t$"**)** $\longleftrightarrow$

match bt **with**

$inl\ f -> \ f\ t$

$|\ inr\ opt -> \ illegal_subst$

define iform `subst_list` :

`Itt_hoas_base!subst_list{'terms<|!!|>; 'v<|!!|>}`

(displayed as "$subst_list\{terms;\ v\}$"**)** $\longleftrightarrow$

$terms@v$

define `unfold_wdt` :

`Itt_hoas_base!weak_dest_bterm`

`{'bt<|!!|>;`

`'bind_case<|!!|>;`

`op, sbt. 'mkterm_case<|!!|>['op; 'sbt]}`

(displayed as

"**match** bt **with**

$B\ _ -> \ bind_case$

$|\ T(op;\ sbt) -> \ mkterm_case[op;\ sbt]$"**)** $\longleftrightarrow$

match bt **with**

$inl\ f\ -> \ bind_case$

$|\ inr\ opt\ -> $ **let**

(op, sbt) = opt

in

$mkterm_case[op;\ sbt]$

B.1.3 Rewrites

$*[1, 12]$ **rewrite** `reduce_subst` $\{|$ **public** reduce $|\}$:

$(B\ x.\ bt[x])@t \longleftrightarrow bt[t]$

$*[1, 10]$ **rewrite** `reduce_wdt_bind` $\{|$ **public** reduce $|\}$:

match $B\ x.\ t[x]$ **with**

$B\ _ -> \ bind_case$

$|\ T(op;\ sbt) -> \ mkterm_case[op;\ sbt]$

$\longleftrightarrow$

$bind_case$

$*[1, 12]$ **rewrite** `reduce_wdt_mk_term` $\{|$ **public** reduce $|\}$:

match $T(op;\ subterms)$ **with**

$B\ _ -> \ bind_case$

$|\ T(o;\ sbt) -> \ mkterm_case[o;\ sbt]$

$\longleftrightarrow$

$mkterm_case[op;\ subterms]$

On occasion, it is also useful to work with subterm lists directly, without the operator. Define another constructor $mk_terms\{l\}$ that represents a list of terms l.

define `unfold_mk_terms` :

`Itt_hoas_base!mk_terms{'l<|!!|>}`

(displayed as "$mk_terms\{l\}$"**)** $\longleftrightarrow$

inr l

define `unfold_weak_dest_terms :`

```
Itt_hoas_base!weak_dest_terms
    {'bt<|!!|>;
     'bind_case<|!!|>;
     l. 'terms_case<|!!|>['l]}
```

(displayed as

"$weak_dest_terms\{bt;\ bind_case;\ l.\ terms_case[l]\}$") $\longleftrightarrow$

match bt **with**

$inl\ x\ -> \ bind_case$

$|\ inr\ y\ -> \ terms_case[y]$

$*[1,10]$ **rewrite** `reduce_weak_dest_terms_bind`

{| **public** reduce |} :

$weak_dest_terms$

$\{B\ x.\ t[x];$

$bind_case;$

$terms.\ terms_case[terms]\}$

$\longleftrightarrow$

$bind_case$

$*[1,10]$ **rewrite** `reduce_weak_dest_terms_mk_term`

{| **public** reduce |} :

$weak_dest_terms$

$\{T(op;\ subterms);$

$bind_case;$

$terms.\ terms_case[terms]\}$

$\longleftrightarrow$

$terms_case[(op, subterms)]$

$*[1,10]$ **rewrite** `reduce_weak_dest_terms_mk_terms`

{| **public** reduce |} :

$weak_dest_terms$

$\{mk_terms\{l\};$

$bind_case;$

$terms.\ terms_case[terms]\}$

$\longleftrightarrow$

$terms_case[l]$

B.2 Itt_hoas_vector module

The `Itt_hoas_vector` module defines the "vector bindings" extensions for the basic ITT HOAS.

B.2.1 Parents

Extends `Itt_hoas_base`

Extends `Itt_nat`

Extends `Itt_list2`

Extends `Itt_fun2`

Extends `Itt_list3`

B.2.2 Terms

The $B\ x :{}^{n}.\ t[x]$ expression, where n is a natural number, represents a "telescope" of n nested `bind` operations. Namely, it stands for $B\ v_1.\ B\ v_2.\ \ldots\ (B\ v_n.\ t[[v_1; v_2; \ldots; v_n]])$.

We also provide an input form $bind\{n;\ \ t\}$ for the important case of a vector binding that introduces a variable that does not occur freely in the bterm body.

The $bt\ @_n\ t$ expression represents the result of substituting term t for the $n + 1$-st binding of the bterm bt.

The $bt @_l tl$ expression represents the result of simultaneous substitution of terms tl (tl must be a list) for the first $\mid tl \mid$ bindings of the bterm bt.

define `unfold_bindn` :

`Itt_hoas_vector!bind{'n<|!!|>; x. 't<|!!|>['x]}`

(displayed as "$B\ x :^{n}.\ t[x]$"**)** $\longleftrightarrow$

$(Ind(n)\ where\ Ind(n)\ =$

$n\ =\ 0 \Rightarrow Ind(n)\ =\ \lambda f.f\ []$

$n > 0 \Rightarrow Ind(n)\ =\ \lambda f.B\ v.\ Ind(n-1)\ (\lambda l.f\ \ v :: l))\ (\lambda x.t[x])$

define `unfold_substn` :

`Itt_hoas_vector!subst{'n<|!!|>; 'bt<|!!|>; 't<|!!|>}`

(displayed as "$bt\ @_n\ t$"**)** $\longleftrightarrow$

$(Ind(n)\ where\ Ind(n)\ =$

$n\ =\ 0 \Rightarrow Ind(n)\ =\ \lambda bt.bt@t$

$n > 0 \Rightarrow Ind(n)\ =\ \lambda bt.B\ v.\ Ind(n-1)\ (bt@v))\ bt$

define `unfold_substl` :

`Itt_hoas_vector!substl{'bt<|!!|>; 'tl<|!!|>}`

(displayed as "$bt@_l tl$"**)** $\longleftrightarrow$

$match\ tl\ with\ [] -> \ (\lambda b.b)\ \mid\ h :: .f\ ->\ (\lambda b.f\ (b@h))\ bt$

define iform `simple_bindn` :

`Itt_hoas_vector!bind{'n<|!!|>; 't<|!!|>}`

(displayed as "$bind\{n;\ t\}$"**)** $\longleftrightarrow$

$B\ :^{n}.\ t$

define iform `bindn_list` :

`Itt_hoas_vector!bind_list{'n<|!!|>; 'terms<|!!|>}`

(displayed as "$bind_list\{n;\ terms\}$"**)** $\longleftrightarrow$

$map(bt.bind\{n;\ bt\}; terms)$

define iform `substl_list` :

`Itt_hoas_vector!substl_list{'terms<|!!|>; 'v<|!!|>}`

(displayed as "$substl_list\{terms;\ v\}$"**)** $\longleftrightarrow$

$terms@_l v$

B.2.3 Rewrites

$*[1,20]$ **rewrite** `reduce_bindn_base` {| **public** reduce |} :

$B\ x :^{0}.\ t[x] \longleftrightarrow t[[]]$

$*[1,16]$ **rewrite** `reduce_bindn_up`

{| **public** reduce Ocaml!lab

[3742:n,

3768:n,

"labels":s]

{some{denormalize_labels}} |} :

$n \in \mathbb{N} \longrightarrow$

$B\ l :^{n+1}.\ t[l] \longleftrightarrow B\ v.\ B\ l :^{n}.\ t[v :: l]$

$*[2,56]$ **rewrite** `reduce_bindn_upleft` :

$n \in \mathbb{N} \longrightarrow$

$B\ l :^{1+n}.\ t[l] \longleftrightarrow B\ v.\ B\ l :^{n}.\ t[v :: l]$

$*[1,36]$ **rewrite** `bind_into_bindone` :

$B\ v.\ t[v] \longleftrightarrow B\ l :^{1}.\ t[hd\{l\}]$

$*[1,23]$ **rewrite** `bindone_into_bind` :

$B\ l :^{1}.\ t[l] \longleftrightarrow B\ v.\ t[[v]]$

$*[7,437]$ **rewrite** `split_bind_sum` :

$m \in \mathbb{N} \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

$B\ l :^{m+n}.\ t[l] \longleftrightarrow B\ l_1 :^{m}.\ B\ l_2 :^{n}.\ t[l_1\ @\ l_2]$

$*[2,26]$ **rewrite** `reduce_bindn_right` :

$n \in \mathbb{N} \longrightarrow$

$B\ l :^{n+1}.\ t[l] \longleftrightarrow B\ l :^{n}.\ B\ v.\ t[l\ @\ [v]]$

$*[1,10]$ **rewrite** `merge_bindn` {| **public** reduce |} :

$m \in \mathbb{N} \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

$B\ :^{m}.\ B\ :^{n}.\ t \longleftrightarrow B\ :^{m+n}.\ t$

$*[1,18]$ **rewrite** `reduce_substn_base` {| **public** reduce |} :

$bt\ @_0\ t \longleftrightarrow bt@t$

$*[1,14]$ **rewrite** `reduce_substn_case` {| **public** reduce |} :

$n\ \in\ \mathbb{N} \longrightarrow$

$bt\ @_{n\ +\ 1}\ t \longleftrightarrow B\ x.\ bt@x\ @_n\ t$

$*[1,10]$ **rewrite** `reduce_bindn_subst` {| **public** reduce |} :

$n\ \in\ \mathbb{N} \longrightarrow$

$B\ v :^{n\ +\ 1}.\ bt[v]@t \longleftrightarrow B\ v :^{n}.\ bt[t :: v]$

$*[8,1766]$ **rewrite** `reduce_substn_bindn1` `bind`$(x.bt[x])$:

$m\ \in\ \mathbb{N} \longrightarrow$

$n\ \in\ \mathbb{N} \longrightarrow$

$n \geq m \longrightarrow$

$(B\ v.\ B\ l :^{n}.\ bt[v :: l])\ @_m\ t \longleftrightarrow$

$B\ l :^{n}.\ bt[$`insert_at`$(l,\ m,\ t)]$

$*[1,18]$ **rewrite** `reduce_substn_bindn2` {| **public** reduce |} :

$m\ \in\ \mathbb{N} \longrightarrow$

$n\ \in\ \mathbb{N} \longrightarrow$

$n \geq m \longrightarrow$

$B\ l :^{n\ +\ 1}.\ bt[l]\ @_m\ t \longleftrightarrow B\ l :^{n}.\ bt[$`insert_at`$(l,\ m,\ t)]$

$*[1,10]$ **rewrite** `reduce_substl_base` {| **public** reduce |} :

$bt@_l[] \longleftrightarrow bt$

$*[1,12]$ **rewrite** `reduce_substl_step`

{| **public** reduce Ocaml!lab

[6051:n,

6077:n,

"labels":s]

{some{denormalize_labels}} |} :

$bt@_l h :: t \longleftrightarrow bt@h@_l t$

$*[1,14]$ **rewrite** `reduce_substl_step1` {| **public** reduce |} :

$(B\ v.\ bt[v])@_l h :: t \longleftrightarrow bt[h]@_l t$

$*[1, 65]$ **rewrite** `reduce_substl_step2` $\{|$ **public** reduce $|\}$:

$n \in \mathbb{N} \longrightarrow$

$B\ v :^{n+1}.\ bt[v]@_l h :: t \longleftrightarrow B\ v :^{n}.\ bt[h :: v]@_l t$

$*[3, 82]$ **rewrite** `reduce_substl_bindn1` $\{|$ **public** reduce $|\}$:

$l \in List \longrightarrow$

$B\ v :^{|l|}.\ bt[v]@_l l \longleftrightarrow bt[l]$

$*[3, 1092]$ **rewrite** `reduce_substl_bindn2` :

$l \in List \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

$n \geq\mid l \mid \longrightarrow$

$B\ v :^{n}.\ bt[v]@_l l \longleftrightarrow B\ v :^{n - |l|}.\ bt[l\ @\ v]$

$*[2, 100]$ **rewrite** `reduce_bsb1` $\{|$ **public** reduce $|\}$:

$n \in \mathbb{N} \longrightarrow$

$B\ v :^{n}.\ B\ w :^{n}.\ bt[w]@_l v \longleftrightarrow B\ w :^{n}.\ bt[w]$

$*[1, 20]$ **rewrite** `reduce_bsb2` $\{|$ **public** reduce $|\}$:

$n \in \mathbb{N} \longrightarrow$

$m \in \mathbb{N} \longrightarrow$

$B\ v :^{n}.\ B\ w :^{n+m}.\ bt[w]@_l v \longleftrightarrow B\ w :^{n+m}.\ bt[w]$

$*[1, 16]$ **rewrite** `unfold_bindnsub` :

$n \in \mathbb{N} \longrightarrow$

$B\ v :^{n+1}.\ bt[v]@_l v \longleftrightarrow B\ u.\ B\ v :^{n}.\ bt[u :: v]@u@_l v$

$*[1, 8]$ **rewrite** `subst_to_substl` : $e@x \longleftrightarrow e@_l[x]$

$*[2, 105]$ **rewrite** `bindn_to_list_of_fun` :

$n \in \mathbb{N} \longrightarrow$

$B\ x :^{n}.\ e[x] \longleftrightarrow B\ x :^{n}.\ e[list_of_fun\{i.\ x_i;\ n\}]$

$*[1, 26]$ **rewrite** `coalesce_bindn_bindn` :

$n \in \mathbb{N} \longrightarrow$

$m \in \mathbb{N} \longrightarrow$

$B\ x :^{n}.\ B\ y :^{m}.\ e[x;\ y] \longleftrightarrow$

$B\ x :^{n+m}.\ e[nth_prefix\{x;\ n\};\ nth_suffix\{x;\ n\}]$

$*[2,71]$ **rewrite** `substl_append_right` :

$l \in List \longrightarrow$

$e@_l(l\ @\ [x]) \longleftrightarrow e@_l l@x$

$*[1,6]$ **rewrite** `substl_append_left` :

$l \in List \longrightarrow$

$e@x@_l l \longleftrightarrow e@_l x :: l$

$*[5,92]$ **rewrite** `substl_substl_list` :

$l_1 \in List \longrightarrow$

$l_2 \in List \longrightarrow$

$e@_l l_1 @_l l_2 \longleftrightarrow e@_l(l_1\ @\ l_2)$

$*[1,16]$ **rewrite** `substl_substl_lof` :

$n \in \mathbb{N} \longrightarrow$

$m \in \mathbb{N} \longrightarrow$

$e@_l list_of_fun\{x.\ f[x];\ m\}@_l list_of_fun\{x.\ g[x];\ n\} \longleftrightarrow$

$e@_l(list_of_fun\{x.\ f[x];\ m\}\ @\ list_of_fun\{x.\ g[x];\ n\})$

B.3 Itt_hoas_debruijn module

The `Itt_hoas_debruijn` module defines a mapping from de Bruijn-like representation of syntax into the HOAS.

B.3.1 Parents

Extends `Itt_hoas_base`

Extends `Itt_hoas_vector`

Extends `Itt_nat`

Extends `Itt_list2`

Extends `Itt_image2`

B.3.2 Terms

B.3.2.1 A de Bruijn-like representation of syntax

Our de Bruijn-like representation of (bound) terms consists of two operators. `var`(*left*, *right*) represents a variable bterm, whose "left index" is *left* and whose "right index" is *right*. Namely, it represent the term $B\ x_1.\ \ldots\ (B\ x_left.\ B\ y.\ B\ z_1.\ \ldots\ (B\ z_right.\ v)\ \ldots)\ \ldots$.

The `mk_bterm`(n; op; btl) represents the compound term of depth n. In other words, `mk_bterm`(n; op; $[B\ v :{}^{n}.\ bt_1[v]; \ldots ; B\ v :{}^{n}.\ bt_k[v]]$) is $B\ v :{}^{n}.\ T(op;\ [bt_1[v]; \ldots ; bt_k[v]])$.

define `unfold_var` :

```
Itt_hoas_debruijn!var{'left<|!!|>; 'right<|!!|>}
```

(displayed as "`var`(*left*, *right*)"**)** $\longleftrightarrow$

$B\ x :{}^{left}.\ B\ v.\ B\ x :{}^{right}.\ v$

define `unfold_mk_bterm` :

```
Itt_hoas_debruijn!mk_bterm
    {'n<|!!|>;
     'op<|!!|>;
     'btl<|!!|>}
```

(displayed as "`mk_bterm`(n; op; btl)"**)** $\longleftrightarrow$

$(Ind(n)\ where\ Ind(n)\ =$

$n\ =\ 0 \Rightarrow Ind(n)\ =\ \lambda btl.T(op;\ btl)$

$n > 0 \Rightarrow Ind(n)\ =\ \lambda btl.B\ v.\ Ind(n-1)\ ((btl@v)))\ btl$

B.3.2.2 Basic operations on syntax

`D` bt is the "binding depth" (i.e. the number of outer bindings) of a bterm bt.

`l` v and `r` v provide a way of computing the l and r indices of a variable `var`(l, r).

try `get_op` bt **with** `Not_found ->` op returns the bt's operator, if bt is a `mk_bterm` and returns op if bt is a variable.

`subterms` bt computes the subterms of the bterm bt.

define `unfold_bdepth` :

`Itt_hoas_debruijn!bdepth{'bt<|!!|>}`

(displayed as "`D` bt"**)** $\longleftrightarrow$

$fix(f.\lambda bt.weak_dest_terms$

$\{bt;$

$1 + (f\ (bt@T(\cdot;\ [])));$

$y.\ 0\})\ bt$

define `unfold_left` :

`Itt_hoas_debruijn!left{'bt<|!!|>}`

(displayed as "`l` bt"**)** $\longleftrightarrow$

$fix(f.\lambda bt.\lambda l.$**match** bt **with**

$B\ _ -> f\ (bt@T(l;\ []))\ (l + 1)$

$|\ T(op;\ ) -> op)\ bt\ 0$

define `unfold_right` :

`Itt_hoas_debruijn!right{'bt<|!!|>}`

(displayed as "`r` bt"**)** $\longleftrightarrow$

$(\mathtt{D}\ bt) - (\mathtt{l}\ bt) - 1$

define `unfold_get_op` :

`Itt_hoas_debruijn!get_op{'bt<|!!|>; 'op<|!!|>}`

(displayed as

"**try** `get_op` bt **with** `Not_found ->` op"**)** $\longleftrightarrow$

$fix(f.\lambda bt.$**match** bt **with**

$B\ _ -> f\ (bt@T(op;\ []))$

$|\ T(op;\ ) -> op)\ bt$

declare `Itt_hoas_debruijn!not_found`

(displayed as "not_found"**)**

define iform `unfold_get_op1` :

`Itt_hoas_debruijn!get_op{'bt<|!!|>}`

(displayed as "$get_op\{bt\}$"**)** $\longleftrightarrow$

try `get_op` bt **with** `Not_found ->` not_found

define `unfold_num_subterms` :

`Itt_hoas_debruijn!num_subterms{'bt<|!!|>}`

(displayed as "$num_subterms\{bt\}$"**)** $\longleftrightarrow$

$fix(f.\lambda bt.$**match** bt **with**

$B\ _ -> \ f\ (bt@T(\cdot;\ []))$

$|\ T(;\ btl) -> |\ btl\ |)\ bt$

define `unfold_nth_subterm` :

`Itt_hoas_debruijn!nth_subterm{'bt<|!!|>; 'n<|!!|>}`

(displayed as "$nth_subterm\{bt;\ n\}$"**)** $\longleftrightarrow$

$fix(f.\lambda bt.$**match** bt **with**

$B\ _ -> \ B\ v.\ f\ (bt@v)$

$|\ T(;\ btl) -> \ btl_n)\ bt$

define `unfold_subterms` :

`Itt_hoas_debruijn!subterms{'bt<|!!|>}`

(displayed as "`subterms` bt"**)** $\longleftrightarrow$

$list_of_fun$

$\{n.\ nth_subterm\{bt;\ n\};$

$num_subterms\{bt\}\}$

B.3.3 Rewrites

$*[1, 18]$ **rewrite** `reduce_mk_bterm_base`

{| **public** reduce Ocaml!lab

[4431:n,

4457:n,

"labels":s]

{some{denormalize_labels}} |} :

`mk_bterm`$(0;\ op;\ btl) \longleftrightarrow T(op;\ btl)$

$*[1, 14]$ **rewrite** `reduce_mk_bterm_step`

{| **public** reduce Ocaml!lab

[4561:n,

4587:n,

"labels":s]

{some{denormalize_labels}} |} :

$n \in \mathbb{N} \longrightarrow$

$\texttt{mk_bterm}(n + 1; op; btl) \longleftrightarrow B\ v.\ \texttt{mk_bterm}(n; op; btl@v)$

$*[1, 50]$ **rewrite** `reduce_mk_bterm_empty`

{| **public** reduce Ocaml!lab

[4744:n,

4770:n,

"labels":s]

{some{denormalize_labels}} |} :

$n \in \mathbb{N} \longrightarrow$

$\texttt{mk_bterm}(n; op; []) \longleftrightarrow B\ x :^{n}.\ T(op; [])$

$*[1, 6]$ **rewrite** `fold_mk_term` :

$T(op; subterms) \longleftrightarrow \texttt{mk_bterm}(0; op; subterms)$

$*[1, 12]$ **rewrite** `reduce_bdepth_mk_term` {| **public** reduce |} :

$\texttt{D}\ T(op; btl) \longleftrightarrow 0$

$*[1, 12]$ **rewrite** `reduce_bdepth_mk_terms`

{| **public** reduce |} :

$\texttt{D}\ mk_terms\{e\} \longleftrightarrow 0$

$*[1, 16]$ **rewrite** `reduce_bdepth_bind` {| **public** reduce |} :

$\texttt{D}\ (B\ v.\ t[v]) \longleftrightarrow 1 + (\texttt{D}\ t[T(\cdot; [])])$

$*[5, 1437]$ **rewrite** `reduce_bdepth_var` {| **public** reduce |} :

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$\texttt{D}\ \texttt{var}(l, r) \longleftrightarrow (l + r) + 1$

$*[4, 79]$ **rewrite** `reduce_bdepth_mk_bterm`

{| **public** reduce |} :

$n \in \mathbb{N} \longrightarrow$

$\texttt{D}\ \texttt{mk_bterm}(n;\ op;\ btl) \longleftrightarrow n$

$*[4, 128]$ **rewrite** `reduce_getop_var` {| **public** reduce |} :

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

try `get_op` `var`(l, r) **with** `Not_found ->` $op \longleftrightarrow op$

$*[2, 91]$ **rewrite** `reduce_getop_mkbterm` {| **public** reduce |} :

$n \in \mathbb{N} \longrightarrow$

try `get_op` `mk_bterm`$(n;\ op;\ btl)$ **with** `Not_found ->` $op' \longleftrightarrow$

op

$*[2, 107]$ **rewrite** `num_subterms_id` {| **public** reduce |} :

