Using the Adelfa Proof Assistant to Construct Proofs of Programming Language Properties

Undergraduate Honors Thesis

Daniel Luick

May 2021

2

0.1 Acknowledgements

I would like to thank my faculty advisor, Gopalan Nadathur, for taking the time to carefully explain the topics I needed to learn to finish this thesis, and for helping me with my thesis, my research, and my career in general.

I would also like to thank Mary Southern, the creator of the proof assistant used by this honors thesis, for additionally helping me with research while writing her Ph.D. thesis.

I would also like to thank Professor Favonia, Professor Eric Van Wyk and Professor Nick Hopper for agreeing to serve on my honors thesis committee.

0.2 Abstract

In this thesis, we demonstrate stating and proving properties of a programming language using a dependently typed lambda calculus called LF and a system called Adelfa which provides mechanized support for reasoning about statements concerning typing derivations in LF. Proving properties in this manner allows the proofs to be undertaken using a formal logic, and builds greater trust in the proofs because the details of the steps are checked mechanically. The property that we consider in our demonstration is subject reduction for the Simply Typed Lambda Calculus. The Simply Typed Lambda Calculus is the theoretical foundation for many important programming languages and more complex lambda calculi, and subject reduction is a nontrivial properties with important equivalents in these more complex systems. Therefore, this proof constitutes a nontrivial demonstration of the usefulness of LF and Adelfa for proving properties of programming languages.

Contents

Chapter 1

Introduction

It is often useful to prove properties of programming languages. However, proofs of properties of programming languages are often difficult: in addition to requiring complex thinking, they are tedious and error prone, requiring many small details to be just right in order for the proof to hold. Formalizing these proofs can help, as this may allow for the proof to be automatically checked for correctness, or written within a proof assistant, which can take care of some of the bookkeeping aspects of the proof.

One method for formalizing proofs involves LF. LF is a dependently typed lambda calculus which allows for the encoding of rule based systems as specifications. In these specifications, relations between objects of the object system can be encoded as a family of types dependent on such objects, and elements of those dependent types represent proofs that specific relations hold. It is then possible to automatically type check an LF object, which allows for automatically checking if a relation holds.

While being able to automatically check if relations between objects hold is useful, it is also often desirable to state properties about relations within the object system, often making use of typical logic constructions such as quantification, implication, conjunction and disjunction. In her PhD thesis [Sou21], Mary Southern describes a logic containing such constructions for stating theorems about specifications encoded in LF, along with a system for proving these properties. She also describes a proof assistant, Adelfa, which implements this logic, and so allows for stating, proving, and automatically checking theorems about specifications encoded in LF.

The goal of this thesis is to demonstrate the usefulness of Adelfa in proving properties of programming languages. To this end, we consider the encoding in Adelfa of the proofs of properties of the Simply Typed Lambda Calculus (the STLC). The STLC is the theoretical foundation for many practical programming languages and more complicated lambda calculi. It is simple enough that proofs of its properties are relatively simple, but complex enough that these properties often translate to useful properties of more complicated languages, and allows the STLC to serve as a useful test of a proof assistant's capabilities. We consider specifically the proof of a property of the STLC that is known as subject reduction. This property states that evaluation does not change a term's type. This is an example of a useful property of a programming language, which translates into a useful equivalent property for more practical languages, and which provides a useful test of the proof assistant.

In Chapter 2 of the thesis, we introduces LF, including its syntax, typing rules and meta-theoretic properties, along with a demonstration of encoding two rule based systems as LF specifications and how properties proven of LF specifications can correspond to proofs about the original rule based system. Chapter 3 presents the logic for reasoning about LF specifications that underlies

Adelfa, and then shows how Adelfa can be used to construct proofs using a simple example concerning computations on lists. Chapter 4 then introduces subject reduction, proves it informally, and formalizes the proof in Adelfa.

Chapter 2

The Edinburgh Logical Framework

This chapter introduces the Edinburgh Logical Framework, known more simply as LF. LF is a dependently typed lambda calculus capable of formalizing rule based systems. In particular, it provides a means through lambda abstractions and contexts for generalizing the notion of binding. The original version [HHP93] allowed a large class of terms, which were then grouped together as equivalence classes of an equality relation. The version of LF used in this thesis, Canonical LF (hereafter just LF), only admits canonical terms.

This chapter begins with an introduction of LF: its syntax, typing rules and several metatheorems. It then demonstrates how rule based systems can be encoded in LF and how proofs about LF encodings can correspond to proofs about the rule based systems themselves. In particular, it introduces an LF encoding of the append relation for lists and the STLC.

2.1 Canonical LF

2.1.1 Syntax

Kinds	K	::=	$\mathrm{Type} \mid \Pi x{:}A.\,K$
Canonical Type Families	A	::=	$P \mid \Pi x{:}A_1.\,A_2$
Atomic Type Families	P	::=	$a \mid P\,M$
Canonical Terms	M	::=	$R \mid \lambda x.\,M$
Atomic Terms	R	::=	$c \mid x \mid R\,M$
Signatures	Σ	::=	$\cdot \mid \Sigma, c : A \mid \Sigma, a : K$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$

Figure 2.1: The Syntax of LF Expressions

The syntax of LF [HL07] is shown in Figure 2.1. LF has three categories of expressions: terms, types which classify terms, and kinds which classify types. Terms include the typical lambda calculus operations of abstraction and application along with variables and constants. Types include a binding operator written as $\Pi x{:}A_1.\,A_2$. This type can be thought of as similar to the typical lambda calculus arrow type, $A_1 \to A_2$, which is the type of functions which take a term of type A_1 and return a term of type A_2. However, unlike in the STLC, terms can appear in LF types, allowing for the construction of types that are *dependent* on terms. In the type $\Pi x{:}A_1.\,A_2$, x may

$\boxed{\vdash \Sigma \; \texttt{sig}}$

$$\frac{}{\vdash \cdot \; \texttt{sig}} \; SIG\_EMPTY$$

$$\frac{\vdash \Sigma \; \texttt{sig} \qquad \cdot \vdash_\Sigma A \; \texttt{type} \qquad c\#\Sigma}{\vdash \Sigma, c : A \; \texttt{sig}} \; SIG\_TERM$$

$$\frac{\vdash \Sigma \; \texttt{sig} \qquad \cdot \vdash_\Sigma K \; \texttt{kind} \qquad a\#\Sigma}{\vdash \Sigma, a : K \; \texttt{sig}} \; SIG\_FAM$$

$\boxed{\vdash_\Sigma \Gamma \; \texttt{ctx}}$

$$\frac{}{\vdash_\Sigma \cdot \; \texttt{ctx}} \; CTX\_EMPTY$$

$$\frac{\vdash_\Sigma \Gamma \; \texttt{ctx} \qquad \Gamma \vdash_\Sigma A \; \texttt{type} \qquad x\#\Gamma}{\vdash_\Sigma \Gamma, x : A \; \texttt{ctx}} \; CTX\_TERM$$

Figure 2.2: The Typing Rules for LF Signatures and Contexts

appear in A_2, making this a function type in which the type of the output may depend on what exactly is supplied as the argument. Types also include constants and applications of types to terms. We will also make use of the syntax $A_1 \rightarrow A_2$ as a shorthand for $\Pi x{:}A_1.\, A_2$ when x does not appear in A_2. Kinds include a distinguished $\texttt{type}$ used to classify base types, along with a similar Π abstraction used to classify dependent types.

