Towards Extracting Explicit Proofs
from Totality Checking in Twelf

Yuting Wang
University of Minnesota, USA
yuting@cs.umn.edu

Gopalan Nadathur
University of Minnesota, USA
gopalan@cs.umn.edu

Abstract

The Edinburgh Logical Framework (LF) is a dependently
type λ-calculus that can be used to encode formal systems.
The versatility of LF allows specifications to be constructed
also about the encoded systems. The Twelf system exploits
the correspondence between formulas and types to give
specifications in LF a logic programming interpretation.
By interpreting particular arguments as input and others
as output, specifications can be seen as describing non-
deterministic functions. If particular such functions can be
shown to be total, they represent constructive proofs of meta-
theorems of the encoded systems. Twelf provides a suite
of tools for establishing totality. However, all the resulting
proofs of meta-theorems are implicit: Twelf’s totality checking
does not yield a certificate that can be given to a proof
checker. We begin the process here of making these proofs
explicit. We treat the restricted situation in Twelf where
context definitions (regular worlds) and lemmas are not used.
In this setting we describe and prove correct a translation
of the steps in totality checking into an actual proof in the
companion logic $\mathcal{M}_2$. We intend in the long term to extend
our translation to all of Twelf and to use this work as the
basis for producing proofs in the related Abella system.

1. Introduction

The Edinburgh Logical Framework (LF) is a general frame-
work for formalizing systems that are presented in a rule-
based and syntax-directed fashion [2]. LF is based on a
dependently-typed λ-calculus and, as such, supports the
higher-order abstract syntax approach to representing binding
structure. Because of its features, LF has proven to be extremely versatile in encoding formal systems such as
programming languages and logics.

One of the purposes for specifying a formal system is to be able to prove properties about it; such properties are
usually called meta-theorems. There are two approaches
to doing this relative to LF specifications. One of these
approaches is implemented by the Twelf system that is
based on according a logic programming interpretation to
LF specifications [3, 8]. In essence, Twelf interprets types as
relations and thereby transforms the question of validity of
a relation into one about the inhabitation of a type. Twelf also allows such relations to be modeled, i.e., it lets
some arguments to be designated as inputs and others to be
identified as outputs. Relations thus define non-deterministic functions from inputs to outputs. Twelf complements such a
treatment with tools for determining if particular relational
specifications represent total functions from ground inputs
to ground outputs. When the relations are about types that
specify particular aspects of formal systems, something that
is possible to do in the rich types language of LF, such
verifications of totality correspond to implicit, constructive
proofs of meta-theorems.

The second approach is to proving meta-theorems is to do
this explicitly within a suitable logic. The $\mathcal{M}_2$ logic has been
described for the specific purpose of constructing such proofs
over LF specifications [9, 10]. Meta-theorems are represented
by $\mathcal{M}_2$ formulas and proved by applying the derivation rules
in $\mathcal{M}_2$. A successful proof returns a certificate in the form of
a proof term that can be checked independently by another
(simpler) program.

Of the two approaches, totality checking has proven to be
vastly more popular: a large number of verifications based
on it have been described in the literature. By contrast,
although an automatic theorem prover based on $\mathcal{M}_2$ has
been developed [7], it has, to our knowledge, not been used in
many reasoning tasks. However, there is a potential drawback
to using totality checking in verification: this process does
not yield an outcome that can be provided to a third party
that can, for instance, easily check its correctness.

This paper begins an effort to address the abovementioned
shortcoming by developing a method for extracting an explicit
proof from totality checking. In this first step, we limit
ourselves to a subset of the kinds of verification treated
by Twelf. In particular, we do not consider meta-theorems
that require the use of contexts (defined via regular worlds
in Twelf) and also disallow the use of lemmas in specifications.
In this setting, we describe and prove correct a procedure
for obtaining a proof in $\mathcal{M}_2$ from the work done by Twelf
towards establishing totality.

The rest of the paper is organized as follows. Section 2
recalls LF and its interpretation in Twelf and introduces
some associated terminology needed in the paper. Section 3
describes totality checking. The components that make it up
have been presented in different settings (e.g., see [2, 8, 11])
and some aspects (such as output coverage checking) have
not been described in the formality needed for what we do
in this paper. Thus, this section represents our attempt to
We then introduce the Twelf system that provides LF specifications a logic programming interpretation, which becomes a basis for proving meta-theoretic properties of systems encoded in LF. We conclude the section with the definitions of a few technical notions like substitution and unification related to LF that will be needed in later parts of this paper.

2. The Edinburgh Logical Framework

We provide a brief summary of the Edinburgh Logical Framework (LF) in this section and explain its use in encoding the syntax and derivation rules of formal systems. We then introduce the Twelf system that provides LF proofs from totality checking. Section 5 concludes the paper with a brief discussion of future directions to this work.

2.1 Syntax and Typing Judgments

The syntax of LF is that of a simply typed λ-calculus, types as judgments, objects as proofs and inhabitation as provability principles. We illustrate this aspect by considering the encoding of the untyped λ-calculus; this example will be used also in later parts of the paper.

The object system syntax is given by the following rule:

\[ M ::= \lambda x.M \mid M_1 M_2 \]

We shall represent these terms using the LF type \( \text{tm} \). For instance, the term \( \lambda x.x \) would be represented by the LF object \( \text{evAbs}(\lambda \text{x}.x) \) encoded as an LF \( \text{evAbs} \) constant representing the lambda abstraction. In general, any object system judgment can be encoded in LF as an LF \text{abs} constant.

The equality relation \( \equiv \) between type families and objects corresponds to \( \beta \eta \)-conversion. The LF type theory guarantees that every well-typed LF term has an unique \( \beta \eta \)-long normal form called its canonical form. Thus, two well-typed LF terms are equal if their canonical forms are identical up to a renaming of bound variables. We will often confuse a well-typed LF term with its canonical form.

2.2 LF as a Specification Language

LF can be used to encode varied formal systems via the types as judgments, objects as proofs and inhabitation as provability principles. We illustrate this aspect by considering the encoding of the untyped λ-calculus; this example will be used also in later parts of the paper.

The object system syntax is given by the following rule:

\[ M ::= \lambda x.M \mid M_1 M_2 \]

We shall represent these terms using the LF type \( \text{tm} \) and object-level constants \( \text{app} \) and \( \text{abs} \) as follows:

\[ \text{evalAbs}(M) \mid \text{evalAbs}((\lambda x.M) N) \]

We will often confuse a well-typed LF term with its canonical form. The equality relation \( \equiv \) between type families and objects corresponds to \( \beta \eta \)-conversion. The LF type theory guarantees that every well-typed LF term has an unique \( \beta \eta \)-long normal form called its canonical form. Thus, two well-typed LF terms are equal if their canonical forms are identical up to a renaming of bound variables. We will often confuse a well-typed LF term with its canonical form.
Valid Families:

\[
\begin{align*}
\Gamma \vdash a : K & \quad \Gamma, x : A \vdash B : K \\
\Gamma \vdash \lambda x : A.B : K & \quad \Gamma \vdash \Pi x : A.B : type \\
\Gamma \vdash \lambda x : A.M : K[M/x] & \\
\Gamma \vdash A : K & \\
\Gamma \vdash A : K' & \\
\Gamma \vdash K \equiv K' & \\
\end{align*}
\]

Valid Objects:

\[
\begin{align*}
\Gamma, x : A \vdash B : M : B & \\
\Gamma \vdash \lambda x : A.M : K & \\
\Gamma \vdash \lambda x : A.B : A & \\
\Gamma \vdash \lambda x : A.M : K[M/x] & \\
\end{align*}
\]

Figure 1. Rules for Valid Type Families and Objects

object system: the \([\cdot]\) mapping is extended to \(\lambda\)-calculus types in such a way that \(\Gamma \vdash T_1 \rightarrow T_2 = [T_1] [T_2]\). The typing rules for the \(\lambda\)-calculus can then be encoded in LF using the signature:

\[
\begin{align*}
of &: \text{tm} \rightarrow tyo : \text{type} \\
of\text{-app} &: \Pi M : \text{tm}; \Pi N : \text{tm} \Pi T : tyo \Pi T : tyo \\
of M (\of\text{-abs} T_1 T) &: of N T_1 \rightarrow of (\of\text{-app} M N) T \\
of\text{-abs} &: \Pi M : \text{tm} \Pi T : tyo \Pi T : tyo \\
\end{align*}
\]

We assume here that \(\Delta \vdash M : T\) is encoded as \([\Delta] \vdash [M] : [T]\), where \([\Delta]\) is an LF context resulting from transforming every \(x_i : T_i\) in \(\Delta\) into \(x_i : \text{tm} ; y_i : of(T_i)\) using the signature.

LF expressions contain a lot of verbose type information. This situation can be eased by making some of the outermost binders in the types assigned to object level constant implicit by using tokens beginning with uppercase letters for the variables they bind. Thus, the constants for encoding the evaluation and typing rules for the \(\lambda\)-calculus can be shown by means of the following signature:

\[
\begin{align*}
\text{eval\text{-app}} &: \text{eval} (\of\text{-abs} M') \rightarrow \text{eval} (\of\text{-app} M N) V \rightarrow \text{eval} (\of\text{-app} M N) V \\
\text{eval\text{-abs}} &: \text{eval} (\of\text{-abs} M) \\
\text{of\text{-app}} &: of M (\of\text{-abs} T_1 T) \rightarrow of N T_1 \rightarrow of (\of\text{-app} M N) T \\
\text{of\text{-abs}} &: (\Pi x : (\text{tm} of x T_1) \rightarrow of (M x T) \rightarrow of (\of\text{-abs} M) (\of\text{-app} T_1 T) \\
\end{align*}
\]

When we make the binding implicit in this way, it will always be the case that the types for the bound variables can be uniquely inferred by a type reconstruction process [1].

When showing terms, we will also leave out the arguments for constants that correspond to the implicit binders, which are assumed that these too can be inferred. Using this convention, the LF object for the evaluation derivation considered earlier can be written as follows:

\[
\begin{align*}
\text{eval\text{-app}} (\text{eval\text{-abs}} (\lambda x : (\text{tm} x)) (\lambda x : (\text{tm} x))) \\
\text{eval\text{-abs}} (\lambda y : (\text{tm} y)) (\lambda y : (\text{tm} y))
\end{align*}
\]

The missing arguments of \(\text{eval\text{-app}}\) and the two occurrences of \(\text{eval\text{-abs}}\) will be filled in here by type reconstruction.

