Well-Typed Languages are Sound

Matteo Cimini[1], Dale Miller[2], and Jeremy G. Siek[1]

[1] Indiana University Bloomington
[2] INRIA and LIX/École Polytechnique

Abstract. Type soundness is an important property of modern programming languages. In this paper we explore the idea that *well-typed languages are sound*: the idea that the appropriate typing discipline over language specifications guarantees that the language is type sound. We instantiate this idea for a certain class of languages defined using small step operational semantics by ensuring the progress and preservation theorems. Our first contribution is a syntactic discipline for organizing and restricting language specifications so that they automatically satisfy the progress theorem. This discipline is not novel but makes explicit the way expert language designers have been organizing a certain class of languages for long time. We give a formal account of this discipline by representing the language specification as (higher-order) logic programs and by giving a meta type system over that collection of formulas. Our second contribution is an analogous methodology and meta type system for guaranteeing that languages satisfy the preservation theorem. Ultimately, we have proved that language specifications that conform to our meta type systems are guaranteed to be type sound. We have implemented these ideas in the *TypeSoundnessCertifier*, a tool that takes language specifications in the form of logic programs and type checks them according to our meta type systems. For those languages that have passed our type checker, our tool could automatically produce a proof of type soundness that can be machine-checked by the Abella proof assistant. This gives us high confidence on our type systems. For those languages that fail our type checker, the tool pinpoints the design mistakes that hinder type soundness. We have applied the *TypeSoundnessCertifier* tool to a large number of programming languages, including those with recursive types, polymorphism, letrec, exceptions, lists and other common types and operators.

1 Introduction

Types and type systems play a fundamental role in programming languages. They provide programmers with abstractions, documentation, and useful invariants. The run-time behavior of programs is oftentimes a delicate and unpredictable matter. However, through the use of types and good design choices, programming languages can often ensure that during run-time, desirable properties are maintained and unpleasant behaviors are eliminated. Of all the properties that we wish to establish for typed languages, type soundness is one of the most important. Type soundness can be summarized with Robin Milner's slogan that says that *well typed programs cannot go wrong*: that is, they cannot get stuck at run-time.

In this paper we explore the idea that *well-typed languages are sound*: the idea that the appropriate typing discipline over language specifications guarantees that the language is type sound.

We instantiate this idea to a certain class of programming languages defined in small step operational semantics and we follow the approach of Wright and Felleisen. In their paper *A Syntactic Approach to Type Soundness* [19], Wright and Felleisen offered an approach to proving type soundness that has become a de facto standard and that relies on two key properties: the progress and type preservation theorems. Progress states that if a program is well-typed then it is either a value, an error, or it performs a reduction. Type preservation states that if a program has some type, a reduction step takes it to a program that has the same type.

Our first contribution is a methodology for organizing and restricting language definitions so that they automatically satisfy the progress theorem. An important aspect of the methodology is the classification of the operators of the language at hand. For example, some operators are *constructors* that build *values*, such as the functional space constructor $\lambda x.e$ in the simply typed λ-calculus (STLC). Some other operators are *eliminators*: e.g., application. Other kinds of operators are *derived operators* (such as `letrec`), *errors* and *error handlers*. The overall discipline is descriptive and simply resembles the way programming languages have been defined for a long time. For example, among other restrictions, the discipline imposes that eliminators must have reduction rules for every value allowed by the type of their argument and that those arguments that need be evaluated to a value must be set as evaluation contexts.

In our formalization of this descriptive methodology, we represent language specifications using logic programs. This is a convenient choice since, as has been argued long ago by Schürmann and Pfenning [15], such specifics are executable, correspond closely with pen & paper specifications, and have a formal semantics that can be the subject of proofs. We give a meta type system over language specifications that directly imposes the mentioned discipline. To make an example, the β rule $(\lambda x.e)\ v \longrightarrow e[v/x]$ can be type checked in the following way. (The application operator is named here as *app*.)

$$\frac{\begin{array}{c}\Gamma(app) = \text{elim} \ \rightarrow \qquad \Gamma(\lambda) = \text{value} \ \rightarrow \ \emptyset \\ \{1,2\} \subseteq ctx(app)\end{array}}{ctx \mid \Gamma \vdash (app \ \ (\lambda x.\ e) \ \ v) \rightarrow e[v/x] \ : app : \text{eliminates } \lambda}$$

That is, the rule is well-typed because the application is an eliminator of the function type and its *eliminating argument*, high-lighted, is a value of the function type. Moreover, the arguments at positions 1 and 2 must be evaluation contexts for the application. The meta typing rule assigns the type "app : eliminates λ" so that the type system has a means to check later whether *app* eliminates all the values of $\rightarrow$, which, in this case, is just the function.