The binding operators Π and λ introduce the usual notions of scope and free or bound variables. We also assume the principle of α-conversion: two expressions are considered equal if they differ only in the choices of the names for bound variables.

The syntax given divides terms between canonical terms and atomic terms, and types between canonical types and atomic types. The division in types becomes useful later during typing judgements, to ensure terms are η-long. The division in terms ensures that terms formed using this syntax never have a β-redex, as the first term of an application can never be a lambda abstraction.

2.1.2   Typing Rules

We can formalize the relationship between terms and types, and types and kinds, by using five different kinds of judgements, each defined by several inference rules. These rules make use of contexts, a list of the variables that may appear free in an expression along with the type of that variable, and signatures, which define the base types and terms which are used as constants.

- $\vdash \Sigma \; \texttt{sig}$ shows that Σ is a valid signature

- $\vdash_\Sigma \Gamma \; \texttt{ctx}$ shows that Γ is a valid context using signature Σ

- $\Gamma \vdash_\Sigma K \; \texttt{kind}$ shows that K is a well formed kind using context Γ and Σ

- $\Gamma \vdash_\Sigma A \; \texttt{type}$ and $\Gamma \vdash_\Sigma P \Rightarrow K$ show that A and P are well formed canonical and atomic types, respectively.

$\boxed{\Gamma \vdash_\Sigma K \text{ kind}}$

$$\frac{}{\Gamma \vdash_\Sigma \text{Type kind}} \; \mathit{CANON\_KIND\_TYPE}$$

$$\frac{\Gamma \vdash_\Sigma A \text{ type} \qquad \Gamma, x : A \vdash_\Sigma K \text{ kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.\, K \text{ kind}} \; \mathit{CANON\_KIND\_PI}$$

$\boxed{\Gamma \vdash_\Sigma A \text{ type}}$

$$\frac{\Gamma \vdash_\Sigma P \Rightarrow \text{Type}}{\Gamma \vdash_\Sigma P \text{ type}} \; \mathit{CANON\_FAM\_ATOM}$$

$$\frac{\Gamma \vdash_\Sigma A_1 \text{ type} \qquad \Gamma, x : A_1 \vdash_\Sigma A_2 \text{ type}}{\Gamma \vdash_\Sigma \Pi x{:}A_1.\, A_2 \text{ type}} \; \mathit{CANON\_FAM\_PI}$$

$\boxed{\Gamma \vdash_\Sigma P \Rightarrow K}$

$$\frac{a : K \in \Sigma}{\Gamma \vdash_\Sigma a \Rightarrow K} \; \mathit{ATOM\_FAM\_CONST}$$

$$\frac{\Gamma \vdash_\Sigma P \Rightarrow \Pi x{:}A.\, K_1 \qquad \Gamma \vdash_\Sigma M \Leftarrow A \qquad K_1 [\![\{ \langle x, M, (A)^- \rangle \}]\!] = K}{\Gamma \vdash_\Sigma P \; M \Rightarrow K} \; \mathit{ATOM\_FAM\_APP}$$

$\boxed{\Gamma \vdash_\Sigma M \Leftarrow A}$

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow P}{\Gamma \vdash_\Sigma R \Leftarrow P} \; \mathit{CANON\_TERM\_ATOM} \qquad \frac{\Gamma, x : A_1 \vdash_\Sigma M \Leftarrow A_2}{\Gamma \vdash_\Sigma \lambda x.\, M \Leftarrow \Pi x{:}A_1.\, A_2} \; \mathit{CANON\_TERM\_LAM}$$

$\boxed{\Gamma \vdash_\Sigma R \Rightarrow A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x \Rightarrow A} \; \mathit{ATOM\_TERM\_VAR} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash_\Sigma c \Rightarrow A} \; \mathit{ATOM\_TERM\_CONST}$$

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow \Pi x{:}A_1.\, A_2 \qquad \Gamma \vdash_\Sigma M \Leftarrow A_1 \qquad A_2 [\![\{ \langle x, M, (A_1)^- \rangle \}]\!] = A}{\Gamma \vdash_\Sigma R \; M \Rightarrow A} \; \mathit{ATOM\_TERM\_APP}$$

Figure 2.3: The Typing Rules for LF Kinds, Types, and Terms

- $\Gamma \vdash_\Sigma M \Leftarrow A$ and $\Gamma \vdash_\Sigma R \Rightarrow A$ show that M and R are well formed canonical and atomic types, respectively.

The inference rules for deriving these judgements are given in Figures 2.2 and 2.3. To be well-formed, a signature must assign types or kinds to distinct constants. Similarly, a well-formed context must assign types to distinct variables. Use is made of the notation $c\#\Sigma$, $a\#\Sigma$, and $x\#\Gamma$ in the rules towards this end; this notation means that c and a are *fresh* to Σ and x is similarly fresh to Γ. All the other judgements require that the signature and context in use is valid. In addition, $\Gamma \vdash_\Sigma P \Rightarrow K$ requires K to be a well formed kind, and $\Gamma \vdash_\Sigma M \Leftarrow A$ and $\Gamma \vdash_\Sigma R \Rightarrow A$ require A to be a well formed type. For the rules to be coherent, it must be the case that in deriving a typing judgement that satisfies these requirements we only need to use further typing judgements that also satisfy the requirements. It can be verified that this property in fact holds for the rules in Figures 2.2 and 2.3.

The rules for typing the application of a type to a term and the application of a term to a term requires us to consider the substitution of a term into a kind and a type, respectively. The canonical nature of the calculus requires any such substitution to be accompanied by a conversion of the resulting term into a canonical form. Use is made towards this end of the notion of *hereditary substitution*. To ensure that the conversion process terminates, substitutions are indexed by a special kind of types called *arity types* that, intuitively, determine the number of arguments that terms possessing them will take. Formally, these types are constructed from the constant type o using the function type constructor $\to$. These types can be viewed as a coarsened form of dependent types, identified by $(A)^-$ for each dependent type A and given as follows: $(P)^- = o$ and $(\Pi x{:}A_1.\,A_2)^- = (A_1)^- \to (A_2)^-$. Substitutions are now defined to be finite collections of triples of the form $\{x, M, \alpha\}$, where x is a variable, M is a canonical term and α is an arity type. If θ is a substitution and E is an expression into when we need to make this substitution, we denote the result of applying such a substitution using the expression $E[\![\theta]\!]$. The types indexing the substitution are used to guide the application in such a way that the process is guaranteed to terminate. While it is not guaranteed that the application will always have a result, the constraints accompanying the typing judgements we consider will ensure this to be the case for the typing rules in Figure 2.3. The precise definition of the application of *hereditary substitutions* and the observations about their use in the typing rules may be found in [Sou21].