2.3 Twelf and Logic Programming in LF

Twelf is a tool based on LF that uses the representation principles just discussed for specifying and reasoning about formal systems. Twelf permits \(A \leftarrow B\) as an alternative syntax for \(B \rightarrow A\), treating \(\leftarrow\) as a left associative operator. An important idea underlying Twelf is that LF types can be given a logic programming interpretation so that they can be executed. The full description of the operational semantics of Twelf can be found in [1]. We consider here only the interpretation for constant definitions of the form

\[c : a_1 \ldots a_n \leftarrow a_{1_1} \ldots a_{m_1} \ldots a_{m_n} \]

where \(a : \Pi x_1 : A_1, \ldots, x_n : A_n,\text{type}\) is an LF type family. In the logic programming setting, this definition of \(c\) is called a clause that has a \(M_1 \ldots M_n\) as its head and \(a_{1_1} \ldots a_{m_1} \ldots a_{m_n}(1 \leq i \leq m)\) as its premises. We think of such a clause in the same way we would a Prolog clause. Thus, we call the type family \(a\) a predicate and we think of the clause as one for \(a\). Given a set of clauses for \(a\), we query Twelf for the solutions of goals of the form \(M_1 \ldots M_n\). Twelf interprets this as a question that asks if a well-formed term of type \(a_{1_1} \ldots a_{m_1} \ldots a_{m_n}\) exists, treating \(M\) and the free variables in \(M_1, \ldots, M_n\) as logic variables. Twelf performs goal-directed proof search using backchaining similar to Prolog, using unification to instantiate logic variables as needed to solve the inhabitation question [1]. We assume that the search will be based on trying the premises in left to right order.

2.4 Substitution and Unification in LF

A substitution \(\sigma\) in LF is a type-preserving mapping from variables to objects that differs from the identity at only finitely many places. We write \(\sigma\) as \((M_1/x_1, \ldots, M_n/x_n)\) where \(M_i/x_i(1 \leq i \leq n)\) are the only non-identity mappings. For any LF term \(t\), \(t[\sigma]\) is the term obtained by applying \(\sigma\) in a capture avoiding way to the free variables in \(t\).

Substitutions transform terms that are well-typed in one context into ones that are well-typed in another context, something that is asserted by the judgment \(\Gamma \vdash \sigma : \Gamma'\) that is defined by the following rules:

\[
\begin{align*}
\text{subst-ext} &: \Gamma \vdash \sigma : \Gamma' \\
\text{subst-typ} &: \Gamma \vdash (\sigma M/x) : (\Gamma' ; x : A)
\end{align*}
\]

Given a \(\sigma\) and an \(\Gamma'\), there is a unique \(\Gamma\) with the smallest domain such that \(\Gamma \vdash \sigma : \Gamma\) is derivable. We shall intend to pick out this \(\Gamma\) when we use this judgment in later sections. We will also want to use the judgment \(\Gamma \vdash \sigma : \Gamma' \quad \text{when the domain of} \quad \sigma\) is a subset of that of \(\Gamma'\). We will assume in this case that the domain of \(\sigma\) is extended with identity substitutions to match that of \(\Gamma'\). If \(\Gamma\) is the context \(x_1 : A_1, \ldots, x_n : A_n\), we write \(\Gamma[\sigma]\) to represent the context \(x_1 : A_1[\sigma], \ldots, x_n : A_n[\sigma]\). The composition of substitutions \(\sigma \circ \theta\) is defined as follows:

\[\sigma \circ \theta = (\sigma \circ \theta, M[\theta]/x)\]

The following lemmas are easily proved by induction on typing derivations related to substitutions:
Lemma 1. If \( \Gamma_2 \vdash \sigma_1 : \Gamma_1 \) and \( \Gamma_3 \vdash \sigma_2 : \Gamma_2 \) are derivable then so is \( \Gamma_3 \vdash \sigma \circ \sigma_2 : \Gamma_1[\sigma_2] \).

Lemma 2. If \( \Gamma \vdash \gamma \sigma : \Gamma' \) and \( \Gamma' \vdash U : V \) are derivable then so is \( \Gamma \vdash U[\sigma] : V[\sigma] \).

A unification problem \( S \) is a finite multiset of equations \( \{ t_i = s_i | 1 \leq i \leq n \} \) where, for \( 1 \leq i \leq n \), \( t_i \) and \( s_i \) are LF terms of the same kind or type. A substitution \( \sigma \) such that \( t_i[\sigma] = s_i[\sigma] \) for all \( i \) such that \( 1 \leq i \leq n \) is a unifier for \( S \). It is a most general unifier (mgu) if for any unifier \( \theta \) of \( S \) there exists a substitution \( \gamma \) such that for any term \( t \) we have \( \gamma \theta[\delta] = t[\sigma][\gamma] \). Not every unification problem in LF has an mgu and unifiability is also not decidable in general. However, these properties hold when all occurrences of free variables in the terms determining the unification problem are strict as per the following definition [3].

Definition 3. An occurrence of a free variable is strict if it is not in the argument of a free variable and all its arguments are distinct bound variables.

We will also be interested often in matching, i.e., unification where free variables occur in the terms on only one side of the equations. In this case decidability and the existence of most general solutions follows if every free variable has at least one strict occurrence in the terms. We refer to a term that satisfies this property as a strict term.

Given a context \( \Gamma = x_1 : A_1, \ldots, x_n : A_n \) we will often write \( M\Gamma \) to mean \( M x_1 \ldots x_n \). Similarly, given a substitution \( \sigma = (M_1/x_1, \ldots, M_n/x_n) \) we will write \( M\sigma \) to mean \( M M_1 \ldots M_n \).

3. Totality Checking in Twelf

Under the logic programming interpretation of LF specifications in Twelf, a type family \( a : \Pi \text{type} \) represents a relation between its arguments. This relation can be read as a meta-theorem about the system specified by interpreting particular arguments as inputs and others as outputs. For instance, the subject reduction theorem for the \( \lambda \)-calculus (STLC) states that the evaluation preserves types: for all terms \( E \) and \( V \) and types \( T \), if \( E \gg V \) and \( \cdot \gg E : T \) holds, then \( \cdot \gg V : T \) holds. Based on our encoding of evaluation and typing for the \( \lambda \)-calculus, we can define the following type family:

\[
\text{subred} : \text{eval}\; E\; V \rightarrow \text{of}\; E\; T \rightarrow \text{of}\; V\; T \rightarrow \text{type}.
\]

By assigning \( E, V, T \) and the first two arguments to \text{subred} as inputs and the third argument to \text{subred} as output, the proof search for \text{subred} \( D_1 \; D_2 \; D_3 \) becomes directional: it queries the existence of a derivation \( D_3 \) for \( \text{of}\; V\; T \) given derivations \( D_1 \) for \text{eval}\; E\; V \) and \( D_2 \) for \( \text{of}\; E\; T \). Thus, we can interpret \text{subred} operationally as a non-deterministic function that computes a ground output \( D_3 \) from ground inputs \( E, V, T, D_1, D_2 \). If we can show that this function is total, i.e., that, given any ground terms for the inputs, proof search will be able to find a satisfying ground term for the output in a finite number of steps, then we would have obtained a constructive proof for the subject reduction theorem.

Twelf uses the above approach to interpret meta-theorems as \textit{totality assertions} and their proofs as \textit{totality checking}. In Twelf notation, the subject reduction theorem is expressed as follows:

\[
\text{subred} : \text{eval}\; E\; V \rightarrow \text{of}\; E\; T \rightarrow \text{of}\; V\; T \rightarrow \text{type}.
\]

sr-app : \text{subred}(\text{eval-app}\; D_{ty1}\; D_{ty2})(\text{of-app}\; D_{ty1}\; D_{ty2}\; D_{ty})
\rightarrow \text{subred}(\text{eval}\; M(\text{abs}\; M'))
\rightarrow \text{subred}(D_{ty1} : \text{of}\; M(\text{arr}\; T_1\; T_2))
\leftarrow \text{subred}(D_{ty2} : \text{eval}\; M(\text{abs}\; M'))
\rightarrow \text{subred}(D_{ty2} : \text{eval}\; (M'\; N)\; V)
\rightarrow \text{subred}(D_{ty} : \text{of}\; (N\; T_1))
\rightarrow \text{subred}(D_{ty} : \text{of}\; V\; T),
\]

sr-abs : \text{subred}(\text{eval-abs}\; D_{ty}\; D_{ty}).

Figure 2. LF signature for \text{subred}

%total D (subred D _ _).

The mode declaration that begins with \%mode designates the explicit arguments prefixed by \+ as inputs and those prefixed by \- as outputs; such a designation must be extended to also include the implicit arguments. The declaration that begins with \%total asserts that \text{subred} represents a total non-deterministic function in the indicated mode. This declaration also identifies an argument on which to base a termination argument as we shall see presently.

To facilitate totality checking, the user must provide clauses for deriving typing judgments of the kind in question. For \text{subred}, these clauses might be the ones shown in Fig.

Observe that these clauses essentially describe a recursive method for constructing a derivation of the output type given ones for the input types; the object constants \text{sr-app} and \text{sr-abs} are used to encode these constructions.

Totality checking in Twelf is broken into mode checking, termination checking, input coverage checking and output coverage checking; it can be shown that the successful completion of each of these checks ensures totality of the non-deterministic function. Our interest is in extracting a proof from Twelf’s verification process. For this, we only need to know the structure of each of the checks and we elide a discussion of how they guarantee totality.

3.1 Mode Checking

Given a type family \( a : \Pi x_1 : A_1, \ldots, x_n : A_n, \text{type} \), we shall refer to the variables \( x_1, \ldots, x_n \) as its parameters. A mode declaration assigns polarities \( p_1, \ldots, p_n \) to these parameters, where \( p_i \) is either a positive polarity \( + \) that designates \( x_i \) to be an \textit{input parameter} or a negative polarity \( - \) that designates \( x_i \) to be an \textit{output parameter}. A mode declaration for \( a \) is well-defined if for any \( i, 1 \leq i \leq n \), such that \( p_i = + \), the parameters occurring in \( A_i \) have polarity \( + \). Thus, the input parameters in a type family with a well-defined mode never depend on its output parameters; this property is necessary for assigning a meta-theorem reading to the modeled type family. The binder of a type family with a well-defined mode can always be rearranged so that it has the form \( a : \Pi x_1 : A_1, \ldots, x_n : A_n, \text{type} \), where \( A_i \) and \( A_0 \) contain parameters that are assigned only positive and negative polarities, respectively. In the following discussion, we will consider only type families with well-defined modes whose binders also have this special form. A type family \( a \) with a well-defined mode represents the meta-theorem that given any ground terms \( \bar{r} \) for the parameters in \( A_1 \), there exists a derivation \( D : a \bar{s} \) that computes ground terms \( \bar{t} \) for the parameters in \( A_0 \). We write this meta-theorem formally as \( \forall \bar{r}. \exists \bar{t}. \exists \bar{d} : a \bar{d} \cdot \bar{r} \rightarrow \bar{t}, \).

