A Debugging System for Language-parameterized Proofs

Charlesowityear Ly, Eswarasanthosh Kumar Mamillapalli, and Matteo Cimini

University of Massachusetts Lowell, Lowell MA 01854, USA

Abstract. Language-parameterized proofs can express proofs of language properties in a way that does not apply just to one language but, rather, to a class of languages. LANG-N-PROVE is a tool that takes a language-parameterized proof and a language definition as input and produces proofs for the Abella proof assistant.
When a language-parameterized proof is incorrect, the tool blames a proof instruction in the Abella code, but that is not helpful for the user. In this paper, we develop a debugging system that not only indicates the proof instruction that failed in the original language-parameterized proof, but also strives to describe the context in which such proof instruction has failed.
We have implemented our debugger within the LANG-N-PROVE tool, and we illustrate its application to several examples of debugging scenarios. Ultimately, our debugging system offers informative error messages.

1 Introduction[*]

When we develop new programming languages, it is desirable to determine whether they afford the properties that were intended at the time of design. Over the past years, some attention has been devoted to the development of techniques and tools for *automating* the verification of languages. Results in this area vary greatly in their scope and methods. Rule formats target bisimilarity properties in process algebras [21], intrinsic typing of evaluators establishes the type soundness of the language [11], automated theorem proving has been used to determine type soundness [16,22,25], as well as other noteworthy approaches.

Language-parameterized proofs [12] have been proposed as an approach to language verification that is based on the fact that proofs for languages often follow a schema that does not apply to one language but, rather, to many languages. Language-parameterized proofs then allow us to write proofs in a way that refers to a language definition given as input without knowing the particular operators, values, and so on, that the language has. To make an example, a part of the progress theorem for the case of `head` e calls the canonical form lemma for the list type and then proves the "progress" of `head` e by appealing to the existence of two reduction rules (one that handles the empty list and one that handles `cons`). But the progress theorem for the case of `fst` e is similar and calls

[*]If the paper is accepted, Matteo Cimini will give the presentation.

Excerpt of LANG-N-PROVE proof	Generated Abella proof
	26 `...`
12 `...`	27 `search.`
13 `for each e in Expression:`	28 `search.`
14 `  search`	29 `search.`
15 `...`	30 `search.` ✗
	31 `search.`
	32 `search.`
	33 `...`

Fig. 1. Intuition for the debugging problem in LANG-N-PROVE

the canonical form lemma for the product type and appeals to the existence of one reduction rule (as pairs are the only canonical form for the product type). Language-parameterized proofs allow language designers to express "if the operation is an elimination form of the type constructor c, apply the canonical form for type c" and "for all value forms of type c, appeal to the existence of a reduction rule."

LANG-N-PROVE [12] is a tool for expressing language-parameterized proofs. The tool takes a language definition and a language-parameterized proof as input and generates the proof code of the Abella proof assistant [8]. Prior work [12,15] has applied LANG-N-PROVE to express the language-parameterized proofs of the "syntactic approach" to type soundness of Wright and Felleisen [28] (canonical form lemmas, and progress and preservation theorems) for a class of functional languages without subtyping [12] and when subtyping is present [15]. [12] and [15] report applying the proofs to a plethora of language definitions and generating type soundness proofs that machine check in Abella (though some small parts must be manually provided. We refer readers to [12,15] to learn more about it.)

Problem in Debugging Language-parameterized Proofs LANG-N-PROVE generates proofs for the Abella proof assistant. When the language-parameterized proof input is incorrect, Abella may fail. The error message that Abella reports refers to the line number of the generated Abella proof. However, this error message is not helpful because language designers work at the level of the language-parameterized proof. To give an intuition of the problem, let us suppose that we are writing a proof and that after a case analysis on the structure of expressions we wrongly think that each case of expression forms can be simply proved with the proof instruction `search`. (This instruction is similar to Coq's `auto` and simply tells Abella to try to derive the goal with the current knowledge.) One of the features of LANG-N-PROVE is that it generates proof instructions by iterating over some grammar of the language given as input. We therefore can write the `for`-loop in Fig. 1 on the left (within a larger proof). Given a language as input, LANG-N-PROVE generates the series of `search` instructions in Fig. 1 on the right (within an even larger proof).

Suppose that, contrary to what we expected, `search` at Line 30 fails in proving the case for an expression form. Through no fault of its own, Abella correctly tells the user that Line 30 of its proof code has failed, but the language designer can hardly understand what the problem is because they work at the level of the language-parameterized proof, to begin with, and also because Line 30 is just one `search` among many. Each `search` has no apparent connection to the `for`-loop that generated them.

It would be desirable to integrate a debugging system into LANG-N-PROVE that can provide more helpful error messages.

Contributions In this paper, we develop a debugging system for LANG-N-PROVE. One of the main components of our debugger aims at "contextualizing" Abella instructions. We have instrumented LANG-N-PROVE so that when it generates Abella instructions it also attaches the context of the LANG-N-PROVE proof that generated them. For instance, suppose that the `search` that failed in the `for`-loop above was for the expression form (*app* e e). Our debugger provides an error message that says that such `search` was generated by Line 15 of the LANG-N-PROVE proof, that such instruction was in the context of a `for`-loop, and that it was generated for the iteration element (*app* e e).

Another component of our debugger aims at analyzing the proof instructions specifically produced by `for`-loops. We consider it an error when the `for`-loop generates an instruction for, say, iteration element (*app* e e), but Abella uses it in the context of the case for, say, (*if* e e e). This may happen, for example, when the `for`-loop generated too few instructions for (*if* e e e). That case was then not completed when Abella executes the proof, and so Abella starts grabbing the following instructions, which now are those generated by the `for`-loop for the next iteration element. (We will see concrete examples of this debugging scenario in Section 3 and 4.)

Our debugger includes other components, and Section 3 motivates and describes them. This paper makes the following contributions:

- We describe our debugging system in details.
- We have integrated our debugger into the implementation of LANG-N-PROVE, and we illustrate how our debugger handles several examples.

Overall, we believe that our debugger provides useful error messages. Our debugger is publicly available [19] and documents all our tests.

Structure of the Paper This paper is organized as follows. Section 2 provides a background overview of LANG-N-PROVE. Section 3 describes our debugging system through a series of motivating examples. Section 4 illustrates our debugger at work with a handful of examples. Section 5 discusses the limitations of our debugger. Section 6 discusses related work, and Section 7 concludes the paper.

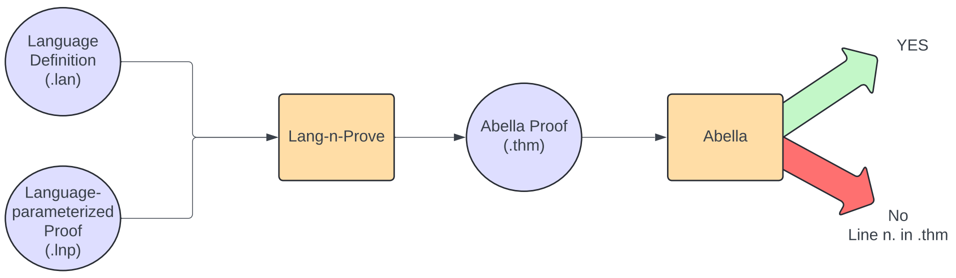
Language Definition (.lan)
Language-parameterized Proof (.lnp)
Lang-n-Prove
Abella Proof (.thm)
Abella
YES
No
Line n. in .thm

Fig. 2. Pipeline of Lang-n-Prove

2 Overview of Lang-n-Prove (Background)

Fig. 2 shows the pipeline of the Lang-n-Prove tool. Lang-n-Prove takes a language definition and a language-parameterized proof as input and generates an Abella proof, which Abella proof checks. We describe these elements below.