The type preservation theorem is not, generally speaking, ensured by a discipline. However, typing is markedly happening. For the β rule we have to ensure that the type of $(\lambda x.e)\ v$ is the same as the type of $e[v/x]$. However, these are

stlc_cbv.mod:

```
module stlc_cbv.

typeOf (abs T1 E)  (arrow T1 T2) :- (pi x\ typeOf x T1 => typeOf (E x) T2).
typeOf (app E1 E2) T2            :- typeOf E1 (arrow T1 T2), typeOf E2 T1.
typeOf tt bool.
typeOf ff bool.
typeOf (if E1 E2 E3) T :- typeOf E1 bool, typeOf E2 T, typeOf E3 T.
step   (app (abs T E) V) (E V)    :- value V.
step (if tt E1 E2) E1.
step (if ff E1 E2) E2.
value  (abs T1 R2).
value tt.
value ff.

% context app E e
% context app v E.
% context if E e e.
```

Fig. 1. Example input of the *TypeSoundnessCertifier*: file stlc_cbv.mod. This is the language specification of STLC with the `if` operator.

expressions with variables and their types depend on the type assumptions on their variables: that is, they depend on Γ of typing judgments. Ideally, we need to check that for all Γ, if $\Gamma \vdash (\lambda x.e)\ v : T$ then $\Gamma \vdash e[v/x] : T$. Such a statement is prohibitive to check due to the quantification over all Γs. Nonetheless we are able to offer a methodology for type preservation. The methodology fixes a *symbolic type environment* Γ^s based on the information extracted from the typing rules on which the expressions of β rely on. We take a practical approach by representing Γ^s as a conjunction of typing formulae and use entailment for checking that the types of $(\lambda x.e)\ v$ and $e[v/x]$ agree. For example, by inspecting the typing rules for application and abstraction we build and check the formula

$$\vdash\ v : T_1\ \wedge\ (x : T_1 \vdash\ e : T_2)\ \Rightarrow\ \vdash\ e[v/x] : T_2.$$

This approach fits naturally a type system formulation. Analogously to the case of progress, we devise a meta type system for languages that automatically satisfy the type preservation theorem.

Ultimately, we have proved that languages that conform to our meta type systems satisfy both progress and type preservation. This validates the methodologies in this paper and, possibly, proves that the invariants that language designers have been using for a long time are correct. As a consequence of our results, language specifications that type check successfully are guaranteed to be type sound: hence the slogan *well-typed languages are sound*.

Based on our results, we have implemented the *TypeSoundnessCertifier* tool. The tool works with language specifications such as that in `stlc_cbv.mod` of Figure 1. This file contains the formulation of the STLC with the `if` operator. The specification language is that of λProlog [9] augmented with convenient context tags for declaratively specifying evaluation contexts. The *TypeSoundnessCertifier* tool can input this file and type check the language specification according to the meta type systems devised in this paper. If type checking succeeds, the tool automatically generates the theorems and proofs for the progress, type preser-

vation and ultimately type soundness theorems that are machine-checked by the Abella [2] proof assistant (which can load and reason with such λProlog specifications). If type checking fails, the tool reports a meaningful error to the user. Were we to forget the tag at line 15 (`% context app E e.`), the *TypeSoundness-Certifier* would reject the specification and tell the user that the first argument of the application must be an evaluation context. Were we to forget one of the reduction rules for `if`, say line 10, the type checking would fail reporting that this eliminator for `bool` does not eliminate *all* the values of type `bool`.

In summary, this paper makes the following contributions.

(1) We offer a complete methodology for ensuring the type soundness of languages (Sections 3, 4 and 5). The target of our methodology is a class of languages that is based on constructors/eliminators and errors/error handlers, that is common in programming languages design. This class of languages is fairly expressive and accommodates modern features such as recursive types, polymorphism, and exceptions.

(2) We formulate the methodology as a meta type system over language specifications (Section 6 and 7). We have proved that our meta type system guarantees the type soundness of languages (Section 8). This validates the common practice that language designers have been used for long and demonstrates the idea that *well-typed languages are sound*.