2.1.3 Metatheoretic Properties

There are several properties of LF which arise as consequences of the typing rules which are useful in constructing arguments about typing judgements. Properties of this kind, which hold of *all* LF judgements irrespective of their specific content, are referred to as *metatheoretic properties* of LF. We present three such properties below that we will refer to later in this thesis. The proofs of these properties can be found in [HL07].

Theorem 2.1 (Instantiation). *If $\Gamma_1, x : A, \Gamma_2 \vdash_\Sigma M_2 \Leftarrow A_2$, $\Gamma_1 \vdash_\Sigma M \Leftarrow A$ and $M_2[\![\{x, M\}]\!] = M_2'$, $\Gamma_2[\![\{x, M\}]\!] = \Gamma_2'$, and $A_2[\![\{x, M\}]\!] = A_2'$, then $\Gamma_1, \Gamma_2' \vdash_\Sigma M_2' \Leftarrow A_2'$.*

Theorem 2.2 (Weakening). *Assume $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.*

1. *For all five formation judgements J, if $\Gamma \vdash_\Sigma J$ then $\Gamma' \vdash_{\Sigma'} J$.*

2. *If $\vdash_\Sigma \Gamma$ ctx then $\vdash_{\Sigma'} \Gamma'$ ctx.*

3. *If $\vdash \Sigma$ sig then $\vdash \Sigma'$ sig.*

$$\frac{}{nat\ z}\ \text{z-nat} \qquad\qquad \frac{nat\ N}{nat\ (s\ N)}\ \text{s-nat}$$

$$\frac{}{list\ nil}\ \text{nil-list} \qquad\qquad \frac{nat\ N \qquad list\ L}{list\ (cons\ N\ L)}\ \text{cons-list}$$

$$\frac{list\ L}{app\ nil\ L\ L}\ \text{app-nil} \qquad \frac{nat\ N \qquad app\ L_1\ L_2\ L_3}{app\ (cons\ N\ L_1)\ L_2\ (cons\ N\ L_3)}\ \text{app-cons}$$

Figure 2.4: Rules for Appending Lists

Theorem 2.3 (Permutation). *For all five formation judgements J, if $\Gamma, x_1 : A_1, x_2 : A_2, \Gamma' \vdash_\Sigma J$ and x_1 does not appear in A_2, then $\Gamma, x_2 : A_2, x_1 : A_1, \Gamma' \vdash_\Sigma J$*

2.2 Representing Rule Based Systems in LF

The primary reason for the interest in LF for this thesis is its ability to encode rule based systems. Rule based systems are systems which define relations of interest based on rules. In this section, we demonstrate two different LF encodings. We also discuss how to be sure that an encoding is correct, and how LF contexts can enable the usage of higher order abstract syntax to ease the creation of encodings.

2.2.1 First Order Specifications

As an first example of such encodings we consider the append relation on lists of natural numbers. To specify this relation we must first identify natural numbers as well as lists over such numbers. We do this in Figure 2.4 and then follow this up with a rule-based presentation of the append relation over such lists.

We can encode this collection of rules in the LF signature Σ_{app}, as seen in Figure 2.5. In this signature, natural numbers and lists are encoded as expressions whose types are given by two type-level constants *nat* and *list*. Each formation rule for a number or list corresponds to an LF constructor for that type.

Relations are encoded in LF using dependent types. Each relation becomes an LF type family indexed by the terms which could make up the relation. Append is a relation between three lists; therefore *append* is indexed by three lists and has the kind $\Pi L_1{:}\textit{list}.\ \Pi L_2{:}\textit{list}.\ \Pi L_3{:}\textit{list}.\ \textit{type}.$ Each rule then becomes a constructor, with the arguments corresponding to the prerequisites and the result type corresponding to the result of the inference rule. The *append-nil* and *append-cons* rules then become the signature constants *append-nil* and *append-cons*. Any type *append L_1 L_2 L_3* can then be viewed as the type of proofs that L_1 and L_2 appended together are L_3. In the rule based system, this would be proven by uses of *append-nil* and *append-cons* rules. In LF, a concrete proof would be a term of this type, constructed using the *append-nil* and *append-cons* constants.

2.2.2 Simply Typed Lambda Calculus

Another system we wish to encode is the Simply Typed Lambda Calculus (STLC). The STLC is the theoretical basis for many practical programming languages, and is the foundation for more complicated lambda calculi. It is complex enough that proofs of its properties contain many of the essential difficulties of proving these properties of more complicated languages, but are still

nat : *type*
z : *nat*
s : ΠN:*nat*. *nat*

list : *type*
nil : *list*
cons : ΠN:*nat*. ΠL:*list*. *list*

append : ΠL_1:*list*. ΠL_2:*list*. ΠL_3:*list*. *type*
append-nil : ΠL:*list*. *append nil L L*
append-cons : ΠN:*nat*. ΠL_1:*list*. ΠL_2:*list*. ΠL_3:*list*.
$\quad\quad$ ΠD:*append L_1 L_2 L_3*. *append (cons N L_1) L_2 (cons N L_3)*

Figure 2.5: Σ_{app}, LF signature for appending lists

relatively simple. Because of this, it is an ideal language for demonstrating proof of properties of languages.

$$\begin{aligned} \textbf{Terms} \quad & M, N \quad ::= \quad x \mid M\ N \mid \lambda x : T.M \\ \textbf{Types} \quad & T \quad ::= \quad unit \mid T_1 \to T_2 \\ \textbf{Contexts} \quad & \Gamma \quad ::= \quad \cdot \mid \Gamma, x : T \end{aligned}$$

Figure 2.6: The Syntax of STLC terms, types and contexts

The syntax of the STLC is given in figure 2.6. In this version of the STLC, types include an arrow type $T_1 \to T_2$ to represent functions from T_1 to T_2, along with a base type *unit* with no constructors. Terms include variables x, applications $(M\ N)$ and lambda abstractions $\lambda x : T.M$; note that parentheses can be inserted in expressions to disambiguate the scope of constructions, as is commonly understood. Contexts are lists of variables of various types. The same variable name cannot appear twice in a context.

The STLC also defines a typing judgement $\Gamma \vdash M : T$ to indicate that term M has type T in context Γ. The rules for this judgement are given in Figure 2.7.