Language Definitions Language definitions are text files with extension `.lan`, and they contain a textual representation of operational semantics. For example, the following is the language definition for the λ-calculus with booleans.
We call this file `stlc_iff.lan`. (By convention, variables `R` are a shorthand for bindings such as (x)E.)

```
Expression E ::= (tt) | (ff) | (if E E E)
                 | x | (abs T (x)E) | (app E E).
Type T ::= (bool) | (arrow T T).
Value V ::= (tt) | (ff) | (abs T R).
Context C ::= [] | (app C E) | (app V C) | (if C E E).

Gamma |- (tt) : (bool).
Gamma |- (ff) : (bool).

Gamma |- (if E1 E2 E3) : T <== Gamma |- E1 : (bool)
                              /\ Gamma |- E2 : T /\ Gamma |- E3 : T.

Gamma |- (app E1 E2) : T2 <== Gamma |- E1 : (arrow T1 T2)
                             /\ Gamma |- E2 : T1.

Gamma |- (abs T1 R) : (arrow T1 T2) <== Gamma, x : T1 |- R : T2.

(app (abs T E) V) --> E[V/x] <== value V.
(if (tt) E1 E2) --> E1.
(if (ff) E1 E2) --> E2.
```

(This type of textual language definitions is not a novelty of LANG-N-PROVE. It is indeed inspired by the style that the Ott tool [26] has adopted long ago.)

To review operational semantics, languages have a grammar and a series of inference rules. The latter define the relations that are of interest of the language, such as a typing and a reduction relation. The "`<==`" symbol above should be read as "provided that." For example, the typing rules for `if` and `app` correspond to the standard rules

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}$$

The language above makes use of evaluation contexts, which declare which arguments of an expression constructor can be evaluated.

Abella proofs It would be natural to describe the second input element (language-parameterized proofs) at this point. However, we prefer to describe the output first, Abella proofs. The reason is that the proof language for language-parameterized proofs is derived from that of Abella. Abella proofs are in .thm files. (Abella requires other files but we are not concerned with them here.) Below is the syntax of the part of Abella that is relevant to this paper. We refer our reader to [8] for Abella's syntax.

$$\begin{array}{lll}
\textsf{Proof} & th ::= & \texttt{Theorem}\ name : f.\ \texttt{Proof}\ p. \\
\textsf{Proof instr.} & p \ ::= & \texttt{intros}\ \overline{name} \mid \texttt{search} \mid name : \texttt{case}\ name \\
 & & \mid name : \texttt{induction on}\ k \\
 & & \mid name : \texttt{apply}\ name\ \texttt{to}\ \overline{name} \\
 & & \mid \texttt{backchain}\ name \mid p.p
\end{array}$$

where *name* is the name of a theorem or hypothesis, k is a number, and where formulae f specify the statement of theorems. As we are concerned with proofs only in this paper, we omit describing their grammar. Intuitively, they include the standard logical connectives and quantifications (∀, ∃).

`intros` $name_1 \cdots name_m$ applies to a goal of the form $\forall X_1, X_2, \ldots X_n . f_1 \Rightarrow f_2 \ldots \Rightarrow f_m \Rightarrow f$, introduces the variables X_1, X_2, …, and X_n, and adds, as hypotheses of the current proof, f_1 with name $name_1$, f_2 with name $name_2$, …, and f_m with name $name_m$. (There may not be one name for each of them but we are not concerned about these details here.) Then, f becomes the new goal. `search` tries to prove the current goal using the knowledge available to the proof. The proof fails when `search` cannot prove the goal. $name_1$: `case` $name_2$ performs a case analysis on the hypothesis $name_2$ while $name_1$: `induction on` k performs an induction on the k-th formula in the series of implications of the goal. These instructions may open a series of proof subcases to be proven. Also, they may introduce newly derived hypotheses, whose given name is based on $name_1$. `induction` loads the inductive hypothesis with name $IH0$, if available, IH_k otherwise, for the first available name IH_k. `backchain` *name* is applied for a theorem *name* of form $\forall X_1, X_2, \ldots X_n . f_1 \Rightarrow f_2 \ldots \Rightarrow f_m \Rightarrow f$. It matches

$$\begin{array}{lll}
\textsf{LNP Proof} & \hat{th} & ::= \texttt{for each}\ Z\ \texttt{in}\ \hat{t},\ \texttt{Theorem}\ \hat{name} : \hat{f}.\ \texttt{Proof}\ \hat{p}. \\
\textsf{LNP Name} & \hat{name} & ::= name \mid name\_(\hat{t}) \\
\textsf{LNP Formula} & \hat{f} & ::= (\hat{name} : pname\ \hat{t_1} \ldots \hat{t_n}) \mid \ldots\ several\ other\ formulae\ \ldots \\
\textsf{LNP Term} & \hat{t} & ::= X \mid (opname\ \hat{t}_1 \cdots\ \hat{t}_n) \mid (X)\hat{t} \mid \hat{t}[\hat{t}/X] \\
 & & \quad \mid Z \mid n \mid cname \mid \hat{t} =_{\texttt{t}} \hat{t} \mid \hat{t}\ \texttt{is}\ cname \mid \hat{t}\ \texttt{in}\ \hat{t} \\
 & & \quad \mid \texttt{isVar}(\hat{t}) \mid \texttt{ofType}(\hat{t}) \\
 & & \quad \mid \texttt{isEliminationForm}(\hat{t}) \mid \texttt{isErrorHandler}(\hat{t}) \\
 & & \quad \ldots\ several\ other\ operations\ \ldots \\
\textsf{LNP Proof instr.} & \hat{p} & ::= \texttt{intros} \mid \texttt{search} \mid \hat{name} : \texttt{case}\ \hat{name} \\
 & & \quad \mid \hat{name} : \texttt{induction on}\ \hat{name} \\
 & & \quad \mid \hat{name} : \texttt{apply}\ \hat{name}\ \texttt{to}\ \hat{name}_1\ \cdots\ \hat{name}_n \\
 & & \quad \mid \texttt{backchain}\ \hat{name} \mid p.p \mid \texttt{noOp} \\
 & & \quad \mid \texttt{for each}\ Z\ \texttt{in}\ \hat{t} : \hat{p} \mid \texttt{if}\ \hat{t}\ \texttt{then}\ \hat{p}\ \texttt{else}\ \hat{p}
\end{array}$$

Fig. 3. Lang-n-Prove proof language

the goal of the current proof with f, and checks that f_1, f_2, ..., and f_m can be derived. We use `backchain` *name* when the theorem *name* successfully concludes the current proof case. Proofs can also be sequentially composed ($p.p$).

Language-parameterized proofs Lang-n-Prove provides a domain-specific proof language for expressing language-parameterized proofs. This proof language contains linguistics that are specific to interrogating operational semantics.