$btl \in List \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

$num_subterms\{\texttt{mk_bterm}(n;\ op;\ btl)\} \longleftrightarrow |\ btl\ |$

$*[2, 136]$ **rewrite** `nth_subterm_id` {| **public** reduce |} :

$n \in \mathbb{N} \longrightarrow$

$m \in \mathbb{N} \longrightarrow$

$k \in \mathbb{N} \longrightarrow$

$k < m \longrightarrow$

$nth_subterm$

$\{\texttt{mk_bterm}(n;\ op;\ list_of_fun\{x.\ f[x];\ m\});$

$k\} \longleftrightarrow$

$B\ v : {}^{n}.\ f[k]@_{l}v$

$*[2, 467]$ **rewrite** `subterms_id` {| **public** reduce |} :

$btl \in List \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

`subterms` `mk_bterm`$(n;\ op;\ btl) \longleftrightarrow map(bt.B\ v : {}^{n}.\ bt@_{l}v; btl)$

$*[6, 502]$ **rewrite** `left_id` {| **public** reduce |} :

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$\texttt{l var}(l, r) \longleftrightarrow l$

$*[2, 551]$ **rewrite** `right_id` $\{|$ **public** reduce $|\}$:

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$\texttt{r var}(l, r) \longleftrightarrow r$

$*[1, 10]$ **rewrite** `subst_var0` $\{|$ **public** reduce $|\}$:

$r \in \mathbb{N} \longrightarrow$

$\texttt{var}(0, r)@t \longleftrightarrow B\ x :^{r}.\ t$

$*[1, 14]$ **rewrite** `subst_var` $\{|$ **public** reduce $|\}$:

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$\texttt{var}(l + 1, r)@t \longleftrightarrow \texttt{var}(l, r)$

$*[1, 16]$ **rewrite** `subst_mkbterm` $\{|$ **public** reduce $|\}$:

$bdepth \in \mathbb{N} \longrightarrow$

$\texttt{mk_bterm}(bdepth + 1;\ op;\ btl)@t \longleftrightarrow$

$\texttt{mk_bterm}(bdepth;\ op;\ btl@t)$

$*[1, 12]$ **rewrite** `bind_var` $\{|$ **public** reduce $|\}$:

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$B\ x.\ \texttt{var}(l, r) \longleftrightarrow \texttt{var}(l + 1, r)$

$*[1, 43]$ **rewrite** `lemma` $\{|$ **public** reduce $|\}$:

$btl \in List \longrightarrow$

$(B\ btl)@v \longleftrightarrow btl$

$*[1, 14]$ **rewrite** `bind_mkbterm` $\{|$ **public** reduce $|\}$:

$bdepth \in \mathbb{N} \longrightarrow$

$btl \in List \longrightarrow$

$B\ x.\ \texttt{mk_bterm}(bdepth;\ op;\ btl) \longleftrightarrow$

$\texttt{mk_bterm}(bdepth + 1;\ op;\ B\ btl)$

$*[1, 1297]$ **rewrite** `lemma1` $\{|$ **public** reduce $|\}$:

$r \in \mathbb{N} \longrightarrow$

$n \ \in \ \mathbb{N} \longrightarrow$

$r \geq n \longrightarrow$

$B\ gamma :^{n}.\ B\ x :^{r}.\ t@_{l}gamma \longleftrightarrow B\ x :^{r}.\ t$

$*[4, 1507]$ **rewrite** `lemma2` {| **public** reduce |} :

$l \ \in \ \mathbb{N} \longrightarrow$

$r \ \in \ \mathbb{N} \longrightarrow$

$n \ \in \ \mathbb{N} \longrightarrow$

$((l \ + \ r) \ + \ 1) \geq n \longrightarrow$

$B\ gamma :^{n}.\ \mathtt{var}(l, r)@_{l}gamma \longleftrightarrow \mathtt{var}(l, r)$

$*[4, 1279]$ **rewrite** `lemma3` {| **public** reduce |} :

$m \ \in \ \mathbb{N} \longrightarrow$

$n \ \in \ \mathbb{N} \longrightarrow$

$m \geq n \longrightarrow$

$B\ gamma :^{n}.\ \mathtt{mk_bterm}(m;\ op;\ btl)@_{l}gamma \longleftrightarrow$

$\mathtt{mk_bterm}(m;\ op;\ btl)$

Define a type of variables Var that is more abstract than the raw de Bruijn representation.

define `unfold_Var` :

`Itt_hoas_debruijn!Var` **(displayed as** "Var"**)** $\longleftrightarrow$

$Img(v : \mathbb{N} \ \times \ \mathbb{N}.\mathbf{let}\ (i, j) = v\ \mathbf{in}\ \mathtt{var}(i, j))$

let *fold_Var* = makeFoldC << Var >> unfold_Var

$*[1, 16]$ **rule** `var_type_univ` {| **public** intro [] |} :

$\langle\Gamma\rangle \ \vdash \ Var \ \in \ \mathbb{U}'_{i}$

$*[1, 6]$ **rule** `var_type_wf` {| **public** intro [] |} :

$\langle\Gamma\rangle \ \vdash \ Var$ `Type`

$*[1, 27]$ **rule** `var_wf` {| **public** intro [] |} :

`[`wf`]` $\langle\Gamma\rangle \ \vdash \ l \ \in \ \mathbb{N} \ \longrightarrow$

`[`wf`]` $\langle\Gamma\rangle \ \vdash \ r \ \in \ \mathbb{N} \ \longrightarrow$

$\langle\Gamma\rangle \vdash \texttt{var}(l, r) \in Var$

$*[3, 42]$ **rule** `var_elim` {| **public** elim [] |} Γ:

$\langle\Gamma\rangle;\ i : \mathbb{N};\ j : \mathbb{N};\ \langle\Delta[\texttt{var}(i, j)]\rangle \vdash C[\texttt{var}(i, j)] \longrightarrow$

$\langle\Gamma\rangle;\ x : Var;\ \langle\Delta[x]\rangle \vdash C[x]$

$*[1, 34]$ **rule** `var_sqsimple`

{| **public** intro []; **public** sqsimple |} :

$\langle\Gamma\rangle \vdash sqsimple\{Var\}$

Boolean operations on variables.

define `unfold_beq_var` :

`Itt_hoas_debruijn!beq_var{'x<|!!|>; 'y<|!!|>}`

(displayed as "$beq_var\{x;\ y\}$"**)** $\longleftrightarrow$

$((\texttt{l}\ x) =_b (\texttt{l}\ y)) \wedge_b ((\texttt{r}\ x) =_b (\texttt{r}\ y))$

$*[1, 30]$ **rewrite** `reduce_beq_var` :

$l_1 \in \mathbb{N} \longrightarrow$

$r_1 \in \mathbb{N} \longrightarrow$

$l_2 \in \mathbb{N} \longrightarrow$

$r_2 \in \mathbb{N} \longrightarrow$

$beq_var\{\texttt{var}(l_1, r_1);\ \texttt{var}(l_2, r_2)\} \longleftrightarrow$

$(l_1 =_b l_2) \wedge_b (r_1 =_b r_2)$

$*[3, 104]$ **rule** `beq_var_wf` {| **public** intro [] |} :

$\texttt{[}wf\texttt{]}\ \langle\Gamma\rangle \vdash x \in Var \longrightarrow$

$\texttt{[}wf\texttt{]}\ \langle\Gamma\rangle \vdash y \in Var \longrightarrow$

$\langle\Gamma\rangle \vdash beq_var\{x;\ y\} \in \mathbb{B}$

$*[5, 101]$ **rule** `beq_var_assert_intro` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash x = y \in Var \longrightarrow$

$\langle\Gamma\rangle \vdash \uparrow beq_var\{x;\ y\}$

$*[10, 193]$ **rule** `beq_var_assert_elim` {| **public** elim [] |} Γ:

$\texttt{[}wf\texttt{]}\ \langle\Gamma\rangle;\ u : \uparrow beq_var\{x;\ y\};\ \langle\Delta[u]\rangle \vdash x_{\langle|\Gamma|\rangle}[] \in Var \longrightarrow$

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\,;\ u : \uparrow beq_var\{x;\ y\};\ \langle\Delta[u]\rangle\ \vdash\ \ y_{\langle|\Gamma|\rangle}[]\ \in\ Var\ \longrightarrow$

$\langle\Gamma\rangle\,;\ u :\ x\ =\ y\ \in\ Var;\ \langle\Delta[u]\rangle\ \vdash\ \ C[u]\ \longrightarrow$

$\langle\Gamma\rangle\,;\ u : \uparrow beq_var\{x;\ y\};\ \langle\Delta[u]\rangle\ \vdash\ \ C[u]$

This is the main theorem that shows that $B\ x.\ e[x]$ commutes with $\texttt{mk_bterm}(n;\ op;\ subterms)$.

$*[1,17]$ **rewrite** `reduce_bind_of_mk_bterm_of_list_of_fun` :

$n\ \in\ \mathbb{N} \longrightarrow$

$m\ \in\ \mathbb{N} \longrightarrow$

$B\ x.\ \texttt{mk_bterm}(n;\ op;\ list_of_fun\{y.\ f[x;\ y];\ m\}) \longleftrightarrow$

$\texttt{mk_bterm}(n\ +\ 1;\ op;\ list_of_fun\{y.\ B\ x.\ f[x;\ y];\ m\})$

$*[6,149]$ **rewrite** `reduce_vec_bind_of_mk_bterm_of_list_of_fun` :

$i\ \in\ \mathbb{N} \longrightarrow$

$n\ \in\ \mathbb{N} \longrightarrow$

$m\ \in\ \mathbb{N} \longrightarrow$

$B\ x :^{i}.\ \texttt{mk_bterm}(n;\ op;\ list_of_fun\{y.\ f[x;\ y];\ m\}) \longleftrightarrow$

$\texttt{mk_bterm}(n\ +\ i;\ op;\ list_of_fun\{y.\ B\ x :^{i}.\ f[x;\ y];\ m\})$

Some equivalences on binds.

$*[3,3852]$ **rewrite** `reduce_bindn_nth` $\{|$ **public** reduce $|\}$:

$n\ \in\ \mathbb{N} \longrightarrow$

$m\ \in\ \mathbb{N} \longrightarrow$

$m\ <\ n \longrightarrow$

$B\ x :^{n}.\ x_m \longleftrightarrow \texttt{var}(m, n\ -\ m\ -\ 1)$

$*[16,2909]$ **rewrite** `reduce_bindn_nth2` $\{|$ **public** reduce $|\}$:

$n_1\ \in\ \mathbb{N} \longrightarrow$

$n_2\ \in\ \mathbb{N} \longrightarrow$

$m\ \in\ \mathbb{N} \longrightarrow$

$m\ <\ n_1 \longrightarrow$

$B\ x_1 : {}^{n_1}.\ B\ x_2 : {}^{n_2}.\ x_1_m \longleftrightarrow \mathtt{var}(m, n_1 + n_2 - m - 1)$

$*[2,12]$ **rewrite** `reduce_bind_var` {| **public** reduce |} :

$n \in \mathbb{N} \longrightarrow$

$l \in \mathbb{N} \longrightarrow$

$B\ x : {}^{n}.\ \mathtt{var}(l, r) \longleftrightarrow \mathtt{var}(n + l, r)$

B.4 Itt_hoas_operator module

The `Itt_hoas_operator` module defines a type *Operator* of abstract operators; it also establishes the connection between abstract operator type and the internal notion of syntax that is exposed by the computational bterms theory (`Base_operator`).

B.4.1 Parents

Extends `Itt_theory`

Extends `Base_operator`

Extends `Itt_nat`

Extends `Itt_list2`

B.4.2 Terms

The `Operator` type is an abstract type with a decidable equality. We only require that an operator have a fixed shape.