We also define small step reduction for the STLC in Figure 2.8. Reduction in the STLC corresponds to evaluation in other programming languages. Beta reduction makes use of substitution,

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \ of\_var$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : T_1 \to T_2} \ of\_lam$$

$$\frac{\Gamma \vdash M_1 : T_1 \to T_2 \quad\quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T_2} \ of\_lam$$

Figure 2.7: The Typing Rules for the STLC

$$\frac{M_1 \Rightarrow M_3}{M_1 \ M_2 \Rightarrow M_3 \ M_2} \ step\_app1 \qquad\qquad \frac{M_2 \Rightarrow M_3}{M_1 \ M_2 \Rightarrow M_1 \ M_3} \ step\_app2$$

$$\frac{M_1 \Rightarrow M_2}{\lambda x : T.M_1 \Rightarrow \lambda x : T.M_2} \ step\_lam \qquad \frac{}{(\lambda x : T.M_1) \ M_2 \Rightarrow M_1[x/M_2]} \ step\_beta$$

Figure 2.8: The Reduction Rules for the STLC

which is defined as usual for lambda calculi.

The STLC can be encoded in LF as $\Sigma_{\mathtt{stlc}}$, given in Figure 2.9. An aspect to note about the STLC, in contrast to the language of lists considered in the previous section, is that expressions in this calculus embody a notion of binding. There are properties associated with binding, such as the fact that two expressions are considered equal if they differ only in the choice of names for bound variables and that substitutions into expressions must respect binding, that should be accounted for in the encoding that is used for this notion. The approach that is chosen in the encoding displayed in Figure 2.9 is based on the approach of *higher-order abstract syntax*. In this approach, binding in object expressions, i.e., in the expressions being represented, is encoded using binding in the meta-language, i.e. the language in which the representations are constructed. The proper treatment of binding in the meta-language then provides a simple and direct treatment of binding in the object language. This choice of representation is to be seen specifically in the way abstraction in STLC is encoded. The constructor *lam* that is used to encode such expressions takes as arguments the representation of an STLC type and a *function term* of type $tm \to tm$. Thus, the STLC expression $\lambda x : unit.x$ would be represented by the LF expression (*lam unit* $\lambda x. x$). One of the benefits of this representation is to be seen in the encoding of *step_beta*, the STLC rule for β-contraction; the substitution that is necessary in effecting this operation is realized transparently through application, and the associated β-conversion, in LF.

The higher-order treatment that we have just described means that we may sometimes have to consider typing judgements where the type in question may contain a Π operator. This is to be seen, for example in the encoding of the typing rule for abstractions in the STLC. For example, in determining if the STLC expression $\lambda x : unit.x$ has the type $unit \to unit$, we would need to construct an LF term that has the LF type (*of* (*lam unit* $\lambda x. x$) (*arr unit unit*)). This would, in turn, lead us to consider the inhabitation of the LF type $\Pi x{:}tm.\, \Pi d{:}of \ x \ unit.\, of \ ((\lambda y.\, y) \ x) \ unit$ and, eventually, to checking the existence of an LF term of type (*of x unit*) in the context $x : tm, d : of \ x \ unit$.

2.2.3 The Adequacy of Encodings

We want eventually to reason about object systems but we think of doing this through their encodings in LF. It is desirable in this context to to show that our LF signature correctly represents the object system. This is a property known as the *adequacy* of the encoding. If adequacy is present, then properties of the LF encoding, such as certain relations holding or theorems proven about the encoding, also hold for the encoded system. Proving adequacy requires establishing an isomorphism between the entities of the encoded system and its LF representation. While this is important to the overall enterprise, we will not actually provide proofs of adequacy in this thesis.

ty : *type*
unit : *ty*
arr : ΠT_1:*ty*. ΠT_2:*ty*. *ty*

tm : *type*
app : ΠY_1:*tm*. ΠY_2:*tm*. *tm*
lam : ΠZ:*ty*. ΠY:Πx:*tm*. *tm*. *tm*

of : ΠM:*tm*. ΠT:*ty*. *type*
of_app : ΠM:*tm*. ΠN:*tm*. ΠT:*ty*. ΠU:*ty*.
    Πa_1:*of M* (*arr U T*). Πa_2:*of N U*. *of* (*app M N*) *T*
of_lam : ΠR:Πx:*tm*. *tm*. ΠT:*ty*. ΠU:*ty*.
    Πa_1:(Πx:*tm*. Πd:*of x T*. *of* (*R x*) *U*). *of* (*lam T R*) (*arr T U*)

step : ΠM_1:*tm*. ΠM_2:*tm*. *type*
step-app1 : ΠM_1:*tm*. ΠM_2:*tm*. ΠN:*tm*. ΠD:*step M_1 M_2*.
    step (*app M_1 N*) (*app M_2 N*)
step-app2 : ΠM:*tm*. ΠN_1:*tm*. ΠN_2:*tm*. ΠD:*step N_1 N_2*.
    step (*app M N_1*) (*app M N_2*)
step-beta : ΠT:*ty*. ΠR:(Πx:*tm*. *tm*). ΠN:*tm*.
    step (*app* (*lam T R*) *N*) (*R N*)
step-lam : ΠT:*ty*. ΠR_1:(Πx:*tm*. *tm*). ΠR_2:(Πx:*tm*. *tm*).
    ΠD:Πx:*tm*. Πd:*x* : *T*. *step* (*R_1 x*) (*R_2 x*).
    step (*lam T R_1*) (*lam T R_2*)

Figure 2.9: $\Sigma_{\texttt{stlc}}$, LF signature for the STLC

2.3   Reasoning informally about LF specifications

It is often the case that we wish to prove some property of a rule based system. For instance, it is fairly obvious that appending a nil list to the end of another list results in the same list. This fact, however, does not immediately follow from our informal definition of append. However, this fact can be proven fairly easy by induction on the structure of the list in induction. The nil case follows from the *app-nil* rule, while the cons case follows from the inductive hypothesis and the *app-cons* rule.

We can prove the same property of the LF encoding. Due to adequacy, a theorem proven about an LF encoding will also hold of the original rule based system. This allows for the benefits of LF, such as being able to automatically check typing judgements, while still proving the theorem for the original system.

Theorem 2.4. *For all terms L, if $\cdot \vdash_{\Sigma_{app}} L : list$ then there exists a term D such that $\cdot \vdash_{\Sigma_{app}} D : append\ L\ nil\ L$ holds.*

Proof. By induction on the structure of L.

If L is *nil*, then *append-nil nil* has type *append nil nil nil*.

If L is *cons N L′*, then we know that $\cdot \vdash_{\Sigma_{app}} N : nat$ and $\cdot \vdash_{\Sigma_{app}} L′ : list$. By the inductive hypothesis, there exists a $D′$ such that $\cdot \vdash_{\Sigma_{app}} D′ : append\ L′\ nil\ L′$. We then know that (*append-cons L′ nil L′ N D′*) has type (*append* (*cons N L′*) *nil cons N L′*) which is a suitable term for $D′$.