(3) We implemented the *TypeSoundnessCertifier* tool that can certify a language as being type sound or it can pinpoint design mistakes (Section 9). We have applied our tool to the type checking of several languages, including variants of STLC and its implicitly typed version with the following features: pairs, `if`–`then`–`else`, lists, sums, unit, tuples, `fix`, `let`, `letrec`, universal types, recursive types and exceptions. We have also considered different evaluation strategies among call-by-value, call-by-name and a parallel reduction strategy, as well as lazy pairs, lazy lists and lazy tuples. In total, we have type checked 103 type sound languages. *TypeSoundnessCertifier* has automatically generated proof of progress and preservation for each of the type checked languages and these proofs have been independently check by an external proof checker. This gives us high confidence in our type systems.

Our tool has not been made public yet. For the vision of interested reviewers, the tool will remain available at the link: `http://cimini.info/esop2017/`.

In the next section, we briefly review some terminology in the context of typed languages that are defined in small step operational semantics.

2 Typed Languages

Let us consider the language `Fpl` defined in Figure 2. This language is a fairly involved programming language with integers, booleans, `if`–`then`–`else`, sums, lists, universal types, recursive types, `fix`, `letrec` and exceptions.

Types and expressions are defined by a BNF grammar. Next, language designers decide which expressions constitute *values*. These are the possible results

Types	T	::=	$\texttt{Bool} \mid \texttt{Int} \mid T \to T \mid \texttt{List}\ T \mid T + T$
Expressions	e	::=	$\texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
			$\mid \texttt{z} \mid \texttt{succ}\ e \mid \texttt{pred}\ e \mid \texttt{isZero}\ e$
			$\mid x \mid \lambda x.e \mid e\ e$
			$\mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\,(x)\,e\,e\,e$
			$\mid \texttt{nil} \mid \texttt{cons}\ e\ e \mid$
			$\mid \texttt{head}\ e \mid \texttt{tail}\ e \mid \texttt{isNil}\ e$
			$\mid \Lambda X.\ e \mid e\ [T]$
			$\mid \texttt{fold}\ e \mid \texttt{unfold}\ e$
			$\mid \texttt{fix}\ e \mid \texttt{letrec}\ x = e\ \texttt{in}\ e$
			$\mid \texttt{raise}\ e \mid \texttt{try}\ e\ \texttt{with}\ e$
Values	v	::=	$\texttt{true} \mid \texttt{false} \mid \texttt{z} \mid \texttt{succ}\ v \mid \lambda x.e \mid \texttt{nil} \mid \texttt{cons}\ v\ v$
			$\mid \texttt{inr}\ v \mid \texttt{inl}\ v \mid \Lambda X.\ e \mid \texttt{fold}\ v$
			$\mid \texttt{nil} \mid \texttt{cons}\ v\ v \mid \texttt{fold}\ v$
Errors	er	::=	$\texttt{raise}\ v$
Contexts	E	::=	$\texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e$
			$\mid \texttt{succ}\ E \mid \texttt{pred}\ E \mid \texttt{isZero}\ E$
			$\mid E\ e \mid v\ E$
			$\mid \texttt{inl}\ E \mid \texttt{inr}\ E \mid \texttt{case}\,(x)\,E\,e\,e$
			$\mid \texttt{cons}\ E\ e \mid \texttt{cons}\ v\ E \mid \texttt{head}\ E \mid \texttt{tail}\ E \mid \texttt{isNil}\ E$
			$\mid e\ [T] \mid \texttt{fold}\ E \mid \texttt{unfold}\ E \mid \texttt{fix}\ E \mid \texttt{letrec}\ x = E\ \texttt{in}\ e$
			$\mid \texttt{raise}\ E \mid \texttt{try}\ E\ \texttt{with}\ e$

Error Contexts, F, are just Contexts but without the (`try` E `with` e) case.

Fig. 2. The syntax of `Fpl` contains a number of features that are all handled by our analysis. This language is not *minimal* since, for example, recursive types can define booleans and lists. $\texttt{case}\,(x)\,e\,e\,e$ is short for $\texttt{case}\,e\,\texttt{of}\,\texttt{inl}\,x \Rightarrow e \mid \texttt{inr}\,y \Rightarrow e$.

of successful computations. Similarly, the language designer may define which expressions constitute *errors*, which are possible outcomes of computations when they fail.