(Part of) the syntax of Lang-n-Prove from [12] is in Fig. 3. LNP proofs can generate a series of theorems and their proofs with `for`-loops. (We can generate one single theorem by iterating over a list of one element. The formalism has only one form to keep the syntax tidy.) Lang-n-Prove makes use of LNP names rather than names. The difference is that while names are essentially strings, LNP names can have a term as a suffix. (We will describe terms shortly.) For example, `for each` Z `in` *Type*, `Theorem` *canonical-form-*_(Z) : ... creates a series of theorems *canonical-form-int*, *canonical-form-bool*, *canonical-form-arrow*, etc., and their proofs, depending on the grammar of types of the language given as input. As before, we are not concerned with formulae, and therefore we omit describing LNP formulae $\hat{f}$ in full. The relevant LNP formula for us is the base case ($\hat{name}$: $pname\ \hat{t_1} \ldots \hat{t_n}$). Here, a formula has been given a name and it is in abstract syntax style (top-level predicate name *pname* applied to arguments). (We describe terms $\hat{t}$ below.) The proof language of Lang-n-Prove is derived from that of Abella. However, it includes a `for`-loop and an `if`-statement. Also, `intros` does not specify names because theorems use formulae with names already, and those names are then given to those formulae. Also, `induction` specifies the name of a hypothesis rather than a number. `noOp` is a no-operation and has no effect. `noOp` will not compile to any Abella proof code. Terms can be metavariables X, LNP variables Z (those bound by `for`-loops), numbers, category names *cname*, and they can be formed in abstract syntax style, that is, a top-level constructor name applied to a list of arguments. Furthermore,

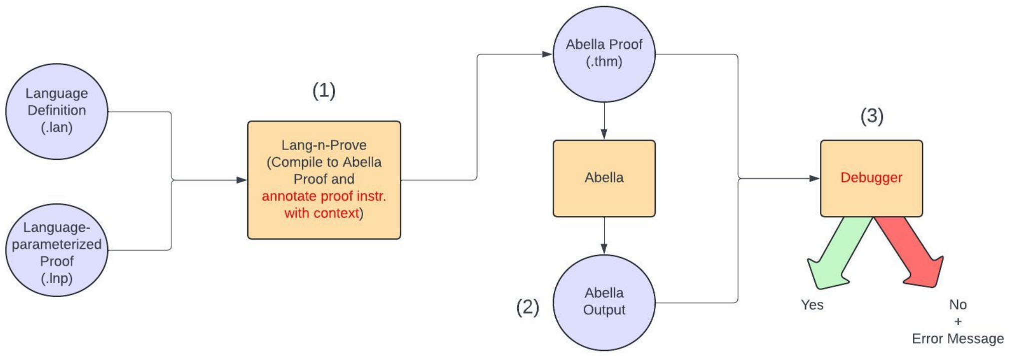
Language Definition (.lan)
Language-parameterized Proof (.lnp)
(1)
Lang-n-Prove (Compile to Abella Proof and annotate proof instr. with context)
Abella Proof (.thm)
Abella
(2)
Abella Output
(3)
Debugger
Yes
No + Error Message

Fig. 4. Pipeline of Lang-n-Prove with our debugging system

Lang-n-Prove contains LNP terms that are specific to operational semantics. Fig. 3 contains only a few of them so that we can give an idea. $\hat{t}$ `is` *cname* holds if, after $\hat{t}$ is evaluated, $\hat{t}$ can be derived by the grammar of *cname*. (Not to repeat it, we mention $\hat{t}$s but with the understanding that they are evaluated first.) $\hat{t}_1$ `in` $\hat{t}_2$ holds whenever $\hat{t}_1$ is an element of the list $\hat{t}_2$. `isVar`$(\hat{t})$ holds whenever $\hat{t}$ is a metavariable, `ofType`$(\hat{t})$ computes the top-level constructor name of $\hat{t}$ and returns the type that its typing rule assigns to it. For example, `ofType`$(tt) = bool$. `isEliminationForm`$(\hat{t})$ and `isErrorHandler`$(\hat{t})$ hold whenever $\hat{t}$ is an elimination form or an error handler, respectively. Lang-n-Prove determines this by inspecting the language, see [12] for the details of these checks and also for the rest of the operations, which are numerous and we cannot describe in this section.

3 A Debugging System for Lang-n-Prove

3.1 Overview of the Debugging System

Fig. 4 shows the pipeline of Lang-n-Prove with our debugging system. (Our additions in red.) This section gives a brief overview. Section 3.2, 3.3, 3.4, and 3.5 describe our debugging system in details with motivating examples.

(1) We have instrumented the function of Lang-n-Prove that compiles LNP proofs to Abella proofs so that every Abella proof instruction generated also carries the context it had in the `.lnp`.

(2) We launch Abella in "annotated mode" on the generated proof and we save the Abella output in order to analyze it after the execution of Abella has terminated. The "annotated mode" provides useful information about the Abella step-by-step execution of the proof given in input.

(3) We have implemented a debugger that takes two elements as input: The Abella `.thm` proof (where instructions now also inform of their `.lnp` context) and the Abella output. The debugger analyzes the two inputs and produces an informative error message when it detects an error.

Our debugger has four components. The first component aims at "contextualizing" errors (described in Section 3.2). The second component aims at determining whether Abella proofs align with the `for`-loops in LNP proofs (described in Section 3.3). The third component determines whether LNP proofs generate too many instructions for a theorem (described in Section 3.4). The fourth determines whether LNP proofs generate too few instructions that do not complete a proof (described in Section 3.5). In the remainder, we describe these components by progressively refining a proof for canonical form lemmas.

All the LNP proofs in this paper are small variations of the proofs in [12] and [15], or they are from [12] and [15] outright when indicated so.

3.2 Contextual Error Messages

We start writing a language-parameterized proof for canonical form lemmas. That is, we would like our proof to work on a host of languages. To recall, these lemmas establish the value forms for each type constructor. Below are two examples of this lemma for booleans and the function type.

$\texttt{Theorem}\ \textit{Canonical-form-list} : \forall e, T,$
$\textit{Main} :\ \emptyset \vdash e : \texttt{Bool} \Rightarrow\ \textit{ValHyp} : (e\ \textit{is value}) \Rightarrow e = \texttt{tt} \vee \texttt{e} = \texttt{ff}$

$\texttt{Theorem}\ \ \textit{Canonical-form-arrow}\!: \forall e, T_1, T_2,$
$\textit{Main} :\ \emptyset \vdash e : (T_1 \to T_2) \Rightarrow\ \textit{ValHyp} : (e\ \textit{is value}) \Rightarrow \exists T, e', e = \lambda x : T.e'$

The proof is by case analysis on the typeability of e, that is, by case analysis on *Main*. For the sake of example, we first try an obviously incorrect proof that simply strives to prove every case with `search`. The following is our LNP proof. The LNP formula that generates the statement of the theorem is rather complicated because it retrieves the values of the type of e. It is lengthy to describe, and since it does not play a role for us, we omit it. Nonetheless, this statement defines the premises *Main* and *ValHyp* above, and we will refer to these premises throughout our paper. We call this file `canonical1.lnp`.

```
for each ty in Type, Theorem Canonical-form-_(ty):
 ... language-parameterized statement ... (contains Main and ValHyp)
Proof.
intros.
_ : case Main. # with _ we do not give a name to derived hypotheses
for each e in Expression:
   search
```

When we run Lang-n-Prove with the following language with only booleans: (We call this file `iff.lan`. We show the relevant parts only.)

```
Expression E ::= (tt) | (ff) | (if E E E).
Type T ::= (bool).
Value V ::= (tt) | (ff).
Gamma |- (tt) : (bool).
Gamma |- (ff) : (bool).
```

The generated Abella proof contains a series of `search` similar to those in the introduction section. This Abella proof succeeds the cases for `tt` and `ff`, but fails that for `if` and blames its `search` at Line 10 of the `.thm` file. As we pointed our earlier, this is not an informative error message.