As in the case of concrete quoted operators, the shape of an abstract operator is a list of numbers, each stating the number of bindings the operator adds to the corresponding subterm; the length of this list is the arity of an operator.

declare const `Itt_hoas_operator!Operator`

(displayed as “*Operator*”**)**

declare `Itt_hoas_operator!shape{'op<|!!|>}`

(displayed as "$\mathit{shape}(\mathit{op})$"**)**

declare `Itt_hoas_operator!is_same_op`

`{'op_1<|!!|>;`

`'op_2<|!!|>}`

(displayed as "$\mathit{is_same_op}(op_1;\ op_2)$"**)**

B.4.3 Rules

Operator is an abstract type.

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `op_univ` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash \mathit{Operator} \in \mathbb{U}_{l'}$

$*[1,7]$ **rule** `op_type` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash \mathit{Operator}$ `Type`

Equal operators must be identical.

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `op_sqeq` {| **public** nth_hyp |} :

$\langle\Gamma\rangle \vdash op_1 = op_2 \in \mathit{Operator} \longrightarrow$

$\langle\Gamma\rangle \vdash op_1 \equiv op_2$

$*[1,10]$ **rule** `op_sqsimple`

{| **public** intro []; **public** sqsimple |} :

$\langle\Gamma\rangle \vdash \mathit{sqsimple}\{\mathit{Operator}\}$

`is_same_op` decides the equality of *Operator*.

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `is_same_op_wf` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash op_1 \in \mathit{Operator} \longrightarrow$

$\langle\Gamma\rangle \vdash op_2 \in \mathit{Operator} \longrightarrow$

$\langle\Gamma\rangle \vdash \mathit{is_same_op}(op_1;\ op_2) \in \mathbb{B}$

$![\langle\Gamma\rangle \vdash \cdot]$ **rule** `is_same_op_eq`

$\{|$ **public** intro [AutoMustComplete]; **public** nth_hyp $|\}$:

$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$

$\langle\Gamma\rangle \vdash \uparrow is_same_op(op_1;\ op_2)$

$![\langle\Gamma\rangle \vdash \cdot]$ **rule** `is_same_op_rev_eq` :

$[wf]\ \langle\Gamma\rangle \vdash op_1 \in Operator \longrightarrow$

$[wf]\ \langle\Gamma\rangle \vdash op_2 \in Operator \longrightarrow$

$\langle\Gamma\rangle \vdash \uparrow is_same_op(op_1;\ op_2) \longrightarrow$

$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator$

$*[5, 127]$ **rule** `op_decidable` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash op_1 \in Operator \longrightarrow$

$\langle\Gamma\rangle \vdash op_2 \in Operator \longrightarrow$

$\langle\Gamma\rangle \vdash Decidable(op_1 = op_2 \in Operator)$

$*[1, 14]$ **rule** `is_same_op_elim`

$\{|$ **public** elim [ThinOption thinT] $|\}$ Γ:

$[wf]\ \langle\Gamma\rangle;\ x : \uparrow is_same_op(op_1;\ op_2);\ \langle\Delta[x]\rangle \vdash op_{1\langle|\Gamma|\rangle}[] \in Operator \longrightarrow$

$[wf]\ \langle\Gamma\rangle;\ x : \uparrow is_same_op(op_1;\ op_2);\ \langle\Delta[x]\rangle \vdash op_{2\langle|\Gamma|\rangle}[] \in Operator \longrightarrow$

$[main]$

1. $\langle\Gamma\rangle$
2. $x : \uparrow is_same_op(op_1;\ op_2)$
3. $op_1 = op_2 \in Operator$
4. $\langle\Delta[x]\rangle$

$\vdash C[x] \longrightarrow$

$\langle\Gamma\rangle;\ x : \uparrow is_same_op(op_1;\ op_2);\ \langle\Delta[x]\rangle \vdash C[x]$

Each operator has a `shape` — a list of natural numbers that are meant to represent the number of bindings in each of the arguments. The length of of the list is the operator's arity.

define iform `unfold_arity` :

`Itt_hoas_operator!arity{'op<|!!|>}`
(displayed as "$arity\{op\}$"**)** $\longleftrightarrow$
$arity(op)$

$![\langle\Gamma\rangle \vdash \cdot]$ **rule** `shape_nat_list` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash shape(op) \in \mathbb{N}\ List$

$*[1, 24]$ **rule** `shape_list` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash shape(op) \in List$

$*[1, 33]$ **rule** `shape_nat_list_eq` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash shape(op_1) = shape(op_2) \in \mathbb{N}\ List$

$*[2, 44]$ **rule** `shape_int_list` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash shape(op_1) = shape(op_2) \in int\ List$

$*[1, 36]$ **rule** `arity_nat` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash arity(op_1) = arity(op_2) \in \mathbb{N}$

$*[1, 36]$ **rule** `arity_int` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash arity(op_1) = arity(op_2) \in int$

$*[3, 46]$ **rule** `shape_int_list_sq` {| **public** intro [] |} :
$\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$
$\langle\Gamma\rangle \vdash shape(op_1) \equiv shape(op_2)$

B.4.4 Concrete Operators

This section establishes the connection between the abstract notion of operator and the internal notion of operator that is exposed by the computational bterms theory

(Base_operator).

Essentially, it *postulates* that the abstract operator is compatible with the notion of operators that we have defined computationally, that the computationally-defined operations on operators act as expected, and that the syntactic operations we defined (such as shape) correspond exactly to the built-in operations of the meta-theory. In a way, this theory establishes the operator expressions as denotations for constants of the *Operator* type — this is similar to how numerals denote constants of type *int*.

First, we define a concrete representation for operators. We will represent an operator by a bterm of the form $operator[op : op]$

declare `Itt_hoas_operator!operator[op:op]`

(displayed as "$operator[op : op]$"**)**

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `op_constant` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash operator[op : op] \in Operator$

`*`$[1, 7]$ **rule** `shape_const_nat_list` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash shape(operator[op : op]) \in \mathbb{N}\ List$

`!`[] **rewrite** `bterm_shape` :

$shape(operator[op : op]) \longleftrightarrow list_of_numlist\{shape(op)\}$

`!`[] **rewrite** `bterm_same_op` :

$is_same_op(operator[op1 : op];\ operator[op2 : op])$

$\longleftrightarrow$

$meta_eq[op1 : op,\ op2 : op]\{true;\ false\}$

B.5 Itt_hoas_destterm module

The `Itt_hoas_destterm` module defines destructors for extracting from a bterm the components corresponding to the de Bruijn-like representation of that bterm.

B.5.1 Parents

Extends `Itt_hoas_base`

Extends `Itt_hoas_vector`

Extends `Itt_hoas_operator`

Extends `Itt_hoas_debruijn`

Extends `Itt_hoas_df`

B.5.2 Terms

The `is_var` operator decides whether a bterm is a `var` or a `mk_bterm`. In order to implement the `is_var` operator we assume that there exist at least two distinct operators (for any concrete notion of operators this would, of course, be trivially derivable but we would like to keep the operators type abstract at this point).

The `dest_bterm` operator is a generic destructor that can extract all the components of the de Bruijn-like representation of a bterm.

declare `Itt_hoas_destterm!op1` **(displayed as** "*op*1"**)**

declare `Itt_hoas_destterm!op2` **(displayed as** "*op*2"**)**

define `unfold_isvar` :

Itt_hoas_destterm!is_var{'bt<|!!|>}

(**displayed as** "is_var(*bt*)") ⟷

$\neg_b$*is_same_op*(**try** get_op *bt* **with** Not_found -> *op*1; **try**

get_op *bt*

with Not_found ->

*op*2)

define `unfold_dest_bterm` :

Itt_hoas_destterm!dest_bterm

{'bt<|!!|>;

l, r. 'var_case<|!!|>['l; 'r];

bdepth, op, subterms. 'op_case<|!!|>['bdepth;

'op; 'subterms]}

(displayed as

"**match** bt **with**

`var`$(l, r) -> var_case[l;\ r]$

$|\ BTerm(bdepth,\ op,\ subterms) -> op_case[bdepth;$

$op;$

$subterms]$") $\longleftrightarrow$

if `is_var`(bt) **then** $var_case[$`l` $bt;$ `r` $bt]$ **else** $op_case[$`D`

$bt;$

try `get_op` bt **with** `Not_found ->` $\cdot$;

`subterms` $bt]$

B.5.3 Rules

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `op1_op` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash op1 \in Operator$

`!`$[\langle\Gamma\rangle \vdash \cdot]$ **rule** `op2_op` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash op2 \in Operator$

B.5.4 Rewrites

`!`[] **rewrite** `ops_distict` {| **public** reduce |} :

$is_same_op(op1;\ op2) \longleftrightarrow false$

$*[1, 13]$ **rewrite** `same_op_id` {| **public** reduce |} :

$op \in Operator \longrightarrow$

$is_same_op(op;\ op) \longleftrightarrow true$

$*[1, 22]$ **rewrite** `is_var_var` {| **public** reduce |} :

$m \in \mathbb{N} \longrightarrow$

$n \in \mathbb{N} \longrightarrow$

$\texttt{is_var}(\texttt{var}(m, n)) \longleftrightarrow true$

$*[1, 20]$ **rewrite** `is_var_mk_bterm` {| **public** reduce |} :

$op \ \in \ Operator \longrightarrow$

$n \ \in \ \mathbb{N} \longrightarrow$

$\texttt{is_var}(BTerm(n, \ op, \ btl)) \longleftrightarrow false$

$*[1, 38]$ **rewrite** `dest_bterm_var` {| **public** reduce |} :

$l \ \in \ \mathbb{N} \longrightarrow$

$r \ \in \ \mathbb{N} \longrightarrow$

match $\texttt{var}(l, r)$ **with**

$\texttt{var}(l, r) -> \ var_case[l; \ r]$

$| \ BTerm(d, \ o, \ s) -> \ op_case[d; \ o; \ s] \longleftrightarrow$

$var_case[l; \ r]$

$*[1, 28]$ **rewrite** `dest_bterm_mk_bterm` {| **public** reduce |} :

$n \ \in \ \mathbb{N} \longrightarrow$

$op \ \in \ Operator \longrightarrow$

$subterms \ \in \ List \longrightarrow$

match $BTerm(n, \ op, \ subterms)$ **with**

$\texttt{var}(l, r) -> \ var_case[l; \ r]$

$| \ BTerm(bdepth, \ op, \ subterms) -> \ op_case[bdepth;$

$op;$

$subterms] \longleftrightarrow$

$op_case[n; \ op; \ map(bt.B \ v : {}^{n}. \ bt@_{l}v; subterms)]$

$*[2, 125]$ **rewrite** `mk_bterm_eta_lof` {| **public** reduce |} :

$l \ \in \ \mathbb{N} \longrightarrow$

$n \ \in \ \mathbb{N} \longrightarrow$

$BTerm(n, \ op, \ list_of_fun\{i. \ B \ v : {}^{n}. \ f[i]@_{l}v; \ l\}) \longleftrightarrow$

$BTerm(n, \ op, \ list_of_fun\{i. \ f[i]; \ l\})$

$*[1, 37]$ **rewrite** `mk_bterm_eta` {| **public** reduce |} :

$n \ \in \ \mathbb{N} \longrightarrow$

$subterms \ \in \ List \longrightarrow$

$BTerm(n,\ op,\ map(bt.B\ v :^{n}.\ bt@_l v; subterms)) \longleftrightarrow$

$BTerm(n,\ op,\ subterms)$

B.6 Itt_hoas_bterm module

The `Itt_hoas_bterm` module defines the inductive type **BTerm** and establishes the appropriate induction rules for this type.