The top part of Figure 3 shows the type system for `Fpl`. The type system is an inference rule system for judgements that, in this paper, have the form $\Gamma \vdash e : T$. A term that is constructed with the application of a type constructor to distinct variables is called a *constructed type*. For example, $\texttt{List}\ T$ and $T_1 \to T_2$ are constructed types. `Int` is a constructed type as well because it is simply a type constructor with arity 0. Expressions like $\texttt{fold}\ e$ and $\texttt{cons}\ e_1\ e_2$ are *constructed expressions*. Given a typing rule such as $\dfrac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \texttt{List}\ T}{\Gamma \vdash\ \texttt{cons}\ e_1\ e_2 : \texttt{List}\ T}$ we say that the high-lighted $\texttt{List}\ T$ is the *assigned type*.

The bottom part of Figure 3 defines the dynamic semantics of `Fpl`. It is defined by a series of *reduction rules*. For a formula $e \longrightarrow e'$, e is the *source* and e' is the *target* of the reduction. In a reduction rule such as (R-HEAD-CONS), i.e. $\texttt{head}\ (\texttt{cons}\ v_1\ v_2) \longrightarrow v_1$, we say that the first argument of `head` is *pattern-matched* against the constructed expression $(\texttt{cons}\ v_1\ v_2)$.

The dynamic semantics of a language is also defined by its *evaluation contexts*, which prescribe within which context we allow reduction to take place. They are defined with the syntactic category Context of Figure 2. For a context definition such as $\texttt{cons}\ E\ e$ we say that the first argument of `cons` is *contextual*.

Error contexts define which contexts are allowed to make the whole computation fail when we spot an error.

We repeat the statement of type soundness. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Type System $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma, x : T \vdash x : T \quad \Gamma \vdash \texttt{true} : \texttt{Bool} \quad \Gamma \vdash \texttt{false} : \texttt{Bool}}{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ (\text{T-IF})$$

$$(\text{T-Z})\ \Gamma \vdash \texttt{z} : \texttt{Int} \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{succ}\ e : \texttt{Int}}\ (\text{T-SUCC}) \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{pred}\ e : \texttt{Int}}\ (\text{T-PRED}) \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{isZero}\ e : \texttt{Bool}}\ (\text{T-ISZERO})$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2}\ (\text{T-LAMBDA}) \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ (\text{T-APP})$$

$$(\text{T-NIL})\ \Gamma \vdash \texttt{nil} : \texttt{List}\ T \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \texttt{List}\ T}{\Gamma \vdash \texttt{cons}\ e_1\ e_2 : \texttt{List}\ T}\ (\text{T-CONS})$$

$$\frac{\Gamma \vdash e : \texttt{List}\ T}{\Gamma \vdash \texttt{head}\ e : T}\ (\text{T-HEAD}) \qquad \frac{\Gamma \vdash e : \texttt{List}\ T}{\Gamma \vdash \texttt{tail}\ e : \texttt{List}\ T}\ (\text{T-TAIL}) \qquad \frac{\Gamma \vdash e : \texttt{List}\ T}{\Gamma \vdash \texttt{isNil}\ e : \texttt{Bool}}\ (\text{T-ISNIL})$$

$$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash \texttt{inl}\ e : T_1 + T_2} \qquad \frac{\Gamma \vdash e : T_2}{\Gamma \vdash \texttt{inr}\ e : T_1 + T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 + T_2 \quad \Gamma, x : T_1 \vdash e_2 : T \quad \Gamma, x : T_2 \vdash e_3 : T}{\Gamma \vdash (\texttt{case}\ e_1\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_2 \mid \texttt{inr}\ y \Rightarrow e_3) : T}\ (\text{T-CASE})$$

$$\frac{\Gamma, X \vdash e : T}{\Gamma \vdash \Lambda X.e : \forall X.T}\ (\text{T-ABST}) \qquad \frac{\Gamma \vdash e : \forall X.T_2}{\Gamma \vdash (e\ [T_1]) : T_2[T_1/X]}\ (\text{T-APPT})$$

$$\frac{\Gamma \vdash e : T[\mu X.T/X]}{\Gamma \vdash \texttt{fold}\ e : \mu X.T} \qquad \frac{\Gamma \vdash e : \mu X.T}{\Gamma \vdash \texttt{unfold}\ e : T[\mu X.T/X]}$$

$$\frac{\Gamma \vdash e : T \to T}{\Gamma \vdash \texttt{fix}\ e : T}\ (\text{T-FIX}) \qquad \frac{\Gamma, x : T_1 \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 : T_2}\ (\text{T-LETREC})$$

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{raise}\ e : T}\ (\text{T-RAISE}) \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \texttt{Int} \to T}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : T}\ (\text{T-TRY})$$