Our solution: We have instrumented Lang-n-Prove so that every instruction that will appear in Abella proofs also carries their line number (and character number) in the LNP proof. Then we have instrumented the function of Lang-n-Prove that compiles LNP proofs to Abella proofs so that every time we traverse a `for`-loop or an `if`-statement we record it as a contextual information. (We will be specific on the information we store shortly.) Abella instructions are then generated with a comment that contains the context in which they have been generated. For example, the failing `search` at Line 10 of the `.thm` file is

10 `search.` # 7,4 - `for`:6,0 ::: $(\textit{if}\ E_1\ \ E_2\ \ E_3)$

The compiler to Abella proofs evaluates `Expression` at Line 6 into its list of grammar items according to the language given as input. Then, it expands the `for`-loop into sequences of instructions. Therefore, the compilation knows the iteration element for which a proof instruction is generated. The comment is structured as follows. It begins with the line and character number of the instruction in the `.lnp` file. This is position 7,4 in `canonical1.lnp`. After that information, there is a sequence of *entries* separated by the character "-". In this case, there is only one entry. Information within the entry is separated by a colon ":". The first two items `for`:6,0 tell us that the instruction was inside the body of the `for`-loop at line number 6 and character number 0. After that, the entry contains two items, both of which are empty strings in this case (that is ::: after `for`:6,0). We will discuss those items in the next section. The last item tells us that the instruction was generated during the loop iteration for the element $(\textit{if}\ E_1\ \ E_2\ \ E_3)$. Our debugger saves contextual information upon traversing `if`-statements, as well. We will show an example in Section 4.

Before we speak about the debugging component that we have developed, we shall discuss the structure of the output of Abella (in "annotated mode"). The last lines of the Abella output for our example are the following

```
<a name="10"></a>
<pre>
Subgoal 3:

Variables: E3 E2 E1
Main : {typeOf empty (if E1 E2 E3) bool}
ValHyp : {value (if E1 E2 E3)}
H1 : {typeOf empty E1 bool}
H2 : {typeOf empty E2 bool}
H3 : {typeOf empty E3 bool}
============================
 if E1 E2 E3 = tt \/ if E1 E2 E3 = ff

Canonical-form-bool < <b>search.</b>
```

We call this a *block*. It starts with an `<a>` marker followed by a `<pre>` marker.

The `<a>` marker provides the line number in the `.thm` file of the instruction being executed. (Not in general: Technically speaking, this number, say k, is of the k-th instruction read from the `.thm` and processed by Abella. However, Lang-n-Prove generates the Abella `.thm` with one instruction per line, and therefore that number is exactly the line number in the `.thm` in our case.)

The `<pre>` marker contains the proof state, including the hypotheses, the goal, the instruction being read, for example. Notice that this marker should end with `</pre>` but, due to the error, the output prematurely ended.

The Abella output contains several blocks, sequentially one after another for each instruction read from the `.thm` file. The block that we have shown above is the last block in our example.

We have then developed a component that parses the Abella output, stores all the blocks in a list, and can retrieve information from blocks. This component retrieves from the last block the line of the last instruction executed by Abella. In our example, that is Line 10 of the `.thm` file. Then it retrieves the comment of this line and produces an error-friendly message from it. In our example, Line 10 is the one above and our debugger produces the message

"*Line 7, character 4 of the .lnp failed. The instruction is in the for-loop at line 6, character 0, and the element of the iteraton is (if E_1 E_2 E_3).*"

3.3 Detecting `for`-loops That Are Not in Synch with Abella

This information is useful. We quickly discovered the instruction in the LNP proof that failed, and that it does not work for the case of `if`. We can debug our LNP proof: Since `if` is not a value, its proof case can be proved by contradiction on *ValHyp*. Our new LNP proof is the following. We call this `canonical2.lnp`

```
Proof.
intros.
_ : case Main.
for each e in Expression  iterating Main at 2 :
   if e is Value
    then search
    else _ : case ValHyp
```

where we ignore for the moment the directive highlighted in gray. This LNP proof now works for `iff.lan`. However, it fails for `stlc_iff.lan` (functions and booleans). The last block of the output of Abella is

```
<a name="11"></a>
<pre>
Subgoal 4:

Variables: T1 E2 E1
Main : {typeOf empty (app E1 E2) bool}
```

```
ValHyp : {value (app E1 E2)}
H1 : {typeOf empty E1 (arrow T1 bool)}
H2 : {typeOf empty E2 T1}
============================
 app E1 E2 = tt \/ app E1 E2 = ff

Canonical-form-bool < <b>search.</b>
```

That is, `search` has failed. However, the case analysis at that point is analysing $(app\ E1\ E2)$, and we know that our LNP proof does not generated `search` for the non-value $(app\ E1\ E2)$. Something subtle is occurring.

The problem is that the `for`-loop iterates over *all* expressions, but abstraction $\lambda x : T.e$ is not typed as a boolean. Our LNP proof considers such case and generates `search` for it because it is a value. The Abella case analysis (on $\emptyset \vdash e :$ `Bool`), however, does not offer the case of $\lambda x : T.e$ at all. The mismatch is that the LNP proof thinks that `search` is for $\lambda x : T.e$ while, in actuality, Abella executes it for $(app\ E1\ E2)$. When this occurs, we say that the `for`-loop at Line 6 is *not in synch with Abella*.

We have developed a component that detects `for`-loops that are not in synch. To do so, we have augmented the syntax of Lang-n-Prove with the optional directive that is highlighted above: `iterating Main at 2`. This directive informs that the loop is connected with the exploration of the case analysis of *Main* and that the element being iterated by Abella will be the second argument of the formula stored as hypothesis *Main* during the execution of the proof.

Then, our instrumentation of Lang-n-Prove also includes this information as contextual information. (These are the fields that were empty with ::: in the previous section.) Lang-n-Prove then generates the following line in the `.thm`

`search.` #8,9-`for`:6,0: *Main* : 2 : $(abs\ T\ (x)E)$

Next, we have developed a component of the debugger that retrieves from this line the hypothesis name (*Main* in this case) and the position of the argument (2 in this case), and retrieves from the output of Abella above the term that Abella is analysing during the proof ($(app\ E1\ E2)$ in this case). With that in hand, we compare it to the last field of the contextual information ($(abs\ T\ (x)E)$ in this case). This is the term that the LNP proof believes it has generated the instruction for. As the two do not match, our debugger provides an error message that says that the `for`-loop is not in synch with Abella:

"*The* `for`*-loop at Line 6, character number 0 is not in synch with Abella. The case analysis was* `Main: typeOf empty (app E1 E2) bool` *but the element of the iteration was* $(abs\ T\ (x)E)$."