B.6.1 Parents

Extends `Itt_hoas_destterm`

Extends `Itt_image2`

Extends `Itt_tunion`

Extends `Itt_subset`

B.6.2 Terms

We define the type **BTerm** as a recursive type. The `compatible_shapes`$(depth; shape; subterms)$ predicate defines when a list of subterms *subterms* is compatible with a specific operator.

define `unfold_compatible_shapes` :

```
Itt_hoas_bterm!compatible_shapes
    {'depth<|!!|>;
     'shape<|!!|>;
     'btl<|!!|>}
```

(displayed as

"`compatible_shapes`$(depth; shape; btl)$"**)** $\longleftrightarrow$

$\forall x,\ y\ \in\ shape,\ btl.\ ((depth\ +\ x)\ =\ \mathtt{D}\ y\ \in\ int)$

define `unfold_dom` :

`Itt_hoas_bterm!dom{'BT<|!!|>}`

(displayed as "$dom\{BT\}$"**)** $\longleftrightarrow$

$(\mathbb{N}\ \times\ \mathbb{N})\ +\ (depth : \mathbb{N}\ \times\ op : Operator\ \times\ \{subterms : BT\ List\ |$

`compatible_shapes`$(depth; shape(op); subterms)\})$

define `unfold_mk` :

`Itt_hoas_bterm!mk{'x<|!!|>}` **(displayed as** "$mk\{x\}$"**)** $\longleftrightarrow$

match x **with**

$inl\ v\ -\ >$ **let** $(left, right)$ = v **in** `var`$(left, right)$

$|\ inr\ t\ -\ >$ **let**

(d, v) = t

in

let (op, st) = v **in** $BTerm(d,\ op,\ st)$

define `unfold_dest` :

`Itt_hoas_bterm!dest{'bt<|!!|>}`

(displayed as "$dest\{bt\}$"**)** $\longleftrightarrow$

match bt **with**

`var`$(l, r)\ -\ >\ inl\ (l, r)$

$|\ BTerm(d,\ op,\ ts)\ -\ >\ inr\ (d, (op, ts))$

define `unfold_Iter` :

`Itt_hoas_bterm!Iter{'X<|!!|>}`

(displayed as "$Iter\{X\}$"**)** $\longleftrightarrow$

$Img(x : dom\{X\}.mk\{x\})$

define `unfold_BT` :

`Itt_hoas_bterm!BT{'n<|!!|>}` **(displayed as** "$BT\{n\}$"**)** $\longleftrightarrow$

$Ind(n)\ where\ Ind(n)\ =$

$n\ =\ 0 \Rightarrow Ind(n)\ =\ Void$

$n > 0 \Rightarrow Ind(n)\ =\ Iter\{Ind(n-1)\}$

define const `unfold_BTerm` :

`Itt_hoas_bterm!BTerm` **(displayed as "BTerm")** $\longleftrightarrow$

$\cup n : \mathbb{N}.BT\{n\}$

define `unfold_BTerm2` :

`Itt_hoas_bterm!BTerm{'i<|!!|>}`

(displayed as "$BTerm\{i\}$"**)** $\longleftrightarrow$

$\{e : \mathbf{BTerm} \mid \mathtt{D}\ e\ =\ i\ \in\ \mathbb{N}\}$

define `unfold_ndepth` :

`Itt_hoas_bterm!ndepth{'t<|!!|>}`

(displayed as "$ndepth\{t\}$"**)** $\longleftrightarrow$

$fix(ndepth.\lambda t.$**match** t **with**

`var`$(l, r)\ -\ >\ 1$

$\mid\ BTerm(bdepth,\ op,\ subterms)\ -\ >\ \mathtt{max}_l(map(x.ndepth\ x; subterms))\ +\ 1)\ t$

B.6.3 Rewrites

Basic facts about `compatible_shapes`

$*[1,8]$ **rewrite** `compatible_shapes_nil_nil`

$\{|$ **public** reduce $|\}$:

`compatible_shapes`$(depth; []; []) \longleftrightarrow True$

$*[1,8]$ **rewrite** `compatible_shapes_nil_cons`

$\{|$ **public** reduce $|\}$:

`compatible_shapes`$(depth; []; h_2 :: t_2) \longleftrightarrow False$

$*[1,8]$ **rewrite** `compatible_shapes_cons_nil`

$\{|$ **public** reduce $|\}$:

`compatible_shapes`$(depth; h_1 :: t_1; []) \longleftrightarrow False$

$*[1,10]$ **rewrite** `compatible_shapes_cons_cons`

$\{|$ **public** reduce $|\}$:

`compatible_shapes`$(depth; h_1 :: t_1; h_2 :: t_2)$

$\longleftrightarrow$

$((depth\ +\ h_1)\ =\ \mathtt{D}\ h_2\ \in\ int)$

$\wedge$ `compatible_shapes`$(depth; t_1; t_2)$

$*[1,16]$ **rewrite** `bt_reduce_base` {| **public** reduce |} :

$BT\{0\} \longleftrightarrow Void$

$*[1,12]$ **rewrite** `bt_reduce_step` {| **public** reduce |} :

$n\ \in\ \mathbb{N} \longrightarrow$

$BT\{n\ +\ 1\} \longleftrightarrow Iter\{BT\{n\}\}$

B.6.4 Rules

$*[1,57]$ **rule** `bt_elim_squash` {| **public** elim [] |} Γ:

$[wf]\ \ \langle\Gamma\rangle;\ \langle\Delta\rangle\ \vdash\ \ n_{\langle|\Gamma|\rangle}[]\ \in\ \mathbb{N}\ \longrightarrow$

$[base]\ \ \langle\Gamma\rangle;\ \langle\Delta\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N}\ \vdash\ \big[P[\mathtt{var}(l,r)]\big]\ \longrightarrow$

$[step]$

1. $\langle\Gamma\rangle$
2. $\langle\Delta\rangle$
3. $depth:\ \mathbb{N}$
4. $op:\ Operator$
5. $subterms:\ BT\{n_{\langle|\Gamma|\rangle}[]\}\ List$
6. `compatible_shapes`$(depth; shape(op); subterms)$

$\vdash\ \big[P[BTerm(depth,\ op,\ subterms)]\big]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ BT\{n\ +\ 1\};\ \langle\Delta\rangle\ \vdash\ \big[P[t]\big]$

$*[1,15]$ **rule** `bt_elim_squash0` {| **public** nth_hyp |} Γ:

$\langle\Gamma\rangle;\ t:\ BT\{0\};\ \langle\Delta\rangle\ \vdash\ \ P[t]$

$*[5,255]$ **rule** `bt_wf_and_bdepth_univ` {| **public** intro [] |} :

$[wf]\ \ \langle\Gamma\rangle\ \vdash\ \ n\ \in\ \mathbb{N}\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \ (BT\{n\}\ \in\ \mathbb{U}_{'l})\ \wedge\ \forall t: BT\{n\}.\ (\mathtt{D}\ t\ \in\ \mathbb{N})$

$*[5,234]$ **rule** `bt_wf_and_bdepth_wf` {| **public** intro [] |} :

$[wf]\ \ \langle\Gamma\rangle\ \vdash\ \ n\ \in\ \mathbb{N}\ \longrightarrow$

$\langle\Gamma\rangle \vdash BT\{n\}\,\mathtt{Type} \wedge \forall t : BT\{n\}.\ (\mathtt{D}\ t \in \mathbb{N})$

$*[1,13]$ **rule** `bt_univ` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash n \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash BT\{n\} \in \mathbb{U}_{l'}$

$*[1,13]$ **rule** `bt_wf` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash n \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash BT\{n\}\,\mathtt{Type}$

$*[1,15]$ **rule** `bterm_univ` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash \mathbf{BTerm} \in \mathbb{U}_{l'}$

$*[1,13]$ **rule** `bterm_wf` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash \mathbf{BTerm}\,\mathtt{Type}$

$*[1,7]$ **rule** `nil_in_list_bterm` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash [] \in \mathbf{BTerm}\ \mathit{List}$

$*[2,49]$ **rule** `bdepth_wf` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash t \in \mathbf{BTerm} \longrightarrow$

$\langle\Gamma\rangle \vdash \mathtt{D}\ t \in \mathbb{N}$

$*[1,13]$ **rule** `bdepth_wf_int` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash t \in \mathbf{BTerm} \longrightarrow$

$\langle\Gamma\rangle \vdash \mathtt{D}\ t \in \mathit{int}$

$*[1,13]$ **rule** `bdepth_wf_positive` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash t \in \mathbf{BTerm} \longrightarrow$

$\langle\Gamma\rangle \vdash (\mathtt{D}\ t) \geq 0$

$*[1,20]$ **rule** `bterm2_wf` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash n \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash BTerm\{n\}\,\mathtt{Type}$

$*[1,16]$ **rule** `bterm2_forward`

{| **public** forward []; **public** nth_hyp |} Γ:

1. $\langle\Gamma\rangle$
2. $x : e \in BTerm\{d\}$

3. $\langle\Delta[x]\rangle$

4. $e_{\langle|\Gamma|\rangle}[] \; \in \; \mathbf{BTerm}$

5. $\mathtt{D}\; e_{\langle|\Gamma|\rangle}[] \; = \; d_{\langle|\Gamma|\rangle}[] \; \in \; \mathbb{N}$

$\vdash \;\; C[x] \;\longrightarrow$

$\langle\Gamma\rangle;\; x:\; e \;\in\; BTerm\{d\};\; \langle\Delta[x]\rangle \;\vdash\;\; C[x]$

$*[1,10]$ **rule** `bterm2_is_bterm` $\{|$ **public** nth_hyp $|\}$ Γ:

$\langle\Gamma\rangle;\; x:\; BTerm\{d\};\; \langle\Delta[x]\rangle \;\vdash\;\; x \;\in\; \mathbf{BTerm}$

$*[2,57]$ **rule** `compatible_shapes_univ` $\{|$ **public** intro [] $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{bdepth} \;\in\; \mathbb{N} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{shape} \;\in\; \mathit{int\ List} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{btl} \;\in\; \mathbf{BTerm}\ \mathit{List} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; \mathtt{compatible_shapes}(\mathit{bdepth}; \mathit{shape}; \mathit{btl}) \;\in\; \mathbb{U}_{'l}$

$*[2,53]$ **rule** `compatible_shapes_wf` $\{|$ **public** intro [] $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{bdepth} \;\in\; \mathbb{N} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{shape} \;\in\; \mathit{int\ List} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{btl} \;\in\; \mathbf{BTerm}\ \mathit{List} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; \mathtt{compatible_shapes}(\mathit{bdepth}; \mathit{shape}; \mathit{btl})\, \mathtt{Type}$

$*[2,35]$ **rule** `bt_subtype_bterm` $\{|$ **public** intro [] $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; n \;\in\; \mathbb{N} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; BT\{n\} \;\sqsubseteq\; \mathbf{BTerm}$

$*[1,59]$ **rule** `dom_wf1` $\{|$ **public** intro [] $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; n \;\in\; \mathbb{N} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; \mathit{dom}\{BT\{n\}\}\, \mathtt{Type}$

$*[1,42]$ **rule** `compatible_shapes_sqstable` :

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{bdepth} \;\in\; \mathit{int} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{shape} \;\in\; \mathit{int\ List} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle \;\vdash\;\; \mathit{btl} \;\in\; \mathbf{BTerm}\ \mathit{List} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; \big[\mathtt{compatible_shapes}(\mathit{bdepth}; \mathit{shape}; \mathit{btl})\big] \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\;\; \mathtt{compatible_shapes}(\mathit{bdepth}; \mathit{shape}; \mathit{btl})$