Dynamic Semantics $\boxed{e \longrightarrow e}$

$$\begin{aligned}
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 &\longrightarrow e_1 && (\text{R-IF-TRUE})\\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 &\longrightarrow e_2 && (\text{R-IF-FALSE})\\
\texttt{pred z} &\longrightarrow \texttt{raise z} && (\text{R-PRED-ZERO})\\
\texttt{pred}\ (\texttt{succ}\ e) &\longrightarrow e && (\text{R-PRED-SUCC})\\
\texttt{isZero z} &\longrightarrow \texttt{true} && (\text{R-ISZERO-ZERO})\\
\texttt{isZero}(\texttt{succ}\ e) &\longrightarrow \texttt{false} && (\text{R-ISZERO-SUCC})\\
(\lambda x.e)\ v &\longrightarrow e[v/x] && (\text{R-APP-LAMBDA})\\
\texttt{head nil} &\longrightarrow \texttt{raise z} && (\text{R-HEAD-NIL})\\
\texttt{head}\ (\texttt{cons}\ v_1\ v_2) &\longrightarrow v_1 && (\text{R-HEAD-CONS})\\
\texttt{tail nil} &\longrightarrow \texttt{raise}\ (\texttt{succ z}) && (\text{R-TAIL-NIL})\\
\texttt{tail}\ (\texttt{cons}\ v_1\ v_2) &\longrightarrow v_2 && (\text{R-TAIL-CONS})\\
\texttt{isNil}\ (\texttt{nil}) &\longrightarrow \texttt{true} && (\text{R-ISNIL-NIL})\\
\texttt{isNil}\ (\texttt{cons}\ v_1\ v_2) &\longrightarrow \texttt{false} && (\text{R-ISNIL-CONS})\\
\texttt{case}\ (x)\ (\texttt{inl}\ v)\ e_2\ e_3 &\longrightarrow e_2[v/x_1] && (\text{R-CASE-INL})\\
\texttt{case}\ (x)\ (\texttt{inr}\ v)\ e_2\ e_3 &\longrightarrow e_3[v/x_1] && (\text{R-CASE-INR})\\
\texttt{unfold}\ (\texttt{fold}\ v) &\longrightarrow v && (\text{R-UNFOLD-FOLD})\\
\texttt{fix}\ v &\longrightarrow v\ (\texttt{fix}\ v) && (\text{R-FIX})\\
\texttt{letrec}\ x = v\ \texttt{in}\ e &\longrightarrow e[(\texttt{fix}\ (\lambda x.v))/x] && (\text{R-LETREC})\\
\texttt{try}\ v\ \texttt{with}\ e &\longrightarrow v && (\text{R-TRY})\\
\texttt{try}\ (\texttt{raise}\ v)\ \texttt{with}\ e &\longrightarrow (e\ v) && (\text{R-TRY-RAISE})
\end{aligned}$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ (\text{CTX}) \qquad F[er] \longrightarrow er\ (\text{ERR-CTX})$$

Fig. 3. The static and dynamic semantics of `Fpl`.

Type Soundness Theorem:
for all expressions e, e', and types T,
if $\emptyset \vdash e : T$ and $e \longrightarrow^* e'$ then either
- *e' is a value,*
- *e' is an error, or*
- *there exists e'' such that $e' \longrightarrow e''$.*

Intuitively, when programs are well-typed they end up in a value or an error, or the computation is simply not finished and continues. A well-typed program does not get stuck in the middle of a computation, that is, *well-typed programs cannot go wrong* (Robin Milner [10]).

3 A Classification of the Operators

A definition of a typed language such as that of Figure 2 does not make important distinctions between the role of operators. Indeed, `cons`, `unfold` and `try` are grouped together within the same syntactic category Expressions, even though they play a very different role within the language. Operators can be classified in *constructors*, *eliminators*, *derived operators*, and *error handlers*.

In this section, we show a method for classifying operators into these classes. This method will be employed in Section 7 to automatically classify operators for language specifications given as input.

Constructors Some operators of the language build values of a certain type. Those operators are called *constructors*. We recognize them by the following characteristics.

> *Constructors have a typing rule whose assigned type is a constructed type. Each constructor build one value and each value is built by a constructor. Also, constructors have no reduction rules.*

In `Fpl`, `true` and `false` are constructors for the type `Bool`. $\lambda x.e$ is constructor for the type $\rightarrow$, and `nil` and `cons` e e are constructors for the type `List`, to name a few examples for `Fpl`.

Eliminators Eliminators can manipulate values of some type. For example, `head` e extracts the first element of the list e when e is reduced to a value. Some other operators simply inspect the identity of a value such as `if` operator. Eliminators have the following characteristics.

> *The typing rule of eliminators assigns a constructed type to one of their arguments: this argument is called the* eliminating argument*. In all the reduction rules for eliminators, the eliminating argument is pattern-matched against a value. For convenience, we say that the rule* eliminates that argument*.*

For example, the eliminating argument of `if` is the first and we say that (R-IF-TRUE) eliminates the first argument.

Derived Operators Some operators are not involved in manipulating values at a primitive level. This is the case of operators such as `fix` and `letrec`, for example. These operators are called *derived operators*. Derived operators have the following characteristics.

> *Derived operators have at least one reduction rule. Also, none of their reduction rules pattern-matches against a constructed expression.*

Error Handlers It is often useful to capture an error produced by a computation and trigger some remedial computation. To this end, programming languages with errors are sometimes augmented with operators that can recognize the occurrence of errors and act accordingly. These latter operators are *error handlers*. One of the most notable examples in programming languages, also present in `Fpl`, is `try`. Error handlers have the following characteristics.

> *Error handlers have at least one reduction rule in which one of its arguments pattern-matches against an error. Analogously to eliminators, we call this argument the* eliminating argument*.*

Common Patterns Outside of the classification of operators, languages typically follow some common patterns for the sake of good design and type soundness.

A value definition such as Values ::= ... | `cons` v v tells us that the operator `cons` can build a value only under some condition: that its two arguments are evaluated to values. These are *valuehood requirements* that dictate when the definition can be applied. Valuehood requirements are used in error definitions (see Errors ::= `raise` v), context definitions (for example, Contexts ::= v E) and also for firing reduction rules (see `fix` $v \longrightarrow v$ (`fix` v)). We adopt the following

> ***P-Val:*** *Value, error, and context definitions, as well as the firing of reduction rules can depend only on valuehood requirements.*

Also, languages typically conform to the following restrictions

> ***P-NoStep:*** *Values and the error do not have reduction rules.*

> ***P-Typ:*** *Each operator has one typing rule and this typing rule assigns a type to each argument of the operator.*

4 A Discipline for the Progress Theorem

In this section we spell out a methodology for ensuring the validity of the progress theorem. We first repeat its statement below.

An expression e progresses *whenever either e is a value, e is an error, or there exists e' such that $e \longrightarrow e'$.*

PROGRESS THEOREM:
For all expressions e and types T,
if $\emptyset \vdash e : T$ then e progresses.

We list the items of the methodology below as a convenient reference. Each item, except for **D0** which has been addressed, is described in detail in the following subsections.

D0 Classify the operators of the language in *constructors*, *eliminators*, *derived operators*, *error handlers* and follow the common patterns as described in Section 3.
D1 Progress-dependent arguments are contextual (this type of arguments is defined in Section 4.1).
D2 Error contexts are evaluation contexts minus the error handler in the eliminating argument.
D3 The context declarations have no circular dependencies.
D4 Each eliminator of a type eliminates all the values of that type.
D5 Error handlers have a reduction rule that is defined for values at their eliminating argument.

4.1 D1. Progress-dependent Arguments

Consider the following definitions and reduction rules.

$$\begin{aligned}
\text{Values} &::= \texttt{cons}\ v\ v \mid \texttt{fold}\ v \\
\text{Errors} &::= \texttt{raise}\ v \\
\text{Contexts} &::= v\ E \\
\texttt{fix}\ v &\longrightarrow v\ (\texttt{fix}\ v) && \text{(R-FIX)} \\
(\lambda x.e)\ v &\longrightarrow e[v/x] && \text{(BETA)} \\
\texttt{try}\ (\texttt{raise}\ v)\ \texttt{with}\ e &\longrightarrow (v\ e) && \text{(R-TRY-RAISE)}
\end{aligned}$$

In all the cases above some arguments are under the restriction to be values or the error (with (R-TRY-RAISE)). This is true also for (BETA) w.r.t. the eliminating argument, where a value is syntactically pattern-matched.

These arguments need to be evaluated so that they become a value or error to enable the definition or reduction rule to apply. Therefore, they need to be in evaluation contexts. For example, since the argument of `fix` is required to be a value for (R-FIX) to fire, `Fpl` automatically needs to have the context Context ::= `fix` E. Were the language to miss such context, the expression `fix` (`head` (`cons` $\lambda x.x$ `nil`)), which is not a value nor an error, would be stuck.

We call these arguments *progress-dependent arguments*. The way to identify them is the following.

> *Arguments that are required to be values in value, error and contexts definitions are progress-dependent.*
> *Arguments in source of reduction rules that are required to be values are progress-dependent.*
> *Eliminating arguments are progress-dependent.*

D1 *Evaluation contexts include all the progress-dependent arguments.*

Notice that **D1** leaves the possibility of evaluation contexts for arguments that are not progress-dependent. Consider for example a λ-calculus with contexts $\mathit{Context} ::= (E\ e) \mid (e\ E)$, that is, the application evaluates its two arguments in parallel. Also consider the reduction rule $\beta' = (\lambda x.e_1)\ e_2 \longrightarrow e_1[e_2/x]$. The first argument is certainly a progress-dependent argument while the second, not encountering any restriction, is not. In this case, whether the second argument is contextual or not does not affect type soundness because a reduction happens either way thanks to β' or a contextual step on the first argument.

4.2 D2. Error Contexts

Language designers define the error contexts. However, not every error context is suitable. The following is a general rule.

> **D2** *Error contexts are evaluation contexts minus the error handler at the eliminating argument.*

Error contexts do not contain the error handler at the eliminating argument for the sake of good design. Indeed, `try` (`raise` e_1) `with` $e_2 \longrightarrow$ `raise` e_1 should not take place, as we expect the semantics of `try` to handle the error.

All other evaluation contexts are error contexts for the sake of good design and for type soundness. Since the error handler is the only operator expecting an error, all other progress-dependent arguments expect a value (by **P-Val**). Therefore, they have no reduction rule for handling the encounter of the error. For example, `succ` (`raise` v) would be stuck if it were not for the error context `succ` F that enables the reduction `succ` (`raise` v) $\longrightarrow$ (`raise` v). Evaluation contexts that are defined for no progress-dependent argument do not strictly need to be in error contexts. For example, for the parallel λ-calculus of the previous section the expression e (`raise` v) does not get stuck because another reduction rule fires anyway. However, evaluation contexts are chosen as such by the language designer because those are *observable* parts of the computation, hence **D2** is the general rule at play.

Finally, error contexts should *only* be evaluation contexts. For example, the reduction `if true then` e `else` (`raise` v) $\longrightarrow$ (`raise` v) should not take place.

4.3 D3. Context Declarations

A Problem with Dependencies Consider the bad context declarations Context ::= `cons` E v | `cons` v E. In this case, the expression `cons` $((\lambda x.x)\ 5)$ $((\lambda x.x)$ `nil`$)$ is simply stuck because the first argument $((\lambda x.x)\ 5)$ waits for the second to be evaluated to a value, and in the meantime $((\lambda x.x)$ `nil`$)$ is not being evaluated because it waits for the first argument to be reduced to a value. Circular dependencies in context declarations jeopardize the type soundness of the language. Therefore,

> **D3** *Evaluation contexts must not have circular dependencies.*

An easy way to check for **D3** is through a graph representation of the dependencies at play. To be precise, for each declaration we have an edge from the index position of E to the index position of a v. `Fpl` has correct context declarations for `cons` because the declarations induce the graph $\{2 \mapsto 1\}$ which is acyclic. The bad context declarations above induce the graph $\{1 \mapsto 2, 2 \mapsto 1\}$, which contains a cycle.

4.4 D4. Eliminators

D4 *For each eliminator of a type T, each value of type T is eliminated by a reduction rule of the eliminator.*

As an example, let us consider the example of the `head` operator.

$$\texttt{head}\ \texttt{nil} \longrightarrow \texttt{raise}\ \texttt{true} \qquad \text{(R-HEAD-NIL)}$$

$$\texttt{head}\ (\texttt{cons}\ v_1\ v_2) \longrightarrow v_1 \qquad \text{(R-HEAD-CONS)}$$

Were we to miss the rule (R-HEAD-NIL), the expression (`head nil`) would be stuck for failing in finding a rule for performing a reduction. As this expression is not a value nor an error, type soundness would be jeopardized.

4.5 D5. Error Handlers

D5 *Error handlers have a reduction rule that is defined for values at their eliminating argument.*

In `Fpl`, the error handler `try` cannot afford to define its step only at the encounter of the error, or an expression such as `try z with` $\lambda x.x$ would be stuck. **D5** imposes that a reduction rule such as

$$\texttt{try}\ v\ \texttt{with}\ e \longrightarrow v \qquad \text{(R-TRY)}$$

exists. Notice that the rule expects a value. Indeed, we should forbid rules such as `try` e_1 `with` $e_2 \longrightarrow e_3$ which apply unrestricted. As the error is also an expression, this rule can non-deterministically preempt the application of the rule that specifically handles the error.

5 Type preservation

We now devise a methodology for checking the validity of the type preservation theorem. First, we repeat the statement of the theorem.

> TYPE PRESERVATION THEOREM :
> *for all expressions e, e' and types T,*
> *if $\emptyset \vdash e : T$ and $e \longrightarrow e'$ then $\emptyset \vdash e' : T$*

Given a reduction rule $e \longrightarrow e'$, we have to ensure that the types of e and e' coincide. However, the rule is defined with variables to be applied to a plurality of expressions. Ideally, we need to check that

$$\text{for all } \Gamma,\ \Gamma \vdash e : T \text{ implies } \Gamma \vdash e' : T.$$

Of course, checking all possible type environments is prohibitive. Therefore, our approach approximates such a check with the use of a *symbolic type environment*. We form symbolic type environments out of the typing rules of operators. For convenience, we simply use the typing premises that we encounter in those rules. This choice accommodates well the fact that typing premises rely on typing assumptions themselves. Consider for example the premise $\Gamma, x : T_1 \vdash\ e : T_2$ of (T-ABS) and $exp = \lambda x.v$. Variables have two levels. Typing exp depends on v, which is the logical variable of the typing rule and ranges over expressions. In turn, after v is instantiated, it contains a particular variable x of the object language, and the type of v depends on this variable. To account for this, the symbolic type environment employs hypothetical typing formulae. For example, the symbolic type environment extracted for exp is $(\Gamma, x : T_1 \vdash\ \ v : T_2)$. The presence of hypothetical typing formulae is axiomatized by the following equation.

$$\frac{\Gamma \vdash e' : T_1}{\Gamma, x : T_1 \vdash\ e : T_2 \equiv\ \Gamma \vdash e[e'/x] : T_2} \qquad \text{(EQ-SUB)}$$

Given a reduction rule, we give a means to compute both the symbolic type environment and its symbolic assigned type. There are two steps for those reduction rules that eliminate an argument ((1) and (2) below) and one step for any other reduction rule (only (1)).

> *(1) Instantiate the typing rule that types the source of the reduction rule.*
> *(2) Instantiate the typing rule that types the eliminated argument of the reduction rule, if it is a constructed expression. The symbolic type environment contains the typing formulae of the premises of the two rules combined. The symbolic assigned type is that of (1).*

With this main ingredient, we can offer a methodology for type preservation. For each reduction rule, apply the following.

> *Construct the symbolic type environment Γ^s of the rule and its symbolic assigned type T. Check whether Γ^s entails that the target of the reduction rule has the same type T.*

We shall see a few examples. Consider the case of `head` and its elimination rule `head` (`cons` v_1 v_2) $\longrightarrow v_1$. We have given the color blue to the target so that later it will be clear where a particular occurrence of v_1 comes from. Instantiating the typing rules (T-HEAD) and (T-CONS) in the way prescribed by (1) and (2), respectively, gives us the following rules.

$$\frac{\Gamma \vdash (\texttt{cons}\ v_1\ v_2) : \texttt{List}\ T}{\Gamma \vdash \texttt{head}\ (\texttt{cons}\ v_1\ v_2) : T} \qquad \frac{\Gamma \vdash v_1 : T \quad \Gamma \vdash v_2 : \texttt{List}\ T}{\Gamma \vdash \texttt{cons}\ v_1\ v_2 : \texttt{List}\ T}$$

The assigned type is the red T in the first rule. For the symbolic type assignment, we collect the typing premises of the two rules. We can restrict ourselves to collect only the typing rules for variables. Indeed, the typing premise of the eliminated argument, such as $\Gamma \vdash (\texttt{cons}\ v_1\ v_2) : \texttt{List}\ T$, is always derivable for it has been unfolded in the second rule. For the case above, we have $\Gamma^s = v_1 : T, v_2 : \texttt{List}\ T$. Finally, we need to check that $\Gamma^s \vdash v_1 : T$. This fact can be trivially established. This means that Γ^s, which can type $\texttt{head}\ (\texttt{cons}\ v_1\ v_2)$ at T, can also type v_1 at T.

Let us now see the example of (BETA): $(\lambda x.e)\ v \longrightarrow e