□

Chapter 3

The Adelfa System

This chapter provides a brief introduction to the Adelfa system. Adelfa is a proof assistant that aids in the construction of proofs of properties about LF specifications [Sou21]. Underlying Adelfa is a logic for stating such properties and a proof system that supports the mechanization of arguments based on this logic. We describe the logic in the first section below. The next section sketches how statements in this logic can be shown to be true using Adelfa; while we do not discuss the proof system explicitly, its underlying characteristics will become clear in the course of the discussion of Adelfa.

3.1   The Logic Underlying Adelfa

The logic underlying Adelfa has been designed by Nadathur and Southern. A detailed presentation of it can be found in [Sou21]. We explain below the structure of this logic to the extent needed in this thesis.

    The starting point for the logic are LF types and terms. This logic is parameterized by a valid LF signature that can be used in constructing these expressions. LF terms and types appearing in formulas in the logic use the same syntax as in Section 2.1.1, with two exceptions. A new method of constructing valid atomic terms is through a nominal constant n. Nominal constants are used to represent variables in a context, and are treated much like other context for operations such as hereditary substitution. Additionally, terms and types can contain variables introduced by universal or existential quantifiers.

$$
\begin{array}{rcl}
\textbf{Block Declarations} \quad \Delta & ::= & \cdot \mid \Delta, y : A \\
\textbf{Block Schema} \quad B & ::= & \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta \\
\textbf{Context Schema} \quad C & ::= & \cdot \mid C, B
\end{array}
$$

Figure 3.1: The Syntax of Context Schema

    One new construction appearing in formulas is context schemas, which are used in universal quantification over contexts. Quantifying over *all* contexts is not particularly useful; it is better to quantify over contexts which match a certain pattern. A context schema specifies such a pattern. As an example, a useful context schema for Σ_{stlc} would restrict LF contexts such that they corresponded to STLC contexts. Any LF context which fit that criteria would consist of a sequence of two variables, the first having type *tm*, the second having type *of y x* where y is the previous variable of type *tm* and x is some STLC type. That is represented by the LF context schema $\{x : \alpha\}y_1 :$

$tm, y_2 : of\ y_1\ x$. A context schema consists of some number of block schemas; the example context schema is a single block schema. A block schema of a list of declarations associating variables with types. It is preceded by a header listing variables, indexed by arity types, which may appear in the declaration. A context matching some context schema can be constructed by concatenating zero or more repetitions of any block schema, with the headers instantiated with terms of the correct arity.

$$\begin{array}{rcl}
\textbf{Context Expressions} \quad G & ::= & \cdot \mid \Gamma \mid G, n : A \\
\textbf{Formulas} \quad F & ::= & \{G \vdash M : A\} \mid \top \mid \bot \mid F_1 \supset F_2 \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \\
& & \Pi\,\Gamma : C.F \mid \forall x : \alpha.F \mid \exists x : \alpha.F
\end{array}$$

Figure 3.2: The Syntax of Formulas

The syntax for formulas in the logic is given in Figure 3.2. Formulas make use of contexts, which are specified as a list of nominal constants with specific types, possibly preceded by a context variable Γ. Atomic formulas, representing LF typing judgements of the form $\Gamma \vdash_\Sigma M \Leftarrow A$, have the form $\{G \vdash M : A\}$. Formulas also include typical logical constructs, such as $\top$ and $\bot$ as true and false $\supset$, $\wedge$ and $\vee$ for implication, conjunction and disjunction. $\forall$ and $\exists$ can be used for universal and existential quantification over LF terms, indexed by an arity type. Π is used for universal quantification over contexts restricted by context schemas.

Finally, we wish to define when a formula in the logic is valid. If a formula F is well formed according to the syntax, then its validity is defined by recursion on the formula structure as follows:

- $\top$ is valid, and $\bot$ is not.

- $\{G \vdash M : A\}$ is valid when $\vdash_\Sigma G\ \texttt{ctx}$, $G \vdash_\Sigma A\ \texttt{type}$ and $G \vdash_\Sigma M \Leftarrow A$ are provable in LF, with nominal constants interpreted as variables bound in a context.

- $F_1 \supset F_2$ is valid if, whenever F_1 is valid, F_2 is valid.

- $F_1 \wedge F_2$ is valid if F_1 and F_2 are valid.

- $F_1 \vee F_2$ is valid if either F_1 or F_2 are valid.

- $\Pi\,\Gamma : C.F$ is valid if, for any context G which matches the context schema C, F is valid when Γ is replaced by G.

- $\forall x : \alpha.F$ is valid if, for any term M of arity type α, $F[\![\{x, M, \alpha\}]\!]$ is valid.

- $\exists x : \alpha.F$ is valid if there exists a term M of arity type α such that $F[\![\{x, M, \alpha\}]\!]$ is valid.

It is good to show that validity is in fact useful in distinguishing between formulas, by demonstrating that there are in fact valid and invalid formulas. An example of a valid formula is given later in the chapter. An example of an invalid formula, using $\Sigma_{\texttt{stlc}}$, is $\forall d : \alpha. \{\cdot \vdash d : of\ (lam\ unit\ \lambda x.x)\ unit\}$. This formula is well formed, but there is no term for which this is a valid formula since a lambda expression cannot have type *unit*. Therefore this formula is not valid, and validity is therefore capable of distinguishing between formulas.

As an example of using this logic that does not include the use of context quantification to state a useful property, we can restate the theorem from Section 2.3 as follows:

$$\forall L : o. \{\cdot \vdash L : list\} \rightarrow \exists D : o. \{\cdot \vdash D : append\ L\ nil\ L\}$$

This formula is a sensible formalization of the theorem about LF derivability relative to Σ_{app} that we saw in Theorem 2.4. To understand this, we observe that, by the semantics of the logic, the formula is claiming that for every LF expression L that can be assigned the type *list* relative to the signature Σ_{app}, it is the case that the righthand side of the implication is true. However, that formula is true only if there is an inhabitant of the type *(append L nil L)*, which is what Theorem 2.4 requires us to show. In addition, since Theorem 2.4 is a formalization of the original property stated about the informal rule based system, the formula in this logic also formalizes the original property stated about the informal rule based system.

3.2 Proving Theorems in Adelfa

This section will show how the Adelfa system can be used to prove the theorem about append that was expressed in the formula at the end of the previous section.