$*[1,55]$ **rule** `dom_wf` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash T \sqsubseteq$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash dom\{T\}$ `Type`

$*[2,186]$ **rule** `dom_monotone` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash T \sqsubseteq S \longrightarrow$

$\langle\Gamma\rangle \vdash S \sqsubseteq$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash dom\{T\} \sqsubseteq dom\{S\}$

$*[2,176]$ **rule** `dom_monotone_set` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash T \subseteq S \longrightarrow$

$\langle\Gamma\rangle \vdash S \sqsubseteq$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash dom\{T\} \subseteq dom\{S\}$

$*[1,14]$ **rule** `iter_monotone` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash T \sqsubseteq S \longrightarrow$

$\langle\Gamma\rangle \vdash S \sqsubseteq$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash Iter\{T\} \sqsubseteq Iter\{S\}$

$*[1,80]$ **rule** `bt_monotone` $\{|$ **public** intro [] $|\}$:

$[wf]$ $\langle\Gamma\rangle \vdash n \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash BT\{n\} \sqsubseteq BT\{n + 1\}$

$*[3,72]$ **rule** `var_wf0` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \vdash X \sqsubseteq$ **BTerm** $\longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash l \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash r \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash$ `var`$(l,r) \in Iter\{X\}$

$*[2,56]$ **rule** `var_wf` $\{|$ **public** intro [] $|\}$:

$[wf]$ $\langle\Gamma\rangle \vdash l \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash r \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash$ `var`$(l,r) \in$ **BTerm**

$*[3,161]$ **rule** `mk_bterm_bt_wf` $\{|$ **public** intro [] $|\}$:

$[wf]$ $\langle\Gamma\rangle \vdash n \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash depth \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash op \in Operator \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $subterms \in BT\{n\}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash$ $\texttt{compatible_shapes}(depth; shape(op); subterms) \longrightarrow$

$\langle\Gamma\rangle \vdash$ $BTerm(depth,\ op,\ subterms) \in BT\{n + 1\}$

$*[6,114]$ **rule** `mk_bterm_wf` {| **public** intro [] |} :

$[wf]$ $\langle\Gamma\rangle \vdash$ $depth \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $op \in Operator \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $subterms \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash$ $\texttt{compatible_shapes}(depth; shape(op); subterms) \longrightarrow$

$\langle\Gamma\rangle \vdash$ $BTerm(depth,\ op,\ subterms) \in \mathbf{BTerm}$

$*[2,72]$ **rule** `mk_bterm_wf2` {| **public** intro [] |} :

$[wf]$ $\langle\Gamma\rangle \vdash$ $d_1 = d_2 \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $op \in Operator \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $subterms \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash$ $\texttt{compatible_shapes}(d_1; shape(op); subterms) \longrightarrow$

$\langle\Gamma\rangle \vdash$ $BTerm(d_1,\ op,\ subterms) \in BTerm\{d_2\}$

$*[3,21]$ **rule** `mk_term_wf` {| **public** intro [] |} :

$[wf]$ $\langle\Gamma\rangle \vdash$ $op \in Operator \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $subterms \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash$ $\texttt{compatible_shapes}(0; shape(op); subterms) \longrightarrow$

$\langle\Gamma\rangle \vdash$ $Term(op,\ subterms) \in \mathbf{BTerm}$

$*[1,18]$ **rule** `mk_term_wf2` {| **public** intro [] |} :

$[wf]$ $\langle\Gamma\rangle \vdash$ $d = 0 \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $op \in Operator \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash$ $subterms \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash$ $\texttt{compatible_shapes}(0; shape(op); subterms) \longrightarrow$

$\langle\Gamma\rangle \vdash$ $Term(op,\ subterms) \in BTerm\{d\}$

$*[7,559]$ **rule** `bt_elim_squash2` {| **public** elim [] |} Γ:

$[wf]$ $\langle\Gamma\rangle;\ \langle\Delta\rangle \vdash$ $n_{\langle|\Gamma|\rangle}[] \in \mathbb{N} \longrightarrow$

$[base]$ $\langle\Gamma\rangle;\ \langle\Delta\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N} \vdash$ $\left[P[\texttt{var}(l,r)]\right] \longrightarrow$

$[step]$

1. $\langle\Gamma\rangle$
2. $n \ > \ 0$
3. $\langle\Delta\rangle$
4. $depth : \ \mathbb{N}$
5. $op : \ Operator$
6. $subterms : \ BT\{n_{\langle|\Gamma|\rangle}[] \ - \ 1\} \ List$
7. $\texttt{compatible_shapes}(depth; shape(op); subterms)$

$\vdash \ \left[P[BTerm(depth, \ op, \ subterms)]\right] \ \longrightarrow$

$\langle\Gamma\rangle ; \ t : \ BT\{n\}; \ \langle\Delta\rangle \ \vdash \ \left[P[t]\right]$

$*[2, 250]$ **rule** `bterm_elim_squash`

{| **public** elim [ThinFirst thinT] |} Γ:

$\langle\Gamma\rangle ; \ \langle\Delta\rangle ; \ l : \ \mathbb{N}; \ r : \ \mathbb{N} \ \vdash \ \left[P[\texttt{var}(l, r)]\right] \ \longrightarrow$

1. $\langle\Gamma\rangle$
2. $\langle\Delta\rangle$
3. $depth : \ \mathbb{N}$
4. $op : \ Operator$
5. $subterms : \ \mathbf{BTerm} \ List$
6. $\texttt{compatible_shapes}(depth; shape(op); subterms)$

$\vdash \ \left[P[BTerm(depth, \ op, \ subterms)]\right] \ \longrightarrow$

$\langle\Gamma\rangle ; \ t : \ \mathbf{BTerm}; \ \langle\Delta\rangle \ \vdash \ \left[P[t]\right]$

$*[4, 105]$ **rule** `bterm_induction_squash1` Γ:

$\langle\Gamma\rangle ; \ \langle\Delta\rangle ; \ l : \ \mathbb{N}; \ r : \ \mathbb{N} \ \vdash \ \left[P[\texttt{var}(l, r)]\right] \ \longrightarrow$

1. $\langle\Gamma\rangle$
2. $\langle\Delta\rangle$
3. $n : \ \mathbb{N}$
4. $depth : \ \mathbb{N}$
5. $op : \ Operator$
6. $subterms : \ BT\{n\} \ List$

7. $\texttt{compatible_shapes}(\mathit{depth}; \mathit{shape}(\mathit{op}); \mathit{subterms})$

8. $\forall t \in \mathit{subterms}.\ \big[P[t]\big]$

$\vdash\ \big[P[\mathit{BTerm}(\mathit{depth},\ \mathit{op},\ \mathit{subterms})]\big]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta\rangle\ \vdash\ \big[P[t]\big]$

$*[8, 514]$ **rewrite** `bind_eta` $\{|$ **public** reduce $|\}$:

$bt\ \in\ \mathbf{BTerm} \longrightarrow$

$(\texttt{D}\ bt)\ >\ 0 \longrightarrow$

$B\ x.\ bt@x \longleftrightarrow bt$

$*[2, 1197]$ **rewrite** `bind_vec_eta` $\{|$ **public** reduce $|\}$:

$n\ \in\ \mathbb{N} \longrightarrow$

$bt\ \in\ \mathbf{BTerm} \longrightarrow$

$(\texttt{D}\ bt) \geq n \longrightarrow$

$B\ \mathit{gamma} : {}^{n}.\ bt@_l \mathit{gamma} \longleftrightarrow bt$

$*[4, 634]$ **rewrite** `subterms_lemma` $\{|$ **public** reduce $|\}$:

$n\ \in\ \mathbb{N} \longrightarrow$

$\mathit{subterms}\ \in\ \mathbf{BTerm}\ \mathit{List} \longrightarrow$

$\forall i : \mathit{Index}(\mathit{subterms}).\ ((\texttt{D}\ (\mathit{subterms_i})) \geq n) \longrightarrow$

$\mathit{map}(bt.B\ v : {}^{n}.\ bt@_l v; \mathit{subterms}) \longleftrightarrow \mathit{subterms}$

$*[5, 1538]$ **rule** `subterms_depth`

$\{|$ **public** intro $[]\ |\}$ $\mathit{shape}_{\langle|\Gamma|\rangle}[]$:

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{bdepth}\ \in\ \mathbb{N}\ \longrightarrow$

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{shape}\ \in\ \mathbb{N}\ \mathit{List}\ \longrightarrow$

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{btl}\ \in\ \mathbf{BTerm}\ \mathit{List}\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \ \texttt{compatible_shapes}(\mathit{bdepth}; \mathit{shape}; \mathit{btl})\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \ \forall i : \mathit{Index}(\mathit{btl}).\ ((\texttt{D}\ (\mathit{btl_i})) \geq \mathit{bdepth})$

$*[2, 19]$ **rule** `subterms_depth2`

$\{|$ **public** intro $[]\ |\}$ $\mathit{shape}_{\langle|\Gamma|\rangle}[]$:

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{bdepth}\ \in\ \mathbb{N}\ \longrightarrow$

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{shape}\ \in\ \mathbb{N}\ \mathit{List}\ \longrightarrow$

$\texttt{[}wf\texttt{]}\ \ \langle\Gamma\rangle\ \vdash\ \ \mathit{btl}\ \in\ \mathbf{BTerm}\ \mathit{List}\ \longrightarrow$

$\langle\Gamma\rangle \vdash$ `compatible_shapes`$(bdepth; shape; btl) \longrightarrow$

$\langle\Gamma\rangle \vdash \forall i : Index(btl).\ ((\mathtt{D}\ (btl_i)) \geq bdepth)$

$*[2,31]$ **rule** `subterms_depth3`

$\{|$ **public** intro $[]$ $|\}$ $shape_{\langle|\Gamma|\rangle}[]$:

$[wf]$ $\langle\Gamma\rangle \vdash bdepth \in \mathbb{N} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash shape \in \mathbb{N}\ List \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash btl \in$ **BTerm** $List \longrightarrow$

$\langle\Gamma\rangle \vdash$ `compatible_shapes`$(bdepth; shape; btl) \longrightarrow$

$\langle\Gamma\rangle \vdash \forall x \in btl.\ ((\mathtt{D}\ x) \geq bdepth)$

$*[2,38]$ **rewrite** `dest_bterm_mk_bterm2` $\{|$ **public** reduce $|\}$:

$n \in \mathbb{N} \longrightarrow$

$op \in Operator \longrightarrow$

$subterms \in$ **BTerm** $List \longrightarrow$

`compatible_shapes`$(n; shape(op); subterms) \longrightarrow$

match $BTerm(n,\ op,\ subterms)$ **with**

`var`$(l, r) -> var_case[l;\ r]$

$|\ BTerm(bdepth,\ op,\ subterms) -> op_case[bdepth;$

$op;$

$subterms] \longleftrightarrow$

$op_case[n;\ op;\ subterms]$

$*[3,86]$ **rewrite** `dest_bterm_mk_term` $\{|$ **public** reduce $|\}$:

$op \in Operator \longrightarrow$

$subterms \in List \longrightarrow$

match $Term(op,\ subterms)$ **with**

`var`$(l, r) -> var_case[l;\ r]$

$|\ BTerm(bdepth,\ op,\ subterms) -> op_case[bdepth;$

$op;$

$subterms] \longleftrightarrow$

$op_case[0;\ op;\ subterms]$

$*[1,79]$ **rewrite** `mk_dest_reduce` $\{|$ **public** reduce $|\}$:

$t \ \in \ \textbf{BTerm} \longrightarrow$

$mk\{dest\{t\}\} \longleftrightarrow t$

$*[1,20]$ **rewrite** `reduce_ndepth1` $\{|$ **public** reduce $|\}$:

$l \ \in \ \mathbb{N} \longrightarrow$

$r \ \in \ \mathbb{N} \longrightarrow$

$ndepth\{\mathtt{var}(l,r)\} \longleftrightarrow 1$

$*[1,26]$ **rewrite** `reduce_ndepth2` $\{|$ **public** reduce $|\}$:

$op \ \in \ Operator \longrightarrow$

$bdepth \ \in \ \mathbb{N} \longrightarrow$

$subs \ \in \ \textbf{BTerm}\ List \longrightarrow$

$\mathtt{compatible_shapes}(bdepth; shape(op); subs) \longrightarrow$

$ndepth\{BTerm(bdepth,\ op,\ subs)\} \longleftrightarrow$

$\mathtt{max}_l(map(x.ndepth\{x\}; subs)) \ + \ 1$

$*[2,228]$ **rule** `iter_monotone_set` $\{|$ **public** intro [] $|\}$:

$\langle\Gamma\rangle \ \vdash \ \ T \ \subseteq \ S \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ S \ \sqsubseteq \ \textbf{BTerm} \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ Iter\{T\} \ \subseteq \ Iter\{S\}$

$*[1,78]$ **rule** `bt_monotone_set` $\{|$ **public** intro [] $|\}$:

$[wf] \ \ \langle\Gamma\rangle \ \vdash \ \ n \ \in \ \mathbb{N} \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ BT\{n\} \ \subseteq \ BT\{n \ + \ 1\}$

$*[7,807]$ **rule** `bt_monotone_set2` $\{|$ **public** intro [] $|\}$:

$[wf] \ \ \langle\Gamma\rangle \ \vdash \ \ k \ \in \ \mathbb{N} \ \longrightarrow$

$[wf] \ \ \langle\Gamma\rangle \ \vdash \ \ n \ \in \ \mathbb{N} \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ n \geq k \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ BT\{k\} \ \subseteq \ BT\{n\}$

$*[3,163]$ **rule** `ndepth_wf` $\{|$ **public** intro [] $|\}$:

$[wf] \ \ \langle\Gamma\rangle \ \vdash \ \ t \ \in \ \textbf{BTerm} \ \longrightarrow$

$\langle\Gamma\rangle \ \vdash \ \ ndepth\{t\} \ \in \ \mathbb{N}$

$*[11,436]$ **rule** `ndepth_correct` $\{|$ **public** intro [] $|\}$:

$[wf] \ \ \langle\Gamma\rangle \ \vdash \ \ t \ \in \ \textbf{BTerm} \ \longrightarrow$

$\langle\Gamma\rangle \;\vdash\; t \;\in\; BT\{ndepth\{t\}\}$

$*[2,42]$ **rule** `bt_subset_bterm` $\{|$ **public** intro $[]$ $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\; n \;\in\; \mathbb{N} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\; BT\{n\} \;\subseteq\; \mathbf{BTerm}$

$*[1,81]$ **rule** `dest_bterm_wf` $\{|$ **public** intro $[]$ $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\; bt \;\in\; \mathbf{BTerm} \;\longrightarrow$

$[wf]\;\; \langle\Gamma\rangle;\; l:\; \mathbb{N};\; r:\; \mathbb{N} \;\vdash\; var_case[l;\; r] \;\in\; T \;\longrightarrow$

$[wf]$

1. $\langle\Gamma\rangle$
2. $bdepth:\; \mathbb{N}$
3. $op:\; Operator$
4. $subterms:\; \mathbf{BTerm}\; List$
5. $\texttt{compatible_shapes}(bdepth; shape(op); subterms)$

$\vdash\;\; op_case[bdepth;\; op;\; subterms] \;\in\; T \;\longrightarrow$

1. $\langle\Gamma\rangle$

$\vdash$

match bt **with**

$\texttt{var}(l,r) -> \; var_case[l;\; r]$

$|\;\; BTerm(bdepth,\; op,\; subterms) -> \; op_case[bdepth;$

$op;$

$subterms] \;\in$

T

$*[1,183]$ **rule** `dest_wf` $\{|$ **public** intro $[]$ $|\}$:

$[wf]\;\; \langle\Gamma\rangle \;\vdash\; t \;\in\; \mathbf{BTerm} \;\longrightarrow$

$\langle\Gamma\rangle \;\vdash\; dest\{t\} \;\in\; dom\{\mathbf{BTerm}\}$

$*[3,102]$ **rule** `bterm_elim` $\{|$ **public** elim $[]$ $|\}$ Γ:

$\langle\Gamma\rangle;\; \langle\Delta\rangle;\; l:\; \mathbb{N};\; r:\; \mathbb{N} \;\vdash\; P[\texttt{var}(l,r)] \;\longrightarrow$

1. $\langle\Gamma\rangle$

2. $\langle\Delta\rangle$

3. $bdepth:\ \mathbb{N}$

4. $op:\ Operator$

5. $subterms:$ **BTerm** *List*

6. $\texttt{compatible_shapes}(bdepth; shape(op); subterms)$

$\vdash\ P[BTerm(bdepth,\ op,\ subterms)]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta\rangle\ \vdash\ P[t]$

$*[1,17]$ **rule** `dom_elim` $\{|$ **public** elim $[]\ |\}\ \Gamma:$

$\langle\Gamma\rangle;\ dom\{T\};\ u:\ \mathbb{N}\ \times\ \mathbb{N};\ \langle\Delta[inl\ u]\rangle\ \vdash\ P[inl\ u]\ \longrightarrow$

1. $\langle\Gamma\rangle$

2. $dom\{T\}$

3. $v:$

$depth:\mathbb{N}\ \times\ op:Operator\ \times\ \{subterms:T\ List\ |$

$\texttt{compatible_shapes}(depth; shape(op); subterms)\}$

4. $\langle\Delta[inr\ v]\rangle$

$\vdash\ P[inr\ v]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ dom\{T\};\ \langle\Delta[t]\rangle\ \vdash\ P[t]$

$*[1,98]$ **rewrite** `dest_mk_reduce` $n:$

$n\ \in\ \mathbb{N}\longrightarrow$

$t\ \in\ dom\{BT\{n\}\}\longrightarrow$

$dest\{mk\{t\}\}\longleftrightarrow t$

$*[3,41]$ **rule** `bt_elim1` $\{|$ **public** elim $[]\ |\}\ \Gamma:$

$[wf]\ \ \langle\Gamma\rangle;\ t:\ BT\{n\ +\ 1\};\ \langle\Delta[t]\rangle\ \vdash\ n_{\langle|\Gamma|\rangle}[]\ \in\ \mathbb{N}\ \longrightarrow$

$[step]\ \ \langle\Gamma\rangle;\ x:\ dom\{BT\{n\}\};\ \langle\Delta[mk\{x\}]\rangle\ \vdash\ P[mk\{x\}]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ BT\{n\ +\ 1\};\ \langle\Delta[t]\rangle\ \vdash\ P[t]$

$*[3,382]$ **rule** `bterm_elim_squash1` $\{|$ **public** elim $[]\ |\}\ \Gamma:$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N}\vdash\ \big[P[\texttt{var}(l,r)]\big]\ \longrightarrow$

1. $\langle\Gamma\rangle$

2. $t:$ **BTerm**

3. $\langle\Delta[t]\rangle$

4. $depth:\ \mathbb{N}$

5. $op:\ Operator$

6. $subterms:$ **BTerm** $List$

7. $\texttt{compatible_shapes}(depth; shape(op); subterms)$

$\vdash\ \big[P[BTerm(depth,\ op,\ subterms)]\big]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle\ \vdash\ \big[P[t]\big]$

$*[3,101]$ **rule** `bterm_elim2` {| **public** elim [] |} Γ:

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N}\vdash\ P[\texttt{var}(l,r)]\ \longrightarrow$

1. $\langle\Gamma\rangle$

2. $t:$ **BTerm**

3. $\langle\Delta[t]\rangle$

4. $bdepth:\ \mathbb{N}$

5. $op:\ Operator$

6. $subterms:$ **BTerm** $List$

7. $\texttt{compatible_shapes}(bdepth; shape(op); subterms)$

$\vdash\ P[BTerm(bdepth,\ op,\ subterms)]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle\ \vdash\ P[t]$

$*[5,122]$ **rule** `bterm_elim3` Γ:

$\langle\Gamma\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N};\ \langle\Delta[\texttt{var}(l,r)]\rangle\ \vdash\ P[\texttt{var}(l,r)]\ \longrightarrow$

1. $\langle\Gamma\rangle$

2. $bdepth:\ \mathbb{N}$

3. $op:\ Operator$

4. $subterms:$ **BTerm** $List$

5. $\texttt{compatible_shapes}(bdepth; shape(op); subterms)$

6. $\langle\Delta[BTerm(bdepth,\ op,\ subterms)]\rangle$

$\vdash\ P[BTerm(bdepth,\ op,\ subterms)]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle\ \vdash\ P[t]$

The following is the actual induction principle (the previous rules are just elimination rules).

$*[6,261]$ **rule** `bterm_induction` {| **public** elim [] |} Γ:

[*base*] $\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle;\ l:\ \mathbb{N};\ r:\ \mathbb{N}\ \vdash\ P[\mathtt{var}(l,r)]\ \longrightarrow$

[*step*]

1. $\langle\Gamma\rangle$
2. $t:\ \mathbf{BTerm}$
3. $\langle\Delta[t]\rangle$
4. $bdepth:\ \mathbb{N}$
5. $op:\ Operator$
6. $subterms:\ \mathbf{BTerm}\ List$
7. $\mathtt{compatible_shapes}(bdepth; shape(op); subterms)$
8. $\forall t \in subterms.\ P[t]$

$\vdash\ P[BTerm(bdepth,\ op,\ subterms)]\ \longrightarrow$

$\langle\Gamma\rangle;\ t:\ \mathbf{BTerm};\ \langle\Delta[t]\rangle\ \vdash\ P[t]$

$*[1,57]$ **rule** `is_var_wf` {| **public** intro [] |} :

[*wf*] $\langle\Gamma\rangle\ \vdash\ t\ \in\ \mathbf{BTerm}\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \mathtt{is_var}(t)\ \in\ \mathbb{B}$

$*[2,123]$ **rule** `subterms_wf1` {| **public** intro [] |} :

[*wf*] $\langle\Gamma\rangle\ \vdash\ t\ \in\ \mathbf{BTerm}\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \neg(\uparrow \mathtt{is_var}(t))\ \longrightarrow$

$\langle\Gamma\rangle\ \vdash\ \mathtt{subterms}\ t\ \in\ \mathbf{BTerm}\ List$

BTerm has a trivial squiggle equality.