After this message, our debugger also provides the contextual error message of the instruction that failed, as it may be helpful.

When users do not use the `iterating` directive in `for`-loops, the fields above are empty as we have seen before (with :::), and those loops are not analyzed to see if they are in synch. Our debugger points out the first occurrence in the Abella output for which a loop is not in synch. Also, when there are nested loops, we point out the outermost `for`-loop that is not in synch.

3.4 Detecting Too Many Proof Instructions for Theorems

Before refining `canonical2.lnp`, we discuss another problem that might occur with it. We consider a version of `stlc_iff.lan` where function application appears before abstraction in the grammar: (We call this `stlc_iff_inverted.lan`)

```
Expression E ::= (tt) | (ff) | (if E E E)
                 | x | (app E E) | (abs T (x)E).
```

and the rest of `stlc_iff_inverted.lan` is the same of `stlc_iff.lan`. The proof generated by Lang-n-Prove for `canonical2.lnp` when this language is given as input prompts Abella to throw an error for the canonical form lemma of booleans. The `for`-loop in the LNP proof generates code for 5 cases: `tt`, `ff`, `if`, `app`, and `abs`[†], in this order because that is how they appear in the grammar. However, when Abella performs the case analysis on *Main*, it provides only 4 cases: `tt`, `ff`, `if`, and `app`. As we discussed in the previous section, the case of `abs` is not given because `abs` is not typed as a boolean. The proof code for these 4 cases is correct and does prove them. This means that Abella finds itself with a completed proof of the lemma, whereas the LNP proof has generated an extra `search` instruction for `abs`. The Abella tool throws an error at the encounter of a proof instruction that does not belong to any theorem.

The last block of the output of Abella for this error is

```
<a name="11"></a>
...
Canonical-form-bool < <b>case Mapp : ValHyp (keep).</b>
Proof completed.
</pre>
<a name="12"></a>
<pre class="code">
Abella <
```

The last block (block 12) prematurely ends without `</pre>`, while the second last block (block 11) announces that a proof has been completed, as highlighted in pink color above.

We have developed a simple component of our debugger. As Abella has failed, we already know that the last block is about an error. Our debugger checks whether the second last block announces a proof completed. If that is the case,

[†]We recall that the case analysis on *Main* does not give the case of variables x to prove because the type environment must be empty. We also recall from [12] that `for` does not iterate over variables x by default.

we also retrieve the name of the completed theorem from the second last block. This is the theorem that is highlighted above in gray color. Our debugger provides the error message

"*The proof of theorem* `Canonical-form-bool` *is completed but the .lnp has generated too many instructions for it.* "

After this message, our debugger also displays the contextual error message of the extra instruction. This message is helpful and says that the extra instruction is the `search` at Line 8 of the LNP proof, and that it was generated by a `for`-loop for the iteration element $(abs\ T\ (x)E)$.

3.5 Detecting Too Few Proof Instructions for Theorems

Whether we have encountered the problem of Section 3.3 or that of Section 3.4, we learn that our `for`-loop should skip some expression forms. Indeed, the case analysis on *Main* gives cases to prove only for expressions that are typed at the type that is the subject of the canonical form lemma or expressions that can be typed at any type such as function application and `if`.

We then refine our LNP proof as follows. We call this file `canonical3.lnp`.

```
1  for each ty in Type, Theorem Canonical-form-_(ty):
2   ... language-parameterized statement ...
3  Proof.
4  intros.
5  _ : case Main.
6  for each e in Expression:
7    if (ty = ofType(e)) or isVar(ofType(e))
8      then if e is Value
9              then search
10             else _ : case ValHyp
11     else noOp
```

The subject of each canonical form lemma is `ty`, which comes from the grammar of types. Line 7 checks that the expression form selected by the iteration has the type `ty` according to its typing rule or that its typing rule assigns a variable to it (as in function application and `if`). If this check is successful then we continue with the treatment that we have seen. Otherwise, we perform `noOp`, that is a no-operation and does not generate any proof instruction. Indeed, the case analysis of *Main* will not consider those cases.

The proofs generated by Lang-n-Prove with `canonical3.lnp` for both `stlc_iff.thm` and `stlc_iff_inverted.thm` are successful. However, a problem now arises if we add subtyping to, say, `stlc_iff.thm`. Lang-n-Prove can add declarative subtyping to `stlc_iff.thm` by declaring a specific grammar rule as follows. We call this file `stlc_iff_sub.thm`.

```
Subtyping S ::= (bool) | (arrow Contra Cov).
```

and the rest of `stlc_iff.thm` remains the same in `stlc_iff_sub.thm`. This grammar rule specifies that the first argument of the function type is contravariant (`Contra`) and its second argument is covariant (`Cov`). Based on this grammar rule, Lang-n-Prove generates the various subtyping rules (see [15] for the details of this). Most importantly for our example, Lang-n-Prove generates the subsumption rule as the last typing rule of the language

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

Case analysis on *Main* then gives an extra case to prove, that for the subsumption rule. The Abella proof generated with `canonical3.lnp` when the language in input is `stlc_iff_sub.thm` covers for all cases but this last one. The proof ends while the theorem has not been completed.

The last block of the output of Abella for this error is

```
<a name="12"></a>
<pre>
Subgoal 5:

Variables: e S
Main : {typeOf empty e bool}
ValHyp : {value e}
H1 : {typeOf empty e S}
H2 : {subtype S bool}
============================
 e = tt \/ e = ff

 Canonical-form-bool  <
```

The last block prematurely ends without `</pre>`. As the LNP proof believes that the proof has been completed, it generated code to start a new theorem at Line 12. This new theorem is `Canonical-form-arrow` in our example. However, the command `Theorem` that starts a new theorem cannot be used in the context of an ongoing proof, which expects proof instructions instead, hence the syntax error. If the incomplete proof were to be the last proof and no next theorems to prove followed, then Line 12 simply would not exist in the `.thm` file, i.e., the 11-th line would be the last line of the file.

Based on these considerations, we have developed a component for checking whether an LNP proof does not generate enough proof instructions to complete a proof. We check that either the line indicated by the last block (Line 12 in this case) does not exist in the `.thm` file or that this line contains a command that is not a proof instruction. In these cases, we can retrieve the name of theorem that was not completed from the last block. This information is highlighted above in gray color. Our debugger provides the following error message

"*The .lnp proof did not generate the proof instructions for completing the proof of theorem* `Canonical-form-bool`."

To fix `canonical3.lnp`, we need to add the proof code that handles the subsumption rule. This code calls inversion subtyping lemmas, which also must be generated. [15] covers this in details.

4 Additional Examples

This section provides a handful of examples that show our debugger at work.

Another Incorrect Treatment of Non-values in Canonical Form Lemmas (Example of contextual error message) We modify `canonical3.lnp` to wrongly perform `search` on non-values. We call this file `canonical4.lnp`. (The modification is highlighted.)

```
1  for each ty in Type, Theorem Canonical-form-_(ty):
2   ... language-parameterized statement ...
3  Proof.
4  intros.
5  _ : case Main.
6  for each e in Expression iterating Main at 2:
7     if (ty = ofType(e)) or isVar(ofType(e))
8       then if e is Value
9              then search
10             else  search
11      else noOp
```

The Abella proof generated when the input is `iff.lan` fails. Differently from the example in Section 3.3, where the problem was that the `for`-loop ended up not being in synch with Abella, here the `for`-loop is in synch but the instruction is simply incorrect for that case.