Adelfa works with a specific LF signature. This signature must be written out in a text file with a `.elf` extension, using a syntax similar to that from Twelf [PS02]. A term or type constant declaration $c : A$ is written `c:A.`, ended with a period. The base kind, `type`, is written `type`. A lambda expression $\lambda x.\,M$ is written `[x] M`, and a pi expression $\Pi x{:}A_1.\,A_2$ is written `{x:A1} A2`. `%` begins a single line comment. Parentheses can be used for grouping. The append signature $\Sigma_{\mathtt{app}}$ is written as follows:

```
% append signature

nat : type.
z : nat.
s : {N:nat} nat.

list : type.
nil : list.
cons : {N:nat} {L:list} list.

append : {L1:list} {L2:list} {L3:list} type.
append-nil : {L:list} append nil L L.
append-cons : {N:nat} {L1:list} {L2:list} {L12:list} {D:append L1 L2 L12}
              append (cons N L1) L2 (cons N L12).
```

Adelfa is a command line program, and is run by issuing commands and tactics ending in a period. The first command, `Specification`, loads a signature file.

```
~/adelfa$ adelfa.byte
Welcome!
>> Specification "append.elf".
```

Beginning the proof of a theorem is done with the `Theorem` command, which is stated using a formula as defined in Section 3.1. The syntax uses `true`, `false`, `F1 /\ F2`, `F1 \/ F2`, `F1 => F2`,

`ctx G:c, F,` `forall M, F,` `exists M, F,` and `{G |- M : A}` to represent formulas in the logic. In addition, atomic formulas with an empty context can simply be stated as `{M : A}`. Also note that the `forall` and `exists` quantifiers can infer the arities of their quantifiers, and multiple quantifiers can be chained together as a space separated list, as in `forall M1 M2 M3, F`. The append theorem from Section 2.3, named `app-nil` here, is stated as follows.

```
>> Theorem app-nil : forall L, {L : list} => exists D, {D : append L nil L}.

Subgoal app-nil:


==================================
forall L, {L : list} => exists D, {D : append L nil L}
```

   Adelfa works by maintaining a proof state, which is a list of nominal constants that are used for term variables bound in the contexts of atomic formulas relevant to the proof state, a list of implicitly universally quantified term variables appearing in the proof state, a similar list of implicitly universally quantified context variables appearing in the proof state, a list of assumption formulas, and a goal formula that needs to be proved. There then exist proof rules which can be used to move between proof states in a way that maintains the soundness of the system. Rather than use these proof rules directly, the command line program is controlled by issuing tactics, which change the state using one or more proof rules.

   The informal proof of *app-nil* uses induction on the structure of the list. It then immediately considers all of the cases in which $\cdot \vdash_\Sigma L \Leftarrow list$ could have been proven. In Adelfa, this can be accomplished with three tactics. The first is `induction`, which signals that we intend to use induction on a specific assumption in the goal formula. This adds an assumption which matches the goal formula, but where the inductive assumption must have a shorter derivation (marked with *) than the original assumption (marked with @). The second tactic, `intros`, introduces any assumptions in the goal formula as assumptions in the proof state. The third tactic, `case`, splits the proof into multiple subgoals, one for each case a specific assumption could have been proven. The results of issuing these commands is seen as follows.

```
app-nil>> induction on 1.

Subgoal app-nil:

IH:forall L, {L : list}* => exists D, {D : append L nil L}


==================================
forall L, {L : list}@ => exists D, {D : append L nil L}

app-nil>> intros.

Subgoal app-nil:

Vars: L:o
IH:forall L, {L : list}* => exists D, {D : append L nil L}
H1:{L : list}@
```

```
===================================
exists D, {D : append L nil L}


app-nil>> case H1.


Subgoal app-nil.1:


Vars: N:o, L1:o
IH:forall L, {L : list}* => exists D, {D : append L nil L}
H2:{N : nat}*
H3:{L1 : list}*


===================================
exists D, {D : append (cons N L1) nil (cons N L1)}


Subgoal app-nil.2 is:
 exists D, {D : append nil nil nil}
```

Due to the order in which the append relation was declared in Adelfa, we first wish to deal with the case where the list was formed using `cons`. In the informal proof, this case requires applying the inductive hypothesis and then supplying the right term using `append-cons` for an existential quantifier. We can translate this to Adelfa with three more tactics. `apply` applies one assumption formula (in this case, the inductive hypothesis) to one or more other assumption formulas. `exists` fills an existential quantifier in the goal formula with a given term. Finally, `search` enables some basic proof search to complete the subgoal.

```
app-nil.1>> apply IH to H3.
Matching formula {L1 : list}* to formula {?1 : list}*


Subgoal app-nil.1:


Vars: D:o, N:o, L1:o
IH:forall L, {L : list}* => exists D, {D : append L nil L}
H2:{N : nat}*
H3:{L1 : list}*
H4:{D : append L1 nil L1}


===================================
exists D, {D : append (cons N L1) nil (cons N L1)}


Subgoal app-nil.2 is:
 exists D, {D : append nil nil nil}


app-nil.1>> exists append-cons N L1 nil L1 D.


Subgoal app-nil.1:
```

```
Vars: D:o, N:o, L1:o
IH:forall L, {L : list}* => exists D, {D : append L nil L}
H2:{N : nat}*
H3:{L1 : list}*
H4:{D : append L1 nil L1}

=================================
{append-cons N L1 nil L1 D : append (cons N L1) nil (cons N L1)}

Subgoal app-nil.2 is:
 exists D, {D : append nil nil nil}

app-nil.1>> search.
Searching for derivation of:
{append-cons N L1 nil L1 D : append (cons N L1) nil (cons N L1)}
Searching for derivation of:
{N : nat}
Searching for derivation of:
{L1 : list}
Searching for derivation of:
{nil : list}
Searching for derivation of:
{L1 : list}
Searching for derivation of:
{D : append L1 nil L1}

Subgoal app-nil.2:

IH:forall L, {L : list}* => exists D, {D : append L nil L}

=================================
exists D, {D : append nil nil nil}
```

Finally, proving the nil case is trivial and can be done with tactics already introduced.

```
app-nil.2>> exists append-nil nil.

Subgoal app-nil.2:

IH:forall L, {L : list}* => exists D, {D : append L nil L}

=================================
{append-nil nil : append nil nil nil}

app-nil.2>> search.
Searching for derivation of:
{append-nil nil : append nil nil nil}
Searching for derivation of:
```

```
{nil : list}
Proof Completed!
```

Chapter 4

Subject Reduction

One useful property to prove of the STLC is subject reduction. Subject reduction for the STLC states that any term that is obtained by one step of evaluation from a given STLC term will have the same type as the original term. This property corresponds to type safety in more practical programming languages with the STLC as a theoretical foundation, which ensures a programming language will never break the rules of its type system, which helps prevent the language from illegally tampering with memory. Subject reduction is also a good property with which to test prospective proof systems, as it is relatively simple but makes use of concepts such as STLC contexts.

4.1 Informal Proof of Subject Reduction

To begin, we will prove subject reduction informally using the STLC as presented in Section 2.2.2. In the proof, we make use of the property that types are preserved in the STLC under substitution, a fact stated precisely in the following theorem.

Theorem 4.1 (Substitution). *If $\Gamma, x : T_1 \vdash M_2 : T_2$ and $\Gamma \vdash M_1 : T_1$, then $\Gamma \vdash M_2[M_1/x] : T_2$.*

This property can be proved by induction on the first typing judgement that translates effectively into an induction on the structure of M_2. We omit the details of the proof.