$*[18,455]$ **rule** `bterm_sqsimple`

{| **public** intro []; **public** sqsimple |} :

$\langle\Gamma\rangle \vdash$ $sqsimple\{\mathbf{BTerm}\}$

$*[2, 40]$ **rule** `bterm_sqsimple2`

{| **public** intro []; **public** sqsimple |} :

$[wf]$ $\langle\Gamma\rangle \vdash$ $n \in \mathbb{N} \longrightarrow$

$\langle\Gamma\rangle \vdash$ $sqsimple\{BTerm\{n\}\}$

Define a Boolean equality (alpha equality) on BTerms.

define `unfold_beq_bterm` :

`Itt_hoas_bterm!beq_bterm{'t1<|!!|>; 't2<|!!|>}`

(displayed as "$beq_bterm\{t_1;\ t_2\}$"**)** $\longleftrightarrow$

$fix(beq_bterm.\lambda t_1.\lambda t_2.$**match** t_1 **with**

`var`$(l_1, r_1) -> $ **match** t_2 **with**

`var`$(l_2, r_2) -> $ $beq_var\{$`var`$(l_1, r_1);$ `var`$(l_2, r_2)\}$

$|$ $BTerm(d_1,\ o_1,\ s_1) -> $ $false$

$|$ $BTerm(d_1,\ o_1,\ s_1) -> $ **match** t_2 **with**

`var`$(l_2, r_2) -> $ $false$

$|$ $BTerm(d_2,\ o_2,\ s_2) -> $ $(d_1\ =_b\ d_2)$

$\wedge_b$ $is_same_op(o_1;\ o_2)$

$\wedge_b$ $(\forall_b t_1,\ t_2\ \in\ s_1,\ s_2.\ (beq_bterm\ t_1\ t_2)))\ t_1\ t_2$

$*[1, 22]$ **rewrite** `reduce_beq_bterm_var_var`

{| **public** reduce |} :

$l_1 \in \mathbb{N} \longrightarrow$

$r_1 \in \mathbb{N} \longrightarrow$

$l_2 \in \mathbb{N} \longrightarrow$

$r_2 \in \mathbb{N} \longrightarrow$

$beq_bterm\{$`var`$(l_1, r_1);$ `var`$(l_2, r_2)\} \longleftrightarrow$

$beq_var\{$`var`$(l_1, r_1);$ `var`$(l_2, r_2)\}$

$*[1, 55]$ **rewrite** `reduce_beq_bterm_var_bterm`

{| **public** reduce |} :

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$d \in \mathbb{N} \longrightarrow$

$o \in Operator \longrightarrow$

$s \in List \longrightarrow$

$beq_bterm\{\texttt{var}(l, r);\ BTerm(d,\ o,\ s)\} \longleftrightarrow false$

*[1, 38] **rewrite** `reduce_beq_bterm_bterm_var`

{| **public** reduce |} :

$l \in \mathbb{N} \longrightarrow$

$r \in \mathbb{N} \longrightarrow$

$d \in \mathbb{N} \longrightarrow$

$o \in Operator \longrightarrow$

$s \in List \longrightarrow$

$beq_bterm\{BTerm(d,\ o,\ s);\ \texttt{var}(l, r)\} \longleftrightarrow false$

*[1, 35] **rewrite** `reduce_beq_bterm_bterm_bterm`

{| **public** reduce |} :

$d_1 \in \mathbb{N} \longrightarrow$

$o_1 \in Operator \longrightarrow$

$s_1 \in \textbf{BTerm}\ List \longrightarrow$

$d_2 \in \mathbb{N} \longrightarrow$

$o_2 \in Operator \longrightarrow$

$s_2 \in \textbf{BTerm}\ List \longrightarrow$

$\texttt{compatible_shapes}(d_1; shape(o_1); s_1) \longrightarrow$

$\texttt{compatible_shapes}(d_2; shape(o_2); s_2) \longrightarrow$

$beq_bterm\{BTerm(d_1,\ o_1,\ s_1);\ BTerm(d_2,\ o_2,\ s_2)\} \longleftrightarrow$

$(d_1 =_b d_2)$

$\wedge_b\ is_same_op(o_1;\ o_2)$

$\wedge_b\ (\forall_b t_1,\ t_2 \in s_1,\ s_2.\ beq_bterm\{t_1;\ t_2\})$

*[10, 326] **rule** `beq_bterm_wf` {| **public** intro [] |} :

$[wf]\ \langle\Gamma\rangle \vdash t_1 \in \textbf{BTerm} \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash t_2 \in$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash$ $beq_bterm\{t_1;\ t_2\} \in \mathbb{B}$

$*[6, 152]$ **rule** `beq_bterm_intro` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash t_1 = t_2 \in$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle \vdash \uparrow beq_bterm\{t_1;\ t_2\}$

$*[24, 791]$ **rule** `beq_bterm_elim` {| **public** elim [] |} Γ:

$[wf]$ $\langle\Gamma\rangle;\ u : \uparrow beq_bterm\{t_1;\ t_2\};\ \langle\Delta[u]\rangle \vdash t_{1\langle|\Gamma|\rangle}[] \in$ **BTerm** $\longrightarrow$

$[wf]$ $\langle\Gamma\rangle;\ u : \uparrow beq_bterm\{t_1;\ t_2\};\ \langle\Delta[u]\rangle \vdash t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle;\ u :\ t_1 = t_2 \in$ **BTerm**; $\langle\Delta[u]\rangle \vdash C[u] \longrightarrow$

$\langle\Gamma\rangle;\ u : \uparrow beq_bterm\{t_1;\ t_2\};\ \langle\Delta[u]\rangle \vdash C[u]$

define `unfold_beq_bterm_list` :

`Itt_hoas_bterm!beq_bterm_list{'l1<|!!|>; 'l2<|!!|>}`

(displayed as "$beq_bterm_list\{l_1;\ l_2\}$"**)** $\longleftrightarrow$

$\forall_b t_1,\ t_2 \in l_1,\ l_2.\ beq_bterm\{t_1;\ t_2\}$

$*[1, 8]$ **rewrite** `reduce_beq_bterm_list_nil_nil`

{| **public** reduce |} :

$beq_bterm_list\{[];\ []\} \longleftrightarrow true$

$*[1, 8]$ **rewrite** `reduce_beq_bterm_list_nil_cons`

{| **public** reduce |} :

$beq_bterm_list\{[];\ u :: v\} \longleftrightarrow false$

$*[1, 8]$ **rewrite** `reduce_beq_bterm_list_cons_nil`

{| **public** reduce |} :

$beq_bterm_list\{u :: v;\ []\} \longleftrightarrow false$

$*[1, 18]$ **rewrite** `reduce_beq_bterm_list_cons_cons`

{| **public** reduce |} :

$beq_bterm_list\{u_1 :: v_1;\ u_2 :: v_2\}$

$\longleftrightarrow$

$beq_bterm\{u_1;\ u_2\} \wedge_b beq_bterm_list\{v_1;\ v_2\}$

$*[2, 15]$ **rule** `beq_bterm_list_wf` {| **public** intro [] |} :

$[wf]$ $\langle\Gamma\rangle \vdash l_1 \in$ **BTerm** $List \longrightarrow$

$[wf]$ $\langle\Gamma\rangle \vdash l_2 \in$ **BTerm** *List* $\longrightarrow$

$\langle\Gamma\rangle \vdash beq_bterm_list\{l_1;\ l_2\} \in \mathbb{B}$

*[2, 173] **rule** `beq_bterm_list_intro` {| **public** intro [] |} :

$\langle\Gamma\rangle \vdash t_1 = t_2 \in$ **BTerm** *List* $\longrightarrow$

$\langle\Gamma\rangle \vdash \uparrow beq_bterm_list\{t_1;\ t_2\}$

*[6, 213] **rule** `beq_bterm_list_elim` {| **public** elim [] |} Γ:

$[wf]$

1. $\langle\Gamma\rangle$
2. $u : \uparrow beq_bterm_list\{t_1;\ t_2\}$
3. $\langle\Delta[u]\rangle$

$\vdash t_{1\langle|\Gamma|\rangle}[] \in$ **BTerm** *List* $\longrightarrow$

$[wf]$

1. $\langle\Gamma\rangle$
2. $u : \uparrow beq_bterm_list\{t_1;\ t_2\}$
3. $\langle\Delta[u]\rangle$

$\vdash t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** *List* $\longrightarrow$

$\langle\Gamma\rangle;\ u :\ t_1 = t_2 \in$ **BTerm** *List*; $\langle\Delta[u]\rangle \vdash C[u] \longrightarrow$

$\langle\Gamma\rangle;\ u : \uparrow beq_bterm_list\{t_1;\ t_2\};\ \langle\Delta[u]\rangle \vdash C[u]$

Simple rules for forward chaining.

*[1, 34] **rule** `beq_bterm_forward` {| **public** forward |} Γ:

$[wf]$ $\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle \vdash t_{1\langle|\Gamma|\rangle}[] \in$ **BTerm** $\longrightarrow$

$[wf]$ $\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle \vdash t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** $\longrightarrow$

$\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle;\ t_{1\langle|\Gamma|\rangle}[] = t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** $\vdash C[\cdot] \longrightarrow$

$\langle\Gamma\rangle;\ x : \uparrow beq_bterm\{t_1;\ t_2\};\ \langle\Delta[x]\rangle \vdash C[x]$

*[1, 34] **rule** `beq_bterm_list_forward` {| **public** forward |} Γ:

$[wf]$ $\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle \vdash t_{1\langle|\Gamma|\rangle}[] \in$ **BTerm** *List* $\longrightarrow$

$[wf]$ $\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle \vdash t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** *List* $\longrightarrow$

$\langle\Gamma\rangle;\ \langle\Delta[\cdot]\rangle;\ t_{1\langle|\Gamma|\rangle}[] = t_{2\langle|\Gamma|\rangle}[] \in$ **BTerm** *List* $\vdash C[\cdot] \longrightarrow$

$\langle\Gamma\rangle\,;\; x : \uparrow beq_bterm_list\{t_1;\; t_2\};\; \langle\Delta[x]\rangle \;\vdash\; C[x]$

Equality reasoning.

$*[2,103]$ **rule** `mk_bterm_simple_eq` $\{|$ **public** intro $[]\ |\}$:

$[wf]\;\langle\Gamma\rangle \vdash d_1 = d_2 \in \mathbb{N} \longrightarrow$

$[wf]\;\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$

$[wf]\;\langle\Gamma\rangle \vdash subterms_1 = subterms_2 \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash \texttt{compatible_shapes}(d_1; shape(op_1); subterms_1) \longrightarrow$

1. $\langle\Gamma\rangle$

$\vdash$

$BTerm(d_1,\ op_1,\ subterms_1)$

$= BTerm(d_2,\ op_2,\ subterms_2) \in \mathbf{BTerm}$

$*[4,113]$ **rule** `mk_bterm_eq` $\{|$ **public** intro $[]\ |\}$:

$[wf]\;\langle\Gamma\rangle \vdash d_1 = d_3 \in \mathbb{N} \longrightarrow$

$[wf]\;\langle\Gamma\rangle \vdash d_2 = d_3 \in \mathbb{N} \longrightarrow$

$[wf]\;\langle\Gamma\rangle \vdash op_1 = op_2 \in Operator \longrightarrow$

$[wf]\;\langle\Gamma\rangle \vdash subterms_1 = subterms_2 \in \mathbf{BTerm}\ List \longrightarrow$

$\langle\Gamma\rangle \vdash \texttt{compatible_shapes}(d_1; shape(op_1); subterms_1) \longrightarrow$

1. $\langle\Gamma\rangle$

$\vdash$

$BTerm(d_1,\ op_1,\ subterms_1)$

$= BTerm(d_2,\ op_2,\ subterms_2) \in BTerm\{d_3\}$

$*[2,22]$ **rule** `bterm_depth_eq` $\{|$ **public** nth_hyp $|\}$:

$\langle\Gamma\rangle \vdash t \in BTerm\{d\} \longrightarrow$

$\langle\Gamma\rangle \vdash d = \texttt{D}\ t \in int$

$*[1,391]$ **rule** `bterm_depth_ge` $\{|$ **public** nth_hyp $|\}$:

$\langle\Gamma\rangle \vdash t \in BTerm\{d\} \longrightarrow$

$\langle\Gamma\rangle \;\vdash\; (\texttt{D}\ t) \geq d$