Our debugger saves contextual information also when traversing an `if`-statement. The line in the `.thm` file where Abella fails has the following comment

`search.` #10,19
-`for`:6,0:*Main*:2:(*if* $E1$ $E2$ $E3$)
-`if`:7,4:::(*bool*) = `ofType`((*if* $E1$ $E2$ $E3$)) `or IsVar(ofType`((*if* $E1$ $E2$ $E3$))
-`if`:8,14:::`not`((*if* $E1$ $E2$ $E3$) `is Value`)

where we have split entries in multiple lines for readability (whereas they are all in the same line in the `.thm` file). The entry for an `if`-statement stores the line and character number that the `if` has in the `.lnp` file. It also stores the condition check of the `if` and whether it succeeded or not (marked with `not`). As it is more informative for the user, the condition has variables already instantiated, that is, $e = (\textit{if}\ E1\ E2\ E3)$ and $ty = (\textit{bool})$ above.

From such comment, our debugger produces the message

"*Line 10, character 19 of the .lnp failed.*
The instruction is in the for-loop at line 6, character 0, and the element of the

iteration is (*if* E_1 E_2 E_3).
The instruction is in the if-statement at line 7, character 4, and the condition was `ofType`*((if* $E1$ $E2$ $E3$)*)* `or IsVar(ofType`*((if* $E1$ $E2$ $E3$)*)*
The instruction is in the if-statement at line 8, character 12, and the condition was `not`*((if* $E1$ $E2$ $E3$) `is Value`)*."

The information that the instruction that failed was inside the `if` of `not`((*if* $E1$ $E2$ $E3$) `is Value`) is a crucial information for debugging `canonical4.lnp`.

Canonical Form Lemmas, Still Wrong (Example of "`for`-loop not in synch") We add pairs to our language with functions and booleans (`stlc_iff.lan`). This new language definition contains the following.
(Only the parts that are relevant to us. We call this file `stlc_iff_pairs.lan`).

```
Expression E ::= (tt) | (ff) | (if E E E)
               | x | (app E E) | (abs T (x)E)
               | (pair E E) | (fst E) | (snd E).
Value V ::= (tt) | (ff) | (abs T R) | (pair V1 V2).

Gamma |- (pair E1 E2) : (times T1 T2) <== Gamma |- E1 : T1
                                      /\ Gamma |- E2 : T2.
```

The Abella proof generated with `canonical3.lnp` when the language in input is `stlc_iff_pairs.thm` fails. Our debugger displays the message

"*The* `for`*-loop at Line 6, character number 0 is not in synch with Abella. The case analysis was* `Main : typeOf empty (pair E1 E2) (times T1 T2)` *but the element of the iteration was* (*app* $E1$ $E2$)*.*"

This reveals that the proof code generated for (*pair* $E1$ $E2$) did not prove the case. Therefore, the instructions generated in the next iteration of the loop, which are for (*app* $E1$ $E2$), are used by Abella while still trying to prove the case for (*pair* $E1$ $E2$). This provides useful information. Of course, after reading a message such as that, we still have to look closely because the problem is then specific to the proof we are making. Here, we have generated `_ : case ValHyp` for (*pair* $E1$ $E2$) because it is not a value and the condition `e is Value` fails. (*pair* $E1$ $E2$) is not a value because we first need to know that $E1$ and $E2$ are values, too (see the grammar of values above). The correct proof does perform `_ : case ValHyp` because *ValHyp* says that (*pair* $E1$ $E2$) *is* a value and therefore we do discover that $E1$ and $E2$ are values thanks to that case analysis. (This is because the grammar of values says that (*pair* $E1$ $E2$) is a value only when $E1$ and $E2$ are values.) That `case` instruction was accidentally performed because it was borrowed from the `else`-branch that is meant for operations such as function applications and *if*, where `_ : case ValHyp` is invoking a contradiction. However, after that `case`, the proof requires `search` to close the case.

We can fix the LNP proof by taking into account that some expression forms may not immediately values, such as (*pair* $E1$ $E2$). That brings us to the LNP proof for canonical form lemmas that has been proposed in [12].

Incorrect Treatment of Elimination Forms in Progress (Example of contextual error message) The progress property ensures that a well-typed expression is a value, an error, or takes a reduction step. The proof of the progress theorem sometimes makes use of a predicate *progresses* e that says as much about the expression e and is sometimes split into two types of theorems. The first is a series of operator-specific progress theorems that formulate the progress property for each individual expression form. Some examples are the following

progress-app: $\forall T, e_1, e_2,$
$\emptyset \vdash (e_1\, e_2) : T \ \Rightarrow e_1$ *progresses* $\Rightarrow e_2$ *progresses* $\Rightarrow (e_1\ e_2)$ *progresses*

progress-head: $\forall T, e,$
$\emptyset \vdash (\texttt{head}\ e) : T \ \Rightarrow e$ *progresses* $\Rightarrow (\texttt{head}\ e)$ *progresses*

The second is a main progress theorem that invokes such operator-specific progress theorems for each expression form. When an operator-specific progress theorem is about an elimination form, its proof calls a canonical form lemma and appeals to the existence of a reduction rule for each value to handle. [12] presents an LNP proof for generating operator-specific progress theorems. Below, we modify this proof to wrongly call `search` for elimination form as if their case could be simply proved immediately. The original proof in [12] is rather involved, therefore we do not describe it and below we also show the relevant part only. We call this file `progress-op1.lnp`.

```
1  for each e in Expression, Theorem Progress-_(e) :
2   ... language-parameterized statement ...
3  Proof.
4  ...
5  if isEliminationForm(e)
6   then search
7  #  the original Line 6 calls the canonical form lemma.
```

The proof generated with `progress-op1.lnp` for `stlc_iff.thm` fails because `search` does not conclude the proof. Our debugger produces the message

"*Line 6, character 6 of the .lnp failed. The instruction is in the if-statement at line 5, character 0, and the condition was* `isEliminationForm((`*if* $E1$ $E2$ $E3$`))`."

An Incomplete Proof for Progress When Subtyping is Present The following is the LNP proof for the progress theorem in [12]. (This is what earlier we have called the "main" progress theorem.) This file is called `progress.lnp`.

```
Theorem Progress-thm :
forall e, forall typ, (Main : typeOf (empty) e typ) -> (progresses e).
Proof.
```

```
Typing_(0): induction on Main.
for each e in Expression:
  for each i in contextualArgs(e): _ : apply IH0 to Typing_(i) endfor.
  backchain on Progress-_(e)
```

The proof is by induction on *Main*. For each well-typed expression form, the inductive hypothesis derive that its arguments progress (as in the *progresses* predicate), and so we can apply the operator-specific progress theorem. We apply it with `backchain`, which concludes the proof case.

When we apply `progress.lnp` to our language `stlc_iff_sub.thm` with declarative subtyping, the Abella proof fails for the same problem that we have encountered in Section 3.5: the case analysis on *Main* now analyzes one more case for the subsumption rule. The LNP proof above, however, covers all cases but that. Our debugger produces the following message

"*The .lnp proof did not generate the proof instructions for completing the proof of theorem* `Progress-thm`."