Theorem 4.2 (Subject Reduction of the STLC). *For all contexts Γ, terms M_1 and M_2 and types T, if $\Gamma \vdash M_1 : T$ and $M_1 \Rightarrow M_2$, then $\Gamma \vdash M_2 : T$.*

Proof. The proof is by induction on the derivation of the small-step evaluation relation, which effectively translates into an induction on the structure of M_1.

If $M_1 \Rightarrow M_2$ was proven using *step_lam*, we know T has the form $T_1 \to T_2$, M_1 has the form $\lambda x : T_1.M_3$, and M_2 has the form $\lambda x : T_1.M_4$, and that $M_3 \Rightarrow M_4$ is provable by a shorter derivation than that for $M_1 \Rightarrow M_2$. Since $\Gamma \vdash M_1 : T$ must have been proven using *step_lam* we know $\Gamma, x : T_1 \vdash M_3 : T_2$ holds. By the inductive hypothesis, we know $\Gamma, x : T_1 \vdash M_4 : T_2$. We can then use *of_lam* to show $\Gamma \vdash M_2 : T$.

If $M_1 \Rightarrow M_2$ was proven using *step_app1*, we know that M_1 has the form $M_3 \ M_4$, that M_2 is $M_5 \ M_4$, and that $M_3 \Rightarrow M_5$ is provable by a shorter derivation than that for $M_1 \Rightarrow M_2$. Since $\Gamma \vdash M_1 : T$ must have been proven using *of_app* we know that for some type T_2 $\Gamma \vdash M_3 : T_2 \to T$ and $\Gamma \vdash M_4 : T_2$. By the inductive hypothesis, we know that $\Gamma \vdash M_5 : T_2 \to T$. We can then use *of_app* to show $\Gamma \vdash M_2 : T$.

If $M_1 \Rightarrow M_2$ was proven using *step_app2*, we know that M_1 has the form $M_3 \ M_4$, that M_2 has the form $M_3 \ M_5$, and that $M_4 \Rightarrow M_5$ is provable by a shorter derivation than that for $M_1 \Rightarrow M_2$.

```
ty : type.
unit : ty.
arr : {Z1:ty} {Z2:ty} ty.


tm : type.
app : {Y1:tm} {Y2:tm} tm.
lam : {Z:ty} {Y:{x:tm}tm} tm.


of : tm -> ty -> type.
of_app : {M:tm}{N:tm}{T:ty}{U:ty}
          {a1:of M (arr U T)} {a2:of N U} of (app M N) T.
of_lam : {R : {x:tm} tm}{T:ty}{U:ty}
          {a1:({x:tm}{z:of x T} of (R x) U)}
          of (lam T R) (arr T U).


step : tm -> tm -> type.
step-app1 : {M1:tm} {M2:tm} {N:tm} {D : step M1 M2}
             step (app M1 N) (app M2 N).
step-app2 : {M:tm} {N1:tm} {N2:tm} {D : step N1 N2}
             step (app M N1) (app M N2).
step-beta : {T:ty} {R:{x:tm}tm} {N:tm}
             step (app (lam T R) N) (R N).
step-lam  : {T:ty} {R1:{x:tm}tm} {R2:{x:tm}tm}
             {D : {x:tm} {d:of x T} step (R1 x) (R2 x)}
             step (lam T R1) (lam T R2).
```

Figure 4.1: Adelfa encoding of $\Sigma_{\texttt{stlc}}$

Since $\Gamma \vdash M_1 : T$ must have been proven using *of_app* we know that for some type T_2 $\Gamma \vdash M_3 : T_2 \to T$ and $\Gamma \vdash M_4 : T_2$. By the inductive hypothesis, we know that $\Gamma \vdash M_5 : T_2$. We can then use *of_app* to show $\Gamma \vdash M_2 : T$.

If $M_1 \Rightarrow M_2$ was proven using *step_beta*, we know that M_1 has the form $(\lambda x : T_2 \to T.M_3) \, M_4$ and M_2 has the form $M_3[x/M_4]$. Since $\Gamma \vdash M_1 : T$ must have been proven using *of_app* we know that $\Gamma, x : T_2 \vdash M_3 : T$ and $\Gamma \vdash M_4 : T_2$ hold. We can then use substitution to show $\Gamma \vdash M_2 : T$. $\square$

4.2   Proving Subject Reduction in Adelfa

Next, we will prove subject reduction of the STLC in Adelfa. To begin, we need a `.elf` file to encode $\Sigma_{\texttt{stlc}}$. This is shown in Figure 4.1. One feature to note is that a lambda expression is encoded as a term of type `ty -> (tm -> tm) -> tm`. This allows the variable in an expression such as $\lambda x.x$ to be represented as an LF variable, making use of higher order abstract syntax. In addition, the STLC context is represented by the LF context, which is assumed to be restricted to a specific context schema as described in Section 3.1. This schema is named `c` and declared as follows.

```
>> Schema c := {T}(x:tm,y:of x T).
```

   Next, we declare the theorem. This follows much like declaring the append theorem in Section 3.1. The new addition is that we use context quantification to express the phrase the idea of quantifying over all STLC contexts.

```
>> Theorem subject_reduction : ctx Gamma:c, forall M1 M2 T D1 D2,
   {Gamma |- D1 : step M1 M2} => {Gamma |- D2 : of M1 T} => exists D3,
   {Gamma |- D3 : of M2 T}.
```

   Like the informal proof, we prove subject reduction by induction on the given derivation of the step judgement. We can use the `intros`, `induction` and `case` tactics to establish this, shown as follows. Note that some of the original text has been replaced with ... for brevity.

```
subject_reduction>> induction on 1.

Subgoal subject_reduction:
...

subject_reduction>>  intros.

Subgoal subject_reduction:
...

subject_reduction>>  case H1.

Subgoal subject_reduction.1:

Vars: D:(o) -> (o) -> o, R1:(o) -> o, T1:o, R2:(o) -> o, D2:o, T:o
Nominals: n3:o, n2:o, n1:o, n:o
Contexts: Gamma{n3, n2, n1, n}:c[]
IH:
    ctx Gamma:c.
      forall M1, forall M2, forall T, forall D1, forall D2,
        {Gamma |- D1 : step M1 M2}* =>
            {Gamma |- D2 : of M1 T} => exists D3, {Gamma |- D3 : of M2 T}
H2:{Gamma |- D2 : of (lam T1 R1) T}
H3:{Gamma |- T1 : ty}*
H4:{Gamma, n:tm |- R1 n : tm}*
H5:{Gamma, n1:tm |- R2 n1 : tm}*
H6:{Gamma, n2:tm, n3:of n2 T1 |- D n2 n3 : step (R1 n2) (R2 n2)}*

==================================
exists D3, {Gamma |- D3 : of (lam T1 R2) T}

Subgoal subject_reduction.2 is:
 exists D3, {Gamma |- D3 : of (R N) T}

Subgoal subject_reduction.3 is:
 exists D3, {Gamma |- D3 : of (app M N2) T}
```

```
Subgoal subject_reduction.4 is:
 exists D3, {Gamma |- D3 : of (app M4 N) T}
```

At the end of this, we have four cases, one for each step rule, with the current proof state reflecting the `step-lam` case.