The requirement that input parameters must not depend on output parameters provides us a means for extending polarity assignments for explicit parameters to cover also
bound in mode checking, we need the definitions of groundedness with respect to the empty context. Given a type family \( a \) with a well-defined mode, mode checking verifies that the clauses for \( a \) that represent a proof for the relevant meta-theorem respect the moding, i.e., that, given ground terms for the input parameters, if backchaining on the clause succeeds, then it will result in ground terms being produced for the output parameters. To formalize mode checking, we need the definitions of groundedness with respect to a context and of input and output consistency.

**Definition 4.** An LF term \( M \) is ground with respect to an LF context \( \Gamma \) if all variables in the canonical form of \( M \) are bound in \( \Gamma \).

Observe that ground terms are a special instance of this definition, i.e., they are terms that are ground with respect to the empty context.

**Definition 5.** Let \( a : \Pi x_1 : A_1, \ldots, x_n : A_n. \text{type} \) be a type family and let \( p_1, \ldots, p_n \) be a well-defined mode for \( a \). We say that \( M_1 \ldots M_n \) is input consistent relative to \( \Gamma \) if for all \( i \) such that \( 1 \leq i \leq n \) and \( p_i = + \) it is the case that \( M_i \) is ground with respect to \( \Gamma \). Similarly, a \( M_1 \ldots M_n \) is output consistent relative to \( \Gamma \) if for all \( i \) such that \( 1 \leq i \leq n \) and \( p_i = - \) it is the case that \( M_i \) is ground with respect to \( \Gamma \).

A term that \( a \) is applied to is called an input argument or an output argument, depending on whether it corresponds to an input parameter or an output parameter. Variables occurring in input arguments (output arguments) are called input variables (resp. output variables).

**Definition 6.** Let \( a : \Pi \Gamma^1, \Pi \Gamma^0. \text{type} \) be a type family with a well-defined mode, where \( \Gamma^1 \) is \( x_1 : A_1, \ldots, x_i : A_i \) and \( \Gamma^0 \) is \( x_{k+1} : A_{k+1}, \ldots, x_n : A_n \). Let \( c : A \leftarrow A_1 \leftarrow \cdots \leftarrow A_m \) be a clause for a where \( A = A_1 \ldots A_m \) and \( A_i = a M_i \ldots M_m \) for \( 1 \leq i \leq m \). Let \( \Gamma_0 \) be the context containing only the input variables that have a strict occurrence in \( A \) and for \( 1 \leq i \leq m \) let \( \Gamma_i \) be the context \( \Gamma_{i-1} \Gamma_i \), where \( \Gamma_i \) contains only the output variables that have a strict occurrence in \( A \). The clause \( c \) is mode consistent if, for \( 1 \leq i \leq m \), \( A_i \) is input consistent relative to \( \Gamma_{i-1} \) and \( A \) is output consistent relative to \( \Gamma_m \). A type family \( a \) is well-moded if it has a well-defined mode and every clause for \( a \) is mode consistent. An LF signature \( \Sigma \) is well-moded if every type family in it is well-moded.

The restriction to only strict occurrences in the above definition is based on the fact that the instantiations of only variables that have such occurrences are guaranteed to be ground when matching with a ground term.

As an example of the application of these definitions, it is easy to see that the mode provided for \( \text{subred} \) is well-defined and that \( \text{sr-app} \) and \( \text{sr-abs} \) are mode consistent. Thus the type family \( \text{subred} \) and the LF signature in Fig. 2 are well-moded. The definition of mode consistency formalizes what is determined by the mode checking algorithm described in the Twelf manual.

### 3.2 Termination Checking

Termination checking verifies that, given a well-moded LF signature and a goal whose input arguments are ground, backchaining on any clause will result in a finite computation. Termination is checked using a termination ordering. The fundamental termination ordering used in Twelf is the subterm ordering: \( M \preceq N \) if \( M \) is a strict subterm of \( N \). For example, the totality declaration \( \text{total } D \left( \text{subred } D \ldots \right) \) tells Twelf to verify termination of \( \text{subred} \) using the subterm ordering on its first argument that has type \( \text{eval } E \). Other orderings such as lexicographical ordering and simultaneous ordering are also supported.

Termination checking assumes that the input arguments involved in termination ordering are ground, a condition that must hold if the signature is well-moded and mode consistent and the input arguments of the original goal are all ground. For every clause, it checks that the input arguments of premises involved in termination ordering are smaller than corresponding inputs in the clause head. For instance, the two premises in \( \text{sr-app} \) have \( \text{Dev}_1 \) and \( \text{Dev}_2 \), respectively, as their first (explicit) argument. These are strict subterms of the input \( \text{eval-app } \text{Dev}_1 \text{Dev}_2 \) in the head of \( \text{sr-app} \).

The following theorem is proved in [8].

**Theorem 7.** Given a well-moded and termination-checked LF signature \( \Sigma \), every execution path for a well-typed and input consistent goal \( A \) will have only finitely many steps.

By the theorem if a type family \( a \) passes mode checking and termination checking, then any call to \( a \) with ground inputs will terminate. This does not, however, guarantee that \( a \) can be interpreted as a total function from its ground inputs to its ground outputs: for some ground inputs the execution might not be able to make progress. For this stronger guarantee, it must pass the input and output coverage checking.

### 3.3 Input Coverage Checking

Coverage checking is the general problem of deciding whether any closed term is an instance of at least one of a given set of patterns [11]. For example, in functional programming languages such as ML, coverage checking is used to decide if a function definition is exhaustive over a given data type. The complexity of coverage checking depends on the underlying term algebra. In ML which only involves simple types and prefix polymorphism, the coverage checking is simple. Coverage checking is significantly more complex in LF since it contains dependent types and we also have to consider patterns with variables of higher-order type.

A coverage goal and a coverage pattern in LF are both given by a term and a context with respect to which it is ground.

**Definition 8.** A coverage goal or pattern is a valid LF typing judgment \( \Gamma ⊢_\Sigma U : V \), where \( U \) is either an object or type and \( V \) is either a type or kind.

A coverage goal or pattern represents the collection of closed terms obtained by instantiating the variables in the context. Given a set of coverage patterns and a coverage goal, the task, as already noted, is to determine if every instance of the goal is an instance of one of the patterns. One possibility is that the goal is immediately covered by one of the patterns in the set.
Definition 9. A coverage goal $\Gamma \vdash \Sigma U : V$ is immediately covered by a pattern $\Gamma' \vdash U' : V'$ if there exists a substitution $\Gamma' \vdash U'[\sigma] : V'[\sigma]$ and $U \equiv U'[\sigma], V \equiv V'[\sigma]$. A coverage goal is immediately covered by a finite set of patterns if it is immediately covered by one of the patterns in the set.

In the more general case, different instances of the coverage goal may be covered by different patterns.

Definition 10. $\Gamma \vdash \Sigma U : V$ is covered by a set of patterns $\mathcal{P}$ if, for every (ground) substitution $\sigma$ such that $\cdot \vdash \Sigma \sigma : \Gamma$, it is the case that $\cdot \vdash U[\sigma] : V[\sigma]$ is immediately covered by $\mathcal{P}$.

If it can be shown that the given coverage goal is immediately covered by the given set of patterns, then the task of coverage checking is obviously done. If the goal is not immediately covered by the patterns, then we consider applying a splitting operation to the goal. Splitting uses knowledge of the signature to generate a set of subgoals whose simultaneous coverage implies coverage of the original goal. We consider here only a restricted form of the operation defined in \cite{11} that disallows splitting on variables of function type and hence the use of context definitions (regular worlds) in Twelf proofs.

Definition 11. Let $\Gamma \vdash \Sigma U : V$ be a coverage goal and let $\Gamma$ be $\Gamma_1, x : A_k, \Gamma_2$ where $A_k$ is an atomic type. Suppose that for every constant $c : \Pi C. A_c$ in $\Sigma$ it is the case that either the unification problem $(A_c = A_c, x = c \Gamma_c)$ does not have a solution or it has an mgu. Splitting is then applicable in this case and it generates the following set of subgoals:

$$\{ \Gamma', \Gamma_2[\sigma'] \vdash U[\sigma] : V[\sigma] \mid c : \Pi C. A_c \in \Sigma, (A_c = A_c, x = c \Gamma_c) \text{ is unifiable and } \sigma' \text{ is its mgu and } \Gamma' \vdash \Sigma \sigma' : (\Gamma_1, x : A_k, \Gamma_2) \}$$

Note that splitting is finitary, i.e., its application produces only finitely many subgoals from a given goal. The following theorem, proved in \cite{11}, shows that splitting preserves the coverage:

Theorem 12. Let $\Gamma \vdash \Sigma U : V$ be a coverage goal and let $\{ \Gamma_i \vdash U_i : V_i \mid 1 \leq i \leq n \}$ result from it by splitting. Then $\Gamma \vdash \Sigma U : V$ is covered by a set of patterns if and only if $\Gamma_1 \vdash U_1 : V_1$ is covered for $i$ such that $1 \leq i \leq n$.

The general notion of coverage checking described here is specialized in Twelf to input coverage checking that verifies that there is some clause to backchain on for any call to a with ground inputs. The coverage patterns and goal to be used in the framework of general coverage checking for this purpose are defined as follows.

Definition 13. For a given type family $a : \Pi I'. \Pi O'. type$ in a well-formed LF signature $\Sigma$, the input coverage goal is $\Gamma' \vdash a \Gamma'$ : $\Pi O'. type$. Corresponding to a clause $c$ for a whose head is a $M_1 \ldots M_n$, let $\sigma_c$ be the substitution $(M_1/x_1, \ldots, M_n/x_k)$ where $k$ is the length of $\Gamma'$ and let $\Gamma'_c$ be the context containing the variables occurring in $M_1, \ldots, M_n$. The set of input patterns for a relative to $\Sigma$ then consists of the following:

$$\{ \Gamma'_c \vdash a \sigma'_c : \Pi O'[\sigma_c].type \mid c \text{ is a clause for } a \text{ in } \Sigma \}$$

The input coverage patterns for $a$ are derived from the clauses for $a$ by fixing the input arguments to $a$ to be those for the head of the respective predicate. Moreover, these arguments in the input coverage goal are set to be variables. Thus, if coverage checking as per Definition 10 verifies that this goal is covered by the corresponding patterns, then any call to $a$ with ground input arguments and variables for the output arguments will match the head of at least one clause for $a$, thus guaranteeing the possibility of backchaining on it.

A terminating procedure for (input) coverage checking that is based on a repeated use of splitting and immediate coverage is described for LF in \cite{11}. We do not present this procedure here since our interest is in the extraction of an explicit proof from the results of totality checking. In particular, we assume that we are presented at the outset with a sequence of splitting operations applied to an input coverage goal that lead to a set of subgoals that pass immediate coverage checking.

### 3.4 Output Coverage Checking

The premises in a clause for the type family correspond to recursive calls. The output arguments in such recursive calls could potentially limit the success set. Since coverage checking based on input coverage assumes success any time the input arguments in a goal match those in the head of a clause, it is necessary to verify that the output arguments of premises do not falsify this assumption. We describe here a method for ensuring that this is the case; this presentation is a formalization of what we understand output coverage checking to be in Twelf \cite{6}.

One way in which the success set may get constrained is if a output variable appears in an input argument. Output freshness is a criterion designed to avoid such a possibility.

Definition 14. A clause $c : A \leftarrow A_1 \leftarrow \ldots \leftarrow A_m$ satisfies the output freshness property iff, for $1 \leq i \leq m$, the sets of output and input variables of $A_i$ are disjoint.

We assume output freshness for clauses in what follows. A more involved requirement is that the form of the output arguments not limit the coverage of the clause. To formalize this we first identify output coverage goals and patterns.

Definition 15. Let $a : \Pi I'. \Pi O'. type$ be a type family in a well-formed LF signature $\Sigma$, where $\Gamma'$ is $x_1 : A_1', \ldots, x_n : A_n'$, and $\Gamma^O$ is $x_{k+1} : A_{k+1}', \ldots, x_m : A_m'$. Further, let $c : A \leftarrow A_1 \leftarrow \ldots \leftarrow A_m$ be a clause in $\Sigma$, where $A$ is a $M_1 \ldots M_n$ and, for $1 \leq i \leq m$, $A_i$ is a $M_i' \ldots M_{i+1}'$. Finally, for $1 \leq i \leq m$, let $\sigma_i$ be the substitution $(M_i/x_1, \ldots, M_k/x_k)$. Then the output coverage pattern for $c$ is $\Gamma'_c, \Gamma^O \vdash A_c : type$ where $\Gamma'_c$ and $\Gamma^O$ are contexts formed respectively from the input and output variables of $A_c$. The output coverage goal for $A_c$ is $\Gamma'_c ; \Gamma^O[\sigma_i] \vdash a \sigma'_i : \Gamma^O : type$.

Intuitively, the output coverage pattern corresponding to a premise $A_i$ is $A_i$ itself. The output coverage goal, on the other hand, is obtained by retaining the inputs to $A_i$ while maximally generalizing its outputs. To check if an output coverage goal is covered by the single pattern, we may split on an output variable but a way that ensures no variable other than the one being split on is instantiated.

Definition 16. Let $\Gamma'_c, \Gamma^O \vdash A : type$ be an output coverage goal where $\Gamma^O$ is $\Gamma_c, x : A_c, \Gamma_2$ for some atomic type $A_c$. Suppose that for every constant $c : \Pi A. C \in \Sigma$ it is the case that either $A_c$ is not an instance of $A_c$, or there is a most general substitution $\sigma$ such that $A_c = A_c[\sigma]$, such a substitution must obviously not instantiate any variable in $\Gamma'O, \Gamma^O$. Then, output splitting generates the set of subgoals:

$$\{ \Gamma'_c, \Gamma_c, \Gamma_2[\sigma] \vdash A[\sigma] : type \mid c : \Pi C. A_c \in \Sigma, A_c \text{ is an instance of } A_c, \sigma' \text{ is a}$$
most general substitution such that \( A_x = A_i[\sigma'] \),
\[ \sigma = ((c, r)[\sigma'] / x, r') \] and \( \Gamma_1 \vdash \sigma : \Gamma_3 \).

Eventually we want every output coverage subgoal that is produced by splitting to be equivalent to the sole output coverage pattern for each premise. Immediate output coverage captures the relevant equivalence notion.

**Definition 17.** An output coverage goal \( \Gamma^i; \Gamma^o \vdash \) \( M' : \text{type} \) is immediately covered by an output coverage pattern \( \Gamma^i \vdash M : \text{type} \) if \( M \equiv M'[\sigma] \) for some substitution \( \sigma \) that only renames variables and is such that \( \Gamma^i \vdash \sigma : \Gamma^i, \Gamma^o \).

An output coverage goal is covered by an output coverage pattern if every subgoal that is produced from it by some applications of output splitting is immediately covered by the pattern. In summary, output coverage checking ensures output freshness for every clause in the signature and it further checks that the output coverage goal for every premise of every clause is covered by the corresponding output coverage pattern for the premise.

4. **Explicit Proofs for Meta-Theorems**

Meta-theorems about formal systems can also be stated and proved in a logic. The logic \( \mathcal{M}_2 \) is designed for doing this based on the LF encodings of such systems \([10]\). \( \mathcal{M}_2 \) is a constructive logic formally presented via a sequent calculus. Formulas in \( \mathcal{M}_2 \) have the form \( \forall \Gamma_1, \exists \Gamma_2, \top \), where \( \Gamma_1 \) and \( \Gamma_2 \) are valid LF contexts. The universal (existential) quantification is omitted when \( \Gamma_1 (\Gamma_2) \) is empty. As we can see, the formulas in \( \mathcal{M}_2 \) are limited to the \( \Pi_1 \) form, where all existential quantifiers follow the universal ones. Although this may not seem very expressive, we have seen in Section 3 that every theorem proved through totality checking has this form.

The judgments in the sequent calculus are of the form \( \Gamma \vdash P \in F \), where \( F \) is a formula, \( P \) is a proof term, \( \Delta \) is a set of assumptions and \( \Gamma \) is a valid LF context containing all free variables occurring in \( P \) and \( F \). Proof terms and assumptions are defined as follows:

- **Proof Terms:** \( P ::= \text{let } y = x \sigma \in P \mid \Gamma \Gamma P \mid \text{split } x \alpha (\Gamma) \in P \mid \langle \sigma \rangle \)
- **Assumptions:** \( \Delta ::= \mid \Delta, P \in F \)

A meta-theorem represented by the formula \( \forall \Gamma_1, \exists \Gamma_2, \top \), is proved by deriving the judgment \( \vdash P \in \forall \Gamma_1, \exists \Gamma_2, \top \). The proof term \( P \) is obtained as an output of the derivation. The resulting \( P \) represents a total function, as shown by the following theorem that is proved in \([9]\).

**Theorem 18.** If \( \vdash P \in \forall \Gamma_1, \exists \Gamma_2, \top \) is derivable for some \( P \), then for every closed substitution \( \Gamma \vdash \sigma_1 : \Gamma \) there exists a substitution \( \Gamma \vdash \sigma_2 : \Gamma_2[\sigma_1] \).

As an example of this theorem, recall from Section 3 the formula stating the subject reduction theorem:
\[ \forall \Gamma, \forall \Gamma_1, \exists \Gamma_2, \top : \text{tm}, \text{V} : \text{tm}, \text{T} : \text{ty}, D_1 : \text{eval} E \text{V}, D_2 : \text{of} E \text{T}. \]
\[ \exists D_3 : \text{of} V \text{T} : \exists D_3 : \text{subred} \text{E} \text{V} \text{T} \text{D}_1 \text{D}_2 \text{D}_3, \top. \]

If we can get a proof term for this formula, then we can conclude that for any closed terms \( D_1 : \text{eval} E \text{V} \) and \( D_2 : \text{of} E \text{T} \), there exists a term \( D_3 : \text{of} V \text{T} \). Given the adequacy of the LF encoding, we can conclude the subject reduction theorem holds for the actual system.

The subsections that follow present the derivation rules for \( \mathcal{M}_2 \).

4.1 The Quantifier Rules

\[
\begin{align*}
\Gamma \vdash \exists \sigma : \Gamma_1 & \quad \Gamma ; \Delta_1, x \in \forall \Gamma_1, \exists \Gamma_2, \top \quad \text{∀-L} \ \\
\Gamma ; \Delta_1, x \in \exists \Gamma_1, \top, \Delta_2 
& \quad \text{let } y = x \sigma \in P \in F \quad \text{∀-L} \\
\Gamma ; \Delta_1, x \in \exists \Gamma_1, \top, \Delta_2 & \quad \text{split } x \alpha (\Gamma_1) \in P \in F \quad \text{∃-L} \\
\Gamma ; \Delta & \quad \text{let } y = x \sigma \in P \in F \quad \text{∀-R} \\
\Gamma ; \Delta & \quad \text{split } x \alpha (\Gamma_1) \in P \in F \quad \text{∃-R} \\
\end{align*}
\]

Figure 3. Quantifier Rules

The quantifier rules for \( \mathcal{M}_2 \) that are presented in Fig. 3 are the most basic ones for the logic. If we ignore the proof terms in boxes, we can see that these rules are similar to the conventional ones for an intuitionsitic logic. The \( \exists \)-L and \( \forall \)-R rules introduce fresh eigenvariables to the context. The \( \forall \)-L rule instantiates an assumption with a witnessing substitution to get a new assumption. \( \exists \)-R finds a witnessing substitution and finishes the proof. Note that weakening is implicit in \( \exists \)-R.

4.2 Recursion

The recursion rule is
\[
\begin{align*}
\Gamma ; \Delta & \quad x \in \forall \Gamma_1, \exists \Gamma_2, \top \quad \text{ recur} \\
\Gamma ; \Delta & \quad \mu x \in F.P \in F \\
\end{align*}
\]

with the proviso that \( \mu x \in F.P \) must terminate in \( x \). This rule adds the goal formula as an inductive hypothesis to the set of assumptions. For a proof based on this rule to be valid, the proof term must represent a terminating computation as the side condition guarantees. This condition is presented formally in Definition 7.8 of \([9]\) using the termination ordering on LF terms that was discussed in Section 3.2.

4.3 Case Analysis

The case analysis rule considers all the possible top-level structures for a ground term instantiating an eigenvariable \( x \) of atomic type \( A_k \) in the context \( \Gamma \) in a judgment of the form \( \Gamma ; \Delta \rightarrow P \in F \). Since \( A_k \) is atomic, the only cases to consider are those where the head of the term is a constant from the LF signature.

To state the rule, we need to extend the definition of proof terms with a case construct:

- **Patterns:** \( R ::= \Gamma', \Gamma'' \rightarrow M \)
- **Cases:** \( \emptyset ::= \emptyset \mid R \rightarrow P \)
- **Proof Terms:** \( P ::= \ldots \mid \text{case } x \alpha (\emptyset) \)

The case rule is
\[
\begin{align*}
\Gamma_1, x : A_k, \Gamma_2, \Delta & \rightarrow \emptyset \in F \quad \text{case} \\
\Gamma, \Delta & \quad \text{case } x \alpha (\emptyset) \in F \\
\end{align*}
\]

where \( \Gamma_1, x : A_k, \Gamma_2, \Delta \rightarrow \emptyset \in F \) is derived by considering all constants in \( \Sigma \) whose target type unifies with \( A_k \). The derivation rules for this judgment are shown in Fig. 3. The \text{siguni} rule produces a new premise (and a new case in the proof term) for every constant \( c \in \Sigma \) that has a type \( \Pi_1 \pi. A_c \).
Totality checking verifies that an LF signature provided by the user represents a total function that computes ground outputs from ground inputs. This section describes an algorithm for generating proofs from totality checking and represents a total function whose termination property is ensured by termination checking.

### 5.2 Initial Step

Starting with the initial sequent \( \vdash \top \equiv \forall \Gamma^I, \exists \Omega^O, \exists D : a \Gamma^I \Gamma^O, \top \), we apply the recur rule to introduce the goal formula as an inductive hypothesis. Then we apply the \( \forall \)-R rule to introduce hypotheses in \( \Gamma^I \) into the LF context.

\[
\Gamma^I, \Delta \equiv \forall \Gamma^I, \exists \Omega^O, \exists D : a \Gamma^I \Gamma^O, \top \equiv \forall \Gamma^I, \exists \Omega^O, \exists D : a \Gamma^I \Gamma^O, \top \quad \forall \text{-R recur}
\]

### 5.3 Case Analysis Steps

In this phase, we translate input coverage checking into applications of the case rule. We assume that input coverage checking succeeds and returns a sequence of splitting operations that leads to subgoals covered by input patterns. We show that every splitting operation can be translated into an application of case. In Section 4.3, we described that splitting on the variable \( x \) in an input coverage goal \( \Gamma_1^I, x : A_c, \Gamma_2^I \vdash A : \Pi^O \cdot \text{type} \) produces a subgoal \( \Gamma_1^I, \Gamma_2^I \equiv \forall \sigma : \Pi^O \cdot \text{type} \) for every constant \( c : \Pi_c \cdot A_c \) such that \( A_c \) unifies with \( A \). This corresponds exactly to case analysis of \( x \). To formally prove the equivalence, we define a relation between input coverage goals and \( M_2 \) sequents:

**Definition 19.** Let \( \equiv \) be a binary relation between input coverage goals and \( M_2 \) sequents. \( \Gamma \equiv \top \) holds if and only...
if \( G \) is an input coverage goal \( \Gamma^i \vdash x : \Pi^O, \text{type} \) and \( S \) is an \( M_2 \) sequent \( \Gamma^i ; \Delta \rightarrow \exists^O, \exists D : A \Gamma^O, \top \) where

\[
\Delta = \forall^i, \exists^O, \exists D : a \Gamma^O, \top.
\]

We overload the \( \equiv \) relation with finite sets of coverage goals and \( M_2 \) sequents:

**Definition 20.** Given a finite set of input coverage goals \( G = \{G_i \mid 1 \leq i \leq n\} \) and \( M_2 \) sequents \( S = \{S_i \mid 1 \leq i \leq n\} \), \( G \equiv S \) if and only if \( G_i \equiv S_i \) for \( 1 \leq i \leq n \).

Then we have the following lemma:

**Lemma 21.** Given \( G \equiv S \) where \( G \) is \( \Gamma^i_1, x : A, \Gamma^i_2 \vdash \forall \sigma : \Pi^O, \text{type} \) and \( S \) is \( \Gamma^i_1, x : A, \Gamma^i_2 ; \Delta \rightarrow \exists^O, \exists D : A \Gamma^O, \top \), if \( G \) is a set of subgoals resulting from splitting on \( x \) in \( G \) and \( S \) is the set of frontier sequents resulting from applying \( \text{case} \) to \( x \) in \( S \), then \( G \equiv S \).

**Proof.** Let \( G_i \) be a subgoal \( \Gamma^i_1, \Gamma^i_2[\sigma] \vdash A[\sigma] : \Pi^O, \text{type} \) resulting from splitting for some constant \( c \in \{c_1, \ldots, c_k\} \) such that \( A_i \) and \( A_k \) unifies and \( \Gamma^i_2 \vdash \forall \sigma = mgu(A_i, x = c_1 \Gamma_i) \) if \( \Gamma^i_1, x : A, \Gamma^i_2 \vdash \forall \sigma : \Pi^O, \text{type} \). By the definition of case analysis, there must be a sequence \( S_i = \Gamma^i_1, \Gamma^i_2[\sigma] ; \Delta \rightarrow \exists^O, \exists D : A[\sigma], \Gamma^O, \top \) from \( G \) resulting in \( \text{case} \) application. Thus we have \( G_i \equiv S_i \) by the definition of \( \equiv \). Since goals in \( G \) one-to-one correspond to sequents in \( S \), we have \( G \equiv S \).

The following lemma leads to an algorithm for translating splitings to applications of \( \text{case} \):

**Lemma 22.** Given a set of input coverage goals \( G \) and a partial \( M_2 \) proof with frontier sequents \( S \) such that \( G \equiv S \), if for some \( G_k \in G \) such that \( G_k = \Gamma^i_1, x : A, \Gamma^i_2 \vdash \forall \sigma : \Pi^O, \text{type} \) splitting on \( x \) in \( x \) results in a new set of coverage goals \( G', \) then applying the \( \text{case} \) rule on \( x \) in \( S_k \) where \( G_k \equiv S_k \) results in a partial \( M_2 \) proof with frontier sequents \( S' \) such that \( G' \equiv S' \).

**Proof.** The coverage goals \( G_i \in G \) for \( i \neq k \) and frontier sequents \( S_i \in S \) for \( i \neq k \) are still present and related by \( \equiv \) after splitting and applying \( \text{case} \). The new coverage goals generated by splitting are related to the new frontier sequents generated by applying \( \text{case} \) by Lemma 21.

Starting with the partial \( M_2 \) proof resulting from the initial step, we translate input coverage checking into applications of \( \text{case} \). We maintain a set of input coverage goals \( G \) and a partial proof with frontier sequents \( S \) such that \( G \equiv S \) for the translation. Initially, \( G \) is a singleton containing \( \Gamma^i_1, \Gamma^i_2[\sigma] = \Pi^O, \top \) and \( S \) is a singleton containing \( \Gamma^i_1, \Gamma^i_2, \exists^O, \exists D : a \Gamma^O, \top \) that is the frontier sequent of the proof in Figure 6. By Lemma 22, for every splitting operation on some \( G_i \in G \), we are able to apply the \( \text{case} \) rule to the corresponding frontier sequent \( S_i \in S \) where \( G_i \equiv S_i \) to get \( G' \) and \( S' \) such that \( G' \equiv S' \). In the end, we get a set of input coverage goals \( G' \) that are immediately covered and a partial proof tree with frontier sequents \( S' \) such that \( G' \equiv S' \).

### 5.4 Generating Proof Trees from Clauses

Let \( G' = \{G_i \mid 1 \leq i \leq n'\} \) and \( S' = \{S_i \mid 1 \leq i \leq n'\} \), where \( G' \equiv S' \), be the input coverage goals and sequents resulting from case analysis steps. To finish the \( M_2 \) proof, we need to derive sequents in \( S' \). For this we define the instantiation of an \( M_2 \) sequent:

**Definition 23.** Let \( S \) be an \( M_2 \) sequent \( \Gamma^i ; \Delta \rightarrow \exists^O, \exists D : A \Gamma^O, \top \) where \( \Delta = \forall^i, \exists^O, \exists D : a \Gamma^O, \top \).

The following lemma shows that immediate coverage in input coverage checking can be reflected into instantiation of \( M_2 \) sequents:

**Lemma 24.** Given an input coverage goal \( G' \) and an input coverage pattern \( G \) such that \( G' \) is immediately covered by \( G \) under a substitution \( \sigma \), if \( S \) and \( S' \) are \( M_2 \) sequents for which \( G \equiv S \) and \( G' \equiv S' \) hold, then \( S' \) is the instantiation of \( S \) under \( \sigma \).

**Proof.** Straightforward from definitions of immediate coverage, instantiation and \( \equiv \).
the variable $D_i : A_i$. A set of sequents \( \{ S_{ci} \mid 0 \leq i \leq m \} \) is defined where \( S_{ci} = \Gamma_{ci} : \Delta_i \rightarrow \exists \Gamma^o[\sigma'_i] ; \exists D : \sigma'_i \Gamma^o \Gamma^o \top \).

The following lemma relates this definition to mode checking:

**Lemma 26.** Given a clause \( c \), the context \( \Gamma_{ci} \) as defined in Definition 23 is a superset of the context \( \Gamma_i \) as defined in Definition 2 for \( i \) such that \( 0 \leq i \leq m \).

**Proof.** It is easy to see that \( \Gamma_i \) can be obtained by removing variables \( D_j \) for \( 1 \leq j \leq i \) and variables that do not have a strict occurrence in the type of \( c \).

By this lemma and the definition of mode consistency, we know that for a constant \( c : A_i, A_1 \cdots A_m \) the input variables to \( A_i \) are contained in \( \Gamma_{ci,1} \) for \( 1 \leq i \leq m \).

The output coverage goals and patterns in Definition 15 are redefined as follows: the output coverage pattern for \( A_i \) is now \( \Gamma_i^o, \Gamma^o \vdash_i A_i : \text{type} \) and the output coverage goal for \( A_i \) is now \( \Gamma_i^o, \Gamma^o[\sigma'_i] \vdash A_i : \text{type} \). This is a superfluous change from Definition 15 since the context \( \Gamma_i^o \) containing input variables to \( A_i \) is closed and a subset of \( \Gamma_{ci,1} \). Splitting and immediate coverage will never affect variables that are not in \( \Gamma_i^o \). Thus splitting and immediate coverage in Definition 16 and 17 are adopted to this new definition in a straightforward manner.

By the definition, \( S_{ci} \) is the sequent \( S \). We construct a proof for \( S \) by recursively constructing proofs for \( \{ S_{ci} \mid 0 \leq i \leq m \} \). In the base step we construct a proof for \( S_{ci,1} \). In the recursive steps we construct proofs for \( S_{ci,1} \) for \( 1 \leq i \leq m \).

**Base Step.** In the base step we translate the construction of outputs into an application of \( \exists \text{-}L \) to \( S_{ci,1} \) as shown in Figure 7. By Definition 29 \( \Gamma_{ci,1} \) contains \( D_1 \): 

\[
\frac{\Gamma_{ci,1} \vdash \sigma_{i,1}' : \Gamma^o[\sigma'_i], D : \sigma_{i,1}' \Gamma^o}{\exists \text{-}L}
\]

**Figure 7.** Finishing the Proof by Applying \( \exists \text{-}R \) Rule

\( A_1, \ldots, D_m : A_m \). Letting \( M_{k+1}, \ldots, M_n \) be the output arguments to the head of \( c \) and \( \sigma'_i = (M_{k+1}/x_{k+1}, \ldots, M_n/x_n) \), \( \sigma_m = (\sigma_{i,1}'(c D_1 \ldots D_m)/D) \) in Figure 7.

**Lemma 27.** \( \Gamma_{ci,1} \vdash \sigma_{i,1}' : \Gamma^o[\sigma'_i], D : \sigma_{i,1}' \Gamma^o \) holds.

**Proof.** By the definition of mode consistency and Lemma 26 \( A \) is output consistent relative to \( \Gamma_{ci,1} \). Thus \( \Gamma_{ci,1} \) contains all free variables in \( \sigma'_i \). By the well-typedness of \( A \), \( \Gamma_{ci,1} \vdash \sigma_{i,1}' : \Gamma^o[\sigma'_i] \) holds. Thus we have 

\[
\frac{\Gamma_{ci,1} \vdash \sigma_{i,1}' : \Gamma^o[\sigma'_i], \Gamma_{ci,1} \vdash c D_1 \ldots D_m : A}{\text{subst-type}}
\]

**Recursive Steps.** For \( i \) such that \( 1 \leq i \leq m \), we create a proof tree for \( S_{ci,1} \) by first translating the recursive call represented by \( A_i = A_{M_{k+1}} \cdots M_n \) and then recursively generating the proof for \( S_{ci,1} \). A recursive call can be translated to an application of the inductive hypothesis in the \( M_2 \) proof as shown in Figure 8, in which \( \sigma'_i = (M_i/x_i, \ldots, M_{k+1}/x_k) \) is a substitution containing input arguments to \( A_i \).

In the simple case when the output arguments \( M_{k+1} \cdots M_n \) to \( A_i \) are variables, the frontier sequent resulting from applying \( \forall \text{-}L \) and \( \exists \text{-}L \) is exactly \( S_{ci,1} \). By recursively generating the proof for \( S_{ci,1} \) we finish the translation. However, in many cases the output arguments in \( A_i \) are not variables. For example, in the subred example, the first premise of \( \text{eval-app} \) is

\[
\text{subred}(\text{Dev} : \text{eval } M (\text{abs } M')) (Dty_i : \text{of } M (\text{arr } T_i T))
\]

which has an output of \( \text{abs } Dty_i \). By applying the inductive hypothesis on \( \text{Dev} \) and \( Dty_i \), we get a new variable \( D_3 : (\text{of } \text{abs } M')(\text{arr } T_i T) \) in the context. We need to perform an inversion (case analysis) on \( D_3 \) to introduce the output variable \( Dty_i \) into the context.

The inversion consists of case rules translated from the output coverage checking. The translation is similar to translating input coverage checking. First we define a relation between output coverage goals and \( M_2 \) sequents.

**Definition 28.** Let \( G \) be a binary relation between output coverage goals and \( M_2 \) sequents. \( G \equiv S \) if \( G \) is an output coverage goal \( \Gamma^o, \Gamma^o \vdash A : \text{type} \) where \( \Gamma^o \) contains input parameters and \( \Gamma^o \) contains output parameters and \( S \) is an \( M_2 \) sequent \( \Gamma^o, \Gamma^o, D : A; \Delta \rightarrow F \) where \( D \) contains the inductive hypothesis and \( F = \exists \Gamma^o[\sigma'_i], \exists D : \sigma_{i,1}' \Gamma^o \Gamma^o \top \).

**Definition 29.** Given a finite set of output coverage goals

\[
\mathbb{G} = \{ G_i \mid 1 \leq i \leq n \}
\]

and \( M_2 \) sequents \( \mathbb{S} = \{ S_i \mid 1 \leq i \leq n \} \), \( \mathbb{G} \equiv \mathbb{S} \) if and only if for each \( G_i \equiv S_i \) for \( 1 \leq i \leq n \).

Then the following lemma holds:

**Lemma 30.** Given \( G \equiv S \) where \( G \) is \( \Gamma^o, \Gamma^o, x : A_i, \Gamma^o \vdash A : \text{type} \) and \( S \) is \( \Gamma^o, \Gamma^o, x : A_i, \Gamma^o, D : A; \Delta \rightarrow F \), if \( \mathbb{G} \) is the set of subgoals resulting from splitting on \( x \) in \( G \) and \( \mathbb{S} \) is the set of sequents resulting from applying case rule to \( x \) in \( S \), then \( \mathbb{G} \equiv \mathbb{S} \).

**Proof.** Similar to Lemma 21. Note that all free variables in \( F \) are bound in \( \Gamma^o \). Since variables in \( \Gamma^o \) are not instantiated by splitting in output coverage checking, they are also not instantiated by applying the corresponding case rule in the \( M_2 \) proof. Thus \( F \) remains the same after applying case.

By Lemma 30 we can prove a lemma similar to Lemma 22 from which we derive an algorithm for translating output coverage checking. Let \( S_{ci,1} \) be the frontier sequent resulting from applying the inductive hypothesis (as shown in Figure 8) and \( G_{ci} \) be the output coverage goal for \( A_i \). By the definition of \( \equiv \) and output coverage goals, we have \( G_{ci} \equiv S_{ci,1} \). Let \( G_{ci} \) be the output coverage patterns for \( A_i \). By Definition 29 and output coverage patterns, we have \( G_{ci} \equiv S_{ci} \).

The output coverage checking performs a sequence of splitting operations to \( G_{ci} \) and returns a set of subgoal \( G' \) immediately covered by \( G_{ci} \). Those splitting operations are translated into applications of case rules to the frontier sequent \( S_{ci,1} \), resulting in a partial proof with frontier sequents \( S' \) such that \( G' \equiv S' \). To relate the frontier sequents in \( S' \) to \( S_{ci} \), we prove the following lemma, which reflects the immediate
coverage in output coverage into equivalence between $M_2$ sequents up to renaming.

**Lemma 31.** Given an output coverage goal $G'$ and an output coverage pattern $G$ such that $G'$ is immediately covered by $G$, if $G \equiv \sigma \circ S$ and $G' \equiv \sigma \circ S'$ then $S'$ is the instantiation of $S$ under $\sigma$ which is a substitution that only renames variables.

**Proof.** Similar to Lemma 29. Because we restricted the instantiation of immediate coverage to a substitution that only renames variables in output coverage checking, it is reflected into equivalence between $M_2$ sequents up to renaming.

By applying Lemma 29 to $G' \equiv \sigma \circ S'$ and $G_{\circ i} \equiv \sigma \circ S_{\circ i}$, we have that for every $S' \in S$, $S'$ is an instantiation of $S_{\circ i}$ under a renaming substitution. The following lemma shows that instantiations of $M_2$ proofs under renaming substitutions are also valid proofs.

**Lemma 32.** Let $S'$ be an instantiation of $S$ under a renaming substitution, if $S$ has a $M_2$ proof, then so does $S'$.

**Proof.** By induction on the structure of the proof for $S$ and case analysis of the rule applied to $S$. The proof for most of cases is straightforward. When the rule applied is case, the proof relies on the fact that for a unification problem, compositions of mgu's with renaming substitutions are still mgu's.

At this point, we recursively generate a proof for $S_{\circ i}$ and apply Lemma 31 to get proofs for sequents in $S'$. This finishes the construction of the proof for $S_{\circ i-1}$.

### 5.4.2 Correctness of the Translation of Clauses

The following lemma shows that proofs translated from clauses are valid:

**Lemma 33.** Given a clause $c : A \leftarrow A_1 \leftarrow \ldots \leftarrow A_m$, the input coverage pattern $G$ derived from $c$ and the $M_2$ sequent $S$ such that $G \equiv \sigma \circ S$. The algorithm for translating clauses to $M_2$ proofs generate an $M_2$ proof $P$ for $S$ from $c$.

**Proof.** We define $M_2$ sequents $\{S_{\circ i} \mid 0 \leq i \leq m\}$ as in Definition 25. By the definition we have $S = S_{\circ 0}$. We prove that the algorithm constructs a correct proof for $S$ by induction on $i$. In the base case, Lemma 27 guarantees the $\equiv$ is correctly applied to get a proof for $S_{\circ m}$. In the inductive case, assume we have a correct proof for $S_{\circ i}$ for some $i$ such that $1 \leq i \leq m$, we prove that applications of $\forall$-L, $\exists$-L and case rules to $S_{\circ i-1}$ are correct. By the definition of mode consistency in Definition 26 and Lemma 26, the context $\Gamma_{\circ i-1}$ in the premise of $\forall$-L, $\Gamma_{\circ i-1} \vdash \pi_j : \Gamma_j$, contains all variables in $\sigma_j$. Thus this premise holds by LF typing rules and $\forall$-L is correctly applied. The correctness of $\exists$-L and case rules is obvious from their definitions. By Lemma 26 and Lemma 37 after applying those rules we get a partial proof with frontier sequents $S$, which is completed by applying Lemma 32 to the proof for $S_{\circ i}$.

### 5.4.3 Correctness of the Instantiation of Proofs

As described in the beginning of Section 5.4, we have to instantiate the proof generated from clauses to derive frontier sequents in the partial proof resulting from case analysis steps. In general, the provability of an $M_2$ sequent is not closed under instantiations. For instance, a case rule might no longer be applicable after an instantiation because the side condition for case, that every unification problem must either have an mgu or no solution, may no longer hold after the instantiation. In our case, an $M_2$ proof to be instantiated is translated from a totality checked clause, which has a particular structure such that the same rules can be applied to prove the instantiation of the $M_2$ sequent. Specifically, it does not use recur rule and case rules in it are translated from splitting in output coverage checking, which only involves matching problems instead of general unification problems. From those observations, we can prove the following instantiation lemma.

**Lemma 34.** If $D$ is an $M_2$ proof generated from clauses as described in Section 5.4.1 for the sequent $S = \Gamma \Delta \rightarrow F$, then given any substitution $\Gamma' \circ \sigma : \Gamma$ and an instantiation $S'$ of $S$ under $\sigma$ there exists a proof $D'$ for $S'$.

**Proof.** The proof is by induction on the structure of $D$ and case analysis of the rule applied to $S$ in $D$. The important case is for the case rule, in which we prove that the structure of the case are maintained under instantiation. Proof for other cases is straightforward. See Theorem 58 in the appendix for the details.

### 5.5 Correctness of the $M_2$ Proof Generation

We have shown that almost all applications of the $M_2$ rules in our proof generation are correct, except for recur rule in the initial step which has a side condition that the proof term must represent a terminating computation. Given an totality checked LF signature $\Sigma$ and a type family $a : \Pi i. \Pi o. \text{type} \in \Sigma$ such that $\Gamma = \frac{\forall l}{\exists \forall o} \circ \sigma : a l \Gamma o \circ \top$ and $\Pi \mu x. \forall l. \exists o. \top \circ P = \mu x. \forall l. \exists o. \top \circ P$. Intuitively, the translation algorithm should reflect the termination ordering in totality checking to that in the proof term $P$. Then the termination property of $P$ is guaranteed by the termination checking. To formally prove that, we need an instantiation lemma for termination ordering. The following lemma proves that subterm ordering holds under instantiation.

**Lemma 35.** Let $\preceq$ be the subterm ordering between $\forall$-L terms. If $M \preceq N$ and $\sigma$ is an substitution, then $M[\sigma] \preceq N[\sigma]$.

**Proof.** The formal definition of $\preceq$ is described in [8]. The proof is by a straightforward induction on the derivation rules for $\preceq$.

By this lemma, the same property can be proved easily for lexicographical and simultaneous ordering. These are all orderings supported by $M_2$. Then we prove that the termination property holds for $P$.

**Lemma 36.** If $P = \mu x. \forall l. \exists o. \top \circ (x_1 : A_1, \ldots, x_k : A_k)$. $P'$ is the proof term generated by the translation algorithm, then $P$ terminates in $x$.

**Proof.** Suppose every recursive call $x \ M_1 \ldots \ M_n$ in the proof term occurs in a position such that $x_1, \ldots, x_k$ are instantiated by $\sigma = \sigma_i \circ \ldots \circ \sigma_m$ where $\sigma_i$ for $1 \leq i \leq m$ are substitutions in case rules along the execution path and $\sigma$ is their compositions, we have to prove that $M_1, \ldots, M_k$ are smaller than $x_1[\sigma], \ldots, x_k[\sigma]$ relative to some termination ordering. In our translated proofs, a recursive call corresponds to a premise $A$ in some clause $c$. Let $M_1, \ldots, M_n$ be the input arguments to $A$ and $M'_1, \ldots, M'_n$ be the input arguments to the head of $c$. By termination checking we have $M_1, \ldots, M_n$ is smaller than $M'_1, \ldots, M'_n$ relative to the termination ordering.
In the proof term $P'$, the recursive call occurs in a position where $x_1,...,x_n$ are instantiated by the case rules translated from input coverage checking. Letting $\sigma$ be the composition of substitutions in those case rules, we have $x_i[\sigma] = M_i[\sigma']$ for $i$ such that $1 \leq i \leq k$, where $\sigma'$ is the substitution for instantiating the proof translated from $c$, as described in Section 5.4. By the translation algorithm, this recursive call has the form $\sigma (M_1[\sigma'])...(M_k[\sigma'])$. Applying Lemma 35 to the order between $M_1,...,M_k$ and $M'_1,...,M'_k$, we get $(M_1[\sigma'])...(M_k[\sigma'])$ is smaller than $x_1[\sigma],...,x_k[\sigma]$ relative to the termination ordering.

Then we have our main theorem:

**Theorem 37.** Given an LF signature $\Sigma$ that is totality checked without using contexts and lemmas, for every type family $a : \Pi \exists \exists O. \top$ in $\Sigma$, the proof generation algorithm generates an $M_2$ proof for $\vdash \exists \exists \exists O. \exists D : a \Gamma \Gamma O. \top$.

**Proof.** We prove that every application of an $M_2$ rule is valid.

1. Obviously, the application of $\forall \Rightarrow$ in the initial step is correct.
2. The applications of case rules translated from input coverage checking are correct by Lemma 22.
3. The translation from clauses to $M_2$ proofs are correct by Lemma 33. By instantiating those proofs the frontier sequents resulting from translating input coverage checking are proved, which is proved by Lemma 24. The instantiated proofs are valid by Lemma 34.
4. Finally, the application of recur rule in the initial step is correct by Lemma 36.

Since all the proof rules are correctly applied, we get an $M_2$ proof for $\vdash \exists \exists \exists O. \exists D : a \Gamma \Gamma O. \top$.

6. Conclusions and Future Works

This paper has described a method for transforming totality checking in a restricted version of Twelf into explicit proofs in the logic $M_2$. We have also proved this method correct. Towards this end, we have adapted arguments for the correctness of particular components of totality checking into an argument for the correctness of the generated $M_2$ proof. Since $M_2$ is a consistent formal logic with established metaproperties, the existence of such a transformation boosts our confidence in totality checking. More importantly, the transformation yields explicit objects that can be traded as proof certificates and whose correctness can be checked independently of the procedure that generated them.

As we have noted already, we have not considered the entire class of specifications for which Twelf supports totality checking. One limitation is that we have not considered the proof of properties that are parameterized by changing signatures that adhere to constraints that are finitely described via regular worlds descriptions in Twelf. Another limitation is that we have not allowed for clauses that contain calls to predicates other than the predicate being defined. We believe our work can be extended to treat totality checking in this more general setting. However, we will need a richer target logic than $M_2$. In particular, the proofs we produce will have to be in the logic $M_2^+$ that extends $M_2$ with judgments with generalized contexts and that contains rules for lemma applications. We plan to examine this issue more carefully in the near future.

Another direction of ongoing inquiry is the correspondence between the formalization and validation of meta-theorems in the LF/Twelf framework and in the two-level logic framework of Abella. It has previously been shown that specifications in LF can be translated into clause definitions in the specification logic used in Abella in a way that preserves essential structure. By exploiting this correspondence, it appears possible to transform $M_2$ (and possibly $M_2^+$) proofs over LF specifications into Abella proofs over the related Abella specifications. The results of this paper and its extension would then yield a path to proofs in another tried and tested logical setup.

Acknowledgments: This work has been partially supported by the NSF Grant CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. J. of Automated Reasoning, 49(2):241–273, 2012.
[2] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In 2nd Symp. on Logic in Computer Science, pages 194–204, Ithaca, NY, June 1987.
[3] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Journal of the ACM, 40(1):143–184, 1993.
[4] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, Logical Frameworks, pages 149–181. Cambridge University Press, 1991.
[5] F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, 6th Symp. on Logic in Computer Science, pages 74–85. IEEE, July 1991.
[6] F. Pfenning and C. Schürmann. Twelf User’s Guide, 1.4 edition, Dec. 2002.
[7] B. Pientka. An insider’s look at LF type reconstruction: everything you (n)ever wanted to know. Journal of Functional Programming, 23(1):1–37, Jan. 2013.
[8] E. Rohwedder and F. Pfenning. Mode and termination analysis for higher-order logic programs. In Proceedings of the European Symposium on Programming, pages 296–310, Apr. 1996.
[9] C. Schürmann. Automating the Meta Theory of Deductive Systems. PhD thesis, Carnegie Mellon University, Oct. 2000. CMU-CS-00-146.
[10] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, 15th Conf. on Automated Deduction (CADE), volume 1421 of Lecture Notes in Computer Science, pages 286–300. Springer, 1998.
[11] C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003, volume 2758 of LNCS, pages 120–135. Springer, 2003.
[12] Z. Snow, D. Baelde, and G. Nadathur. A meta-programming approach to realizing dependently typed logic programming. In ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), pages 187–198, 2010.
A. Instantiation Theorem for $\mathcal{M}_2$

Theorem 38. If $D$ is an $\mathcal{M}_2$ proof generated from clauses as described in Section 5.4 for the sequent $S = \Gamma_1, \Gamma_2; \Delta \rightarrow F$, where $\Gamma_1$ contains input parameters and $\Gamma_2$ contains output parameters, then any substitution $\Gamma_1' \vdash_2 \sigma : \Gamma_1$ there exists a proof for $S' = \Gamma_1', \Gamma_2; \Delta[\sigma_1] \rightarrow F[\sigma_1]$.

Proof. By induction on the structure of $D$. According to the last rule used in $D$, there are the following cases:

Case: $\forall$-$\mathcal{R}$. The proof looks like:

$$
\frac{\Gamma_1, \Gamma_2 \vdash_\forall \sigma : \Gamma_3}{\Gamma_1, \Gamma_2, \Delta[\sigma_1], \forall \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1} \quad \forall$-$\mathcal{R}
$$

Applying I.H. to $D'$, we get

$$
\frac{\Gamma'_1, \Gamma'_2[\sigma_1]; \Delta[\sigma_1], \forall \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1}{\Gamma'_1, \Gamma'_2; \Delta[\sigma_1] \rightarrow F_1} \quad \forall$-$\mathcal{R}
$$

Case: $\exists$-$\mathcal{R}$. The proof looks like:

$$
\frac{\Gamma_1, \Gamma_2 \vdash \exists \sigma : \Gamma_3}{\Gamma_1, \Gamma_2, \Delta[\sigma_1], \exists \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1} \quad \exists$-$\mathcal{R}
$$

Applying I.H. to $D'$, we get

$$
\frac{\Gamma'_1, \Gamma'_2[\sigma_1]; \Delta[\sigma_1], \exists \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1}{\Gamma'_1, \Gamma'_2; \Delta[\sigma_1] \rightarrow F_1} \quad \exists$-$\mathcal{R}
$$

Case: $\exists$-$\mathcal{L}$. The proof looks like:

$$
\frac{\Gamma_1, \Gamma_2 \vdash_\exists \sigma : \Gamma_3}{\Gamma_1, \Gamma_2, \Delta[\sigma_1], \exists \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1} \quad \exists$-$\mathcal{L}
$$

Applying I.H. to $D'$, we get

$$
\frac{\Gamma'_1, \Gamma'_2[\sigma_1]; \Delta[\sigma_1], \exists \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1}{\Gamma'_1, \Gamma'_2; \Delta[\sigma_1] \rightarrow F_1} \quad \exists$-$\mathcal{L}
$$

Case: $\forall$-$\mathcal{L}$. The proof looks like:

$$
\frac{\Gamma_1, \Gamma_2 \vdash_\forall \sigma : \Gamma_3}{\Gamma_1, \Gamma_2, \Delta[\sigma_1], \forall \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1} \quad \forall$-$\mathcal{L}
$$

Applying I.H. to $D'$, we get

$$
\frac{\Gamma'_1, \Gamma'_2[\sigma_1]; \Delta[\sigma_1], \forall \Gamma_3, \Gamma_3, \Delta_2, \Delta_2[\sigma_1] \rightarrow F_1}{\Gamma'_1, \Gamma'_2; \Delta[\sigma_1] \rightarrow F_1} \quad \forall$-$\mathcal{L}
$$

Case: case. This is the only non-trivial case. The proof looks like:

$$
\frac{\Gamma'_1, \Gamma'_2[\sigma_1]; \Delta[\sigma_1] \rightarrow F[\sigma_1]}{\Gamma'_1, \Gamma'_2; \Delta \rightarrow F[\sigma_1]} \quad \text{case}
$$

Since $\sigma_1$ only instantiate input variables and case rules are applied only on output variables, $x$ and variables in $\Gamma_2$ will not be instantiated by $\sigma_1$ (but their types will). For a constant $c : \Pi c. A_x$, in $\Sigma$, we solve the unification problem ($c_a = A_x, x = c \Gamma_a$). If it does not have a solution, signonuni is applied. If it has an mgu, siguni is applied. If it has an uninfl but not an mgu, case rule is not applicable. Thus in a correct proof such as $D$, the last situation does rise. We prove that after the instantiation, the same rule can be applied to get a correct application of case rule.

The application of case rule after unification looks like:

$$
\frac{\Gamma'_1, x : A_x[\sigma_1], \Gamma_3, \Delta[\sigma_1] \rightarrow F[\sigma_1]}{\Gamma'_1, x : A_x[\sigma_1], \Gamma_3, \Delta[\sigma_1] \rightarrow F[\sigma_1]} \quad \text{case}
$$

For a constant $c : \Pi c. A_x$ in $\Sigma$, the unification problem becomes ($A_x[\sigma_1] = A_x, x = c \Gamma_a$), which is equivalent to ($A_x = A_x, x = c \Gamma_a$) since $\sigma_1$ does not contain variables in $\Pi c. A_x$. If ($A_x = A_x, x = c \Gamma_a$) does not have a solution, so does ($A_x[\sigma_1] = A_x, x = c \Gamma_a$). Thus we can apply the same signonuni rule in this case. If ($A_x = A_x, x = c \Gamma_a$) has an mgu $\sigma$, according to the translation algorithm the original proof looks like:

$$
\frac{D_1, \Gamma_1, \Gamma_2[\sigma]; \Delta[\sigma_1] \rightarrow F[\sigma_1]}{D_1, \Gamma_1, \Gamma_2[\sigma]; \Delta[\sigma_1] \rightarrow F[\sigma_1]} \quad \text{signuni}
$$

By applying I.H. to $D_1$, we get the following proof:

$$
\frac{D'_1, \Gamma'_1, \Gamma'_2[\sigma]; \Delta[\sigma_1] \rightarrow F[\sigma_1]}{D'_1, \Gamma'_1, \Gamma'_2[\sigma]; \Delta[\sigma_1] \rightarrow F[\sigma_1]} \quad \text{signuni}
$$

Let $\sigma'_1 = (\sigma_1, \sigma_d)$ such that $\Gamma'_1, \Gamma'_2 \vdash_\exists \sigma'_1 : (\Gamma_1, \Gamma_2)$. We have $\Gamma'_1, \Gamma'_2 \vdash_\exists \sigma \circ \sigma'_1 : \Gamma_c$. Let

$$
\sigma' = (\sigma_d, (c \Gamma_c)[\sigma \circ \sigma'_1/x, x \circ \sigma'_1])
$$

such that $\Gamma'_1, \Gamma'_2 \vdash_\exists \sigma' : (\Gamma_1, x : A_x[\sigma_1], \Gamma_c)$. If $\sigma'$ is an mgu for ($A_x[\sigma_1] = A_x, x = c \Gamma_a$), then we get a proof after instantiation using signuni rule:

$$
\frac{D'_1, \Gamma'_1, \Gamma'_2[\sigma'_1]; \Delta[\sigma_1] \rightarrow F[\sigma_1]}{D'_1, \Gamma'_1, \Gamma'_2[\sigma'_1]; \Delta[\sigma_1] \rightarrow F[\sigma_1]} \quad \text{signuni}
$$

where $D'_2$ is the rest of the instantiated proof. Here we use the fact that $\Gamma'_2[\sigma][\sigma_1] = \Gamma_2[(\sigma, (c \Gamma_c)[\sigma \circ \sigma'_1/x, x \circ \sigma'_1]) = \Gamma'_2[\sigma][\sigma'_1]$.

We then prove that $\sigma'$ is indeed an mgu for ($A_x[\sigma_1] = A_x, x = c \Gamma_a$).
1. \( \sigma' \) is a unifier. Since \( \sigma \) is an mgu for \((A_x = A_c, x = e \Gamma_c)\), we have \( A_x[\sigma] = A_c[\sigma] \) which is equivalent to \( A_x = A_c[\sigma] \). Thus \( A_x[\sigma_1][\sigma'] = A_x[\sigma_1] = A_c[\sigma_x][\sigma_1] = A_c[\sigma_c \circ \sigma_1'] = A_c[\sigma'] \). Also \( x[\sigma'] = (c \Gamma_c)[\sigma_c \circ \sigma_1'] = (c \Gamma_c)[\sigma'] \).

2. \( \sigma' \) is an mgu. Let \( \sigma_2 = (\sigma_{12}, M/x, \sigma_{c2}) \) be a unifier for \((A_x[\sigma_1] = A_c, x = c \Gamma_c)\) such that \( \Gamma_2 \vdash \sigma_2 : (\Gamma_2, x : A_x[\sigma_1]) \Gamma_c \). Let \( \sigma_2' = (\sigma_2, \sigma_{c2}) \) such that \( \Gamma_2', x : A_x[\sigma_1], \Gamma_c \vdash \sigma_2' : (\Gamma_2, x : A_x, \Gamma_c) \). We have \( \sigma_2' \circ \sigma_2 \) as a unifier for \((A_x = A_c, x = c \Gamma_c)\) such that \( \Gamma_2 \vdash \sigma_2 : (\Gamma_1, x : A_x, \Gamma_c) \). Since \( \sigma \) is an mgu for \((A_x = A_c, x = c \Gamma_c)\), there exists some \( \sigma_3 = (\sigma_{13}, \sigma_c) \) such that \( \Gamma_2 \vdash \sigma_3 : (\Gamma_1, \Gamma_c') \) and \( \sigma \circ \sigma_3 = \sigma_2' \circ \sigma_2 \). From \( \sigma \circ \sigma_3 = \sigma_2' \circ \sigma_2 \), we have \( \sigma_{13} = \sigma_1 \circ \sigma_{12} \) and \( \sigma_c \circ \sigma_3 = \sigma_{c2} \). Letting \( \sigma_4 = (\sigma_{12}, \sigma_{c3}) \), we prove that \( \sigma_2 = \sigma' \circ \sigma_4 \). For this we need to prove the following:

- \( \sigma_{12} = \sigma_{c1} \circ \sigma_4 \). This is obvious.
- \( \sigma_{c2} = (\sigma_2 \circ \sigma_2') \circ \sigma_4 \). We have \( \sigma_1' \circ \sigma_4 = (\sigma_1 \circ \sigma_{12}, \sigma_{c3}) = (\sigma_{13}, \sigma_c) = \sigma_3 \). Thus \( \sigma_{c2} = \sigma_2 \circ \sigma_3 = \sigma_c \circ (\sigma_1' \circ \sigma_4) = (\sigma_2 \circ \sigma_2') \circ \sigma_4 \).
- \( M = (c \Gamma_c)[\sigma_c \circ \sigma_1'][\sigma_4] \). Since \( x[\sigma_2] = (c \Gamma_c)[\sigma_2] \), i.e., \( M = (c \Gamma_c)[\sigma_2] \), we have \( M = (c \Gamma_c)[\sigma_2] = (c \Gamma_c)[\sigma_c \circ \sigma_1'] \circ \sigma_4 = (c \Gamma_c)[\sigma_c \circ \sigma_1'][\sigma_4] \). Thus, for any unifier \( \sigma_2 \) for \((A_x[\sigma_2] = A_c, x = c \Gamma_c)\) there exists a substitution \( \sigma_4 \) such that \( \sigma_2 = \sigma' \circ \sigma_4 \). We conclude that \( \sigma' \) is an mgu for \((A_x[\sigma_2] = A_c, x = c \Gamma_c)\). 

\( \square \)