Incorrect Loop for the Main Progress Theorem (Example of "`for`-loop not in synch") As we have pointed out, for some languages, introduction forms are also values. Their progress case can be proved with `search`. This is the case for `stlc_iff.thm` with values `tt`, `ff`, and abstraction. (Introduction forms such as (`pair` e_1 e_2) need more proof code that explores the progress of e_1 and e_2, instead). If those languages are the intended target of our LNP proofs, we may be tempted to modify `progress.thm` and prove the induction on *Main* with two `for`-loops. The first covers values and the second covers the rest of the expressions (below, this will be the iteration at Line 8 over the list difference (`Expression` − `Value`)). Below, we show the code of this incorrect attempt. We call this file `progress1.thm`.

```
1  Theorem Progress-thm :
2  forall e, forall typ, (Main : typeOf (empty) e typ) -> (progresses e).
3  Proof.
4  Typing_(0): induction on Main.
5  for each v in Value iterating Main at 2:
6   search
7  endfor.
8  for each e in Expression - Value iterating Main at 2:
9    for each i in contextualArgs(e): _ : apply IH0 to Typing_(i) endfor.
10   backchain on Progress-_(e)
```

When we apply `progress1.lnp` to our language `stlc_iff.thm`, the Abella proof fails because the induction on *Main* still includes cases for non-values of `stlc_iff.thm`, such as the function application and `if`. Our debugger detects that the Abella cases do not align with the loop and produces the following message

"*The* `for`*-loop at Line 5, character number 0 is not in synch with Abella. The case analysis was* `Main : typeOf empty (if E1 E2 E3) typ` *but the element of the iteration was* (*abs T R*)."

Our debugger does not prevent from proving a case analysis with two `for`-loops, in general. However, our debugger informs the user if the Abella proof gives to the first loop some cases that are meant for the second loop. This is the scenario above, indeed.

Correct Abella Proof but Incorrect LNP Proof (Example of "`for`-loop not in synch") Consider the language `unit.lan` with only the unit type.

```
Expression E ::= (unit).
Type T ::= (unitType).
Value V ::= (unit).

Gamma |- (unit) : (unitType).
```

Suppose that that we wanted to develop an LNP proof of the progress theorem that works on languages that contain introduction forms only. Not only so, but also where introduction forms are also values. We have discussed this type of languages in our previous examples. (Perhaps we wanted to focus on this particular class of languages first in order to progressively develop a full LNP proof.) Their characteristic is that one simple `search` proves their progress case. Consider the following LNP proof for the progress theorem. We call this file `progress2.lnp`.

```
1 Theorem Progress-thm :
2 forall e, forall typ, (Main : typeOf (empty) e typ) -> (progresses e).
3 Proof.
4 _ : induction on Main.
5  for each ty in Type  iterating Main at 2:
6   search
```

Our intention is to iterate over expression forms, but we have mistaken the loop at Line 5 and we iterate over types. Since `unit.lan` has only one type and one value, it so happens that Lang-n-Prove generates one `search` that is used for the progress case of the `unit` value. That is, Lang-n-Prove generates an Abella proof that succeeds without any problem. Our debugger, however, points out the mismatch between the loop and Abella, and produces the message

"*This generated* `.thm` *is valid, all proofs are completed.*
The `for`*-loop at Line 5, character number 0 is not in synch with Abella. The case analysis was* `Main : typeOf empty unit unitType` *but the element of the iteration was* (*unitType*)."

The user can certainly keep and use the generated Abella proof if that particular mechanization was what they sought. However, if the user cared about providing a proof that worked over the intended class of languages, then this error message indicates that the `for`-loop is not in synch with Abella.

Too Many Instructions When Subtyping Is Not Present The following is the LNP proof presented in [15] for the progress theorem, which handles the extra case of the subsumption rule. This file is called `progress_sub.lnp`.

```
Theorem Progress-thm :
forall e, forall typ, (Main : typeOf (empty) e typ) -> (progresses e).

 ... same exact code of progress.lnp ...
 followed by the following last line
backchain on IH0
```

In particular, the last case is for the typing rule for subsumption and can be proved by applying the inductive hypothesis. (We use `backchain` to close the case at once.) When we apply `progress_sub.lnp` to a language that does not have subtyping, say, `stlc_iff.thm`, then the `for`-loop in `progress.lnp` completes the proof. Yet, `progress_sub.lnp` generates the extra instruction `backchain on IH0`. Our debugger produces the following message

"*The proof of theorem* `Progress-thm` *is completed but the .lnp has generated too many instructions for it.*"

5 Limitations

Our debugging system presents some limitations, which we would like to discuss.

The directive `iterating Main at 2` is limited to point to an argument of a formula, that is arg_2 in a formula ($predname\ arg_1\ arg_2\ \ldots\ arg_n$) that is stored with hypothesis name `Main`. However, arg_2 itself may be a constructed term of the form ($op\ arg'_1\ \ldots\ arg'_m$) and the case analysis at play may actually be exploring some arg'_k by cases. Our debugger does not have the capability to specify a subterm.

To detect that `for`-loops are not in synch, we need to fetch the last field of a `for` entry of the comment of an instruction, $\cdots$: (*if* $E1\ E2\ E3$) for example, and compare it with an argument of a hypothesis, for example, the second argument of `Main : typeOf empty (if E1 E2 E3) typ`. This equality check is tricky because Abella may choose different names for variables in some proofs. This equality check is related to α-equivalence, but something more subtle happens because there may be unexplored hypotheses of the proof state that restrict the shape that variables can take. (Unexplored in the sense that if we were to do a case analysis, they would provide the actual shape of that variable.) As further investigation of this subtle problem is needed, we currently approximate this equality check, and there are `for`-loops that we do not catch as not in synch. (The inability to detect all bugs is rather common with static analyses.)

Lang-n-Prove does not only generate the proof of theorems but also provides various operations for generating the statements of such theorems. Language designers may mistake the Lang-n-Prove code for generating such theorem statements. However, our debugger does not address them and only strives to debug proofs.

A Lang-n-Prove proof may fail because the language definition in input is incorrect while the proof itself is correct. In these cases, our debugger blames some point in the proof. It would be interesting to explore ways to understand whether the language definition is at fault and point out an error therein.

Our debugger analyzes the output after the execution of Abella terminated. However, a proof may complete successfully and yet have an instruction of a `for`-loop that is not in synch. This instruction may be very early in the proof code. It would be interesting to adopt more efficient solutions that stem from the research area of *online monitoring* (see [10] for an excellent survey on the topic) where we would monitor Abella in its step-by-step execution of proof instructions and discover the not-in-synch problem immediately when it occurs (rather than letting the proof continue and analyzing the output at the end.)