One new tactic used is the `prune` tactic. In the `step-lam` case, applying the inductive hypothesis creates a term which must be applied to several unneeded nominal constants. This can be removed using the `prune` tactic.

```
subject_reduction.1>> case H2.
...

subject_reduction.1>> apply IH to H6 H10.

...
H11:{Gamma, n5:tm, n6:of n5 T1 |- D1 n6 n5 n4 n3 n2 n1 n : of (R2 n5) T2}
...

subject_reduction.1>> prune H11.

...
H11:{Gamma, n5:tm, n6:of n5 T1 |- D1 n6 n5 : of (R2 n5) T2}
...

subject_reduction.1>> exists of_lam R2 T1 T2 ([x] [x1] D1 x1 x).

...

subject_reduction.1>> search.
Searching for derivation of:
{Gamma |- of_lam R2 T1 T2 ([x][x1]D1 x1 x) : of (lam T1 R2) (arr T1 T2)}
Searching for derivation of:
{Gamma |- R2 : {x:tm}tm}
Searching for derivation of:
{Gamma |- T1 : ty}
Searching for derivation of:
{Gamma |- T2 : ty}
Searching for derivation of:
{Gamma |- [x][x1]D1 x1 x : {x:tm}{z:of x T1}of (R2 x) T2}

Subgoal subject_reduction.2:

Vars: T1:o, R:(o) -> o, N:o, D2:o, T:o
Nominals: n:o
Contexts: Gamma{n}:c[]
IH: ...
H2:{Gamma |- D2 : of (app (lam T1 R) N) T}
```

```
H3:{Gamma |- T1 : ty}*
H4:{Gamma, n:tm |- R n : tm}*
H5:{Gamma |- N : tm}*


=================================
exists D3, {Gamma |- D3 : of (R N) T}


Subgoal subject_reduction.3 is:
 exists D3, {Gamma |- D3 : of (app M N2) T}


Subgoal subject_reduction.4 is:
 exists D3, {Gamma |- D3 : of (app M4 N) T}
```

The next case, dealing with `step-beta`, makes use of Adelfa's `inst` tactic. This tactic allows for the use of instantiation for LF, shown to be a meta-theoretic property of LF in Section 2.1.3. This translates, in the informal proof, to the usage of the STLC meta-theoretic property of instantiation, which comes automatically with the encoding due to higher order abstract syntax.

```
subject_reduction.2>> case H2.


...


subject_reduction.2>> case H10.


...
H7:{Gamma |- N : tm}
...
H15:{Gamma, n2:tm, n3:of n2 D3 |- D6 n2 n3 : of (R n2) T}
...


subject_reduction.2>> inst H15 with n2 = N.
Searching for derivation of:
{Gamma |- N : tm}
...
H11:{Gamma |- D5 : of N D3}
...
H16:{Gamma, n3:of N D3 |- D6 N n3 : of (R N) T}
...


subject_reduction.2>> inst H16 with n3 = D5.
Searching for derivation of:
{Gamma |- D5 : of N D3}


...
H17:{Gamma |- D6 N D5 : of (R N) T}
...


subject_reduction.2>> exists D6 N D5.
```

```
subject_reduction.2>> search.
Searching for derivation of:
{Gamma |- D6 N D5 : of (R N) T}


Subgoal subject_reduction.3:


Vars: D:o, N1:o, M:o, N2:o, D2:o, T:o
Contexts: Gamma{}:c[]
IH: ...
H2:{Gamma |- D2 : of (app M N1) T}
H3:{Gamma |- M : tm}*
H4:{Gamma |- N1 : tm}*
H5:{Gamma |- N2 : tm}*
H6:{Gamma |- D : step N1 N2}*


==================================
exists D3, {Gamma |- D3 : of (app M N2) T}


Subgoal subject_reduction.4 is:
 exists D3, {Gamma |- D3 : of (app M4 N) T}
```

From here, the proof makes use of tactics which have already been covered to prove the step-app2 and step-app1 cases. The full input to Adelfa for the entire proof shown in Figure 4.2.

```
Specification "stlc.elf".

Schema c :=
 {T}(x:tm,y:of x T).

Theorem subject_reduction : ctx Gamma:c, forall M1 M2 T D1 D2,
  {Gamma |- D1 : step M1 M2} => {Gamma |- D2 : of M1 T} => exists D3,
  {Gamma |- D3 : of M2 T}.

  induction on 1. intros. case H1.

  % step-lam case
  case H2.
  apply IH to H6 H10.
  prune H11.
  exists of_lam R2 T1 T2 ([x] [x1] D1 x1 x).
  search.

  % step-beta case
  case H2.
  case H10.
  inst H15 with n2 = N.
  inst H16 with n3 = D5.
  exists D6 N D5.
  search.

  % step-app2 case
  case H2.
  apply IH to H6 H12.
  exists of_app M N2 T U a1 D3.
  search.

  % step-app1 case
  case H2.
  apply IH to H6 H11.
  exists of_app M4 N T U D3 a2.
  search.
```

Figure 4.2: Full proof of subject reduction

Chapter 5

Conclusion

This thesis demonstrates an example of a proof of a property of a programming language in Adelfa, a proof assistant assisting in the creation and verification of proofs of rule based systems using a logic for reasoning about encodings of those systems in LF. The particular property proven is subject reduction of the Simply Typed Lambda Calculus. This property is a useful property with a nontrivial proof, and is proven for a lambda calculus that is a useful model for practical programming languages and more complicated lambda calculi. By demonstrating the proof of this property, we hope to have demonstrated the usefulness of Adelfa in creating proofs of programming languages.

One of the goals for this work was to identify enhancements to Adelfa that would make it a more flexible system in formalizing properties of programming languages. A particular addition that we suggest in this spirit is that of logic level definitions. Similar to the definitions in Abella [BCG+14], these would allow for declaring a name, possibly indexed by terms, and a formula, and being able to use the definition as an if or only if statement to create or use the declared definitions. This would allow for a new method for encoding properties of interest. In particular, this would allow for the usage of the full formula syntax of the logic when defining relations, rather than simply LF syntax. This is useful as the formula syntax is richer, including existential and universal quantification.

Bibliography

[BCG+14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[PS02] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.4 edition, December 2002.

[Sou21] Mary Southern. *A Framework for Reasoning About LF Specifications*. PhD thesis, University of Minnesota, May 2021. http://arxiv.org/abs/2105.04110.