6 Related Work

Great work has been done to make proof assistants more accessible and user-friendly. It is important, however, to clarify what *is not* an alternative system to ours. The error message systems that proof assistants such as Coq, Isabelle, Agda, and so on, adopt in order to pinpoint errors within one proof, are not an alternative system to ours, as those error messages are related to the broad problem of debugging a mechanized proof. Our work, on the other hand, is about *debugging the algorithms* that traverse and interrogate language definitions and that build proofs by means of that. As far as we know, the ability to express language-parameterized proofs is currently seen in Lang-n-Prove only. The closest related works to this in the realm of proof assistants are automated proof scripts. Proof assistants such as Coq, Isabelle, and others, can automate the generation of proof instructions with specific scripts. Although they do not traverse and interrogate language definitions given in input, they perform automatic proof generation. When these scripts fail, the error message that is provided is that of the failing instruction that has been generated. This is akin to the problem of Lang-n-Prove (without our debugger) that we have described in the introduction section.

There are a number of works in automated language verification. Some works use automated proving by feeding a first-order theorem prover [16], while others by using sophisticated automation in higher-order logic programming [22,25]. In these works, theorems are automated while Lang-n-Prove strives to *specify* an algorithm that works for classes of languages.

Some works have used type systems for the analysis of the type soundness of languages, both extrinsically [14] and intrinsically [11]. The latter has been especially successful [4–7,9,17,24,27]. Other works [1,3,13,20] automate rule formats [21] in order to establish bisimilarity properties of process algebras. These works do not express algorithms for generating proofs for classes of languages. They check one specific property at a time for a language given in input. The error messages they provide points to an error in the language definition for not affording the property. From this perspective, our debugger can learn much from this impressive body of work. The inability to blame language definitions is indeed one of the limitations that we have discussed in the previous section.

7 Conclusion

In this paper, we have described a debugging system for Lang-n-Prove. Our debugger detects four types of issues in the debugging of language-parameterized proofs. We have implemented our system into Lang-n-Prove, and we have illustrated its application to several examples (11 examples in total). Overall, we believe that our debugger provides useful error messages.

In the future, we would like to address the limitations that we have discussed in Section 5. We also would like to conduct a study on documenting the journey towards the development of new language-parameterized proofs through the helpful error messages of our debugger. An idea that can stress-test our debugger could be to target language-parameterized proofs for languages with time and probability, beginning with analyzing the few systems that we have experience with [2, 18, 23] as a starting point.

Our debugger is publicly available at [19].

References

1. Aceto, L., Caltais, G., Goriac, E.I., Ingolfsdottir, A.: Preg axiomatizer – a ground bisimilarity checker for gsos with predicates. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) Algebra and Coalgebra in Computer Science. pp. 378–385. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
2. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using timed rebeca. In: Mousavi, M.R., Ravara, A. (eds.) Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011, Aachen, Germany, 10th September, 2011. EPTCS, vol. 58, pp. 1–19 (2011). https://doi.org/10.4204/EPTCS.58.1, https://doi.org/10.4204/EPTCS.58.1
3. Aceto, L., Goriac, E., Ingólfsdóttir, A.: Meta SOS - A maude based SOS metatheory framework. In: Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013. pp. 93–107 (2013)
4. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Flum, J., Rodriguez-Artalejo, M. (eds.) Computer Science Logic. Lecture Notes in Computer Science, vol. 1683, pp. 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48168-0_32

5. Appel, A.W., Leroy, X.: A list-machine benchmark for mechanized metatheory: (extended abstract). Electronic Notes in Theoretical Computer Science **174**(5), 95–108 (2007). https://doi.org/10.1016/j.entcs.2007.01.020
6. Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter (1999), in Workshop on Dependent Types in Programming, Gothenburg
7. Bach Poulsen, C., Rouvoet, A., Tolmach, A., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for imperative languages. Proceedings of the ACM on Programming Languages (PACMPL) **2**(POPL) (dec 2017). https://doi.org/10.1145/3158104
8. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: A system for reasoning about relational specifications. Journal of Formalized Reasoning **7**(2), 1–89 (2014). https://doi.org/10.6092/issn.1972-5787/4650
9. Benton, N., Hur, C., Kennedy, A., McBride, C.: Strongly typed term representations in coq. Journal of Automated Reasoning **49**(2), 141–159 (2012). https://doi.org/10.1007/s10817-011-9219-0
10. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: A survey of runtime monitoring instrumentation techniques. In: Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19th September 2017. pp. 15–28 (2017). https://doi.org/10.4204/EPTCS.254.2
11. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**, 56–68 (1940). https://doi.org/10.2307/2266170
12. Cimini, M.: Lang-n-prove: A dsl for language proofs. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. pp. 16–29. SLE 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567512.3567514
13. Cimini, M.: A declarative validator for GSOS languages. In: Castellani, I., Scalas, A. (eds.) Proceedings 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2023, Paris, France, 22 April 2023. EPTCS, vol. 378, pp. 14–25 (2023). https://doi.org/10.4204/EPTCS.378.2, https://doi.org/10.4204/EPTCS.378.2
14. Cimini, M., Miller, D., Siek, J.G.: Extrinsically typed operational semantics for functional languages. In: Lämmel, R., Tratt, L., de Lara, J. (eds.) Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020. pp. 108–125. ACM (2020). https://doi.org/10.1145/3426425.3426936
15. Galasso, S., Cimini, M.: Language-parameterized proofs for functional languages with subtyping. In: Gibbons, J., Miller, D. (eds.) Functional and Logic Programming. pp. 291–310. Springer Nature Singapore, Singapore (2024)
16. Grewe, S., Erdweg, S., Wittmann, P., Mezini, M.: Type systems for the masses: Deriving soundness proofs and efficient checkers. In: Murphy, G.C., Steele Jr., G.L. (eds.) 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). pp. 137–150. Onward! 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814228.2814239
17. Harper, R., Stone, C.: A type-theoretic interpretation of Standard ML. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction: Essays in Honor of Robin Milner. MIT Press (2000). https://doi.org/10.5555/345868.345906
18. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H., Cimini, M.: Ptrebeca. Science of Computer Programming **128**(C), 22–50 (oct 2016). https://doi.org/10.1016/j.scico.2016.03.004

19. Ly, C., Mamillapalli, E.K., Cimini, M.: The lang-n-prove tool extended with a debugger. Available at the GitHub repo https://github.com/mcimini/lnpDebug (2024)
20. Mousavi, M.R., Reniers, M.A.: Prototyping SOS meta-theory in maude. Electronic Notes in Theoretical Computer Science **156**(1), 135–150 (2006)
21. Mousavi, M.R., Reniers, M.A., Groote, J.F.: Sos formats and meta-theory: 20 years after (2007)
22. Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems. In: Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction. pp. 202–206. CADE-16, Springer-Verlag, London, UK, UK (1999). https://doi.org/10.1007/3-540-48660-7_14
23. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingolfsdottir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using timed rebeca. Science of Computer Programming **89**, 41–68 (2014). https://doi.org/https://doi.org/10.1016/j.scico.2014.01.008, https://www.sciencedirect.com/science/article/pii/S0167642314000239, special issue on the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011)
24. Rouvoet, A., Bach Poulsen, C., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 284–298. ACM (2020). https://doi.org/10.1145/3372885.3373818
25. Schürmann, C.: Automating the Meta Theory of Deductive Systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (Aug 2000), http://reports-archive.adm.cs.cmu.edu/anon/2000/abstracts/00-146.html, available as Technical Report CMU-CS-00-146
26. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming **20**(01), 71–122 (2010)
27. Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming. pp. 19:1–19:15. PPDP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3354166.3354184
28. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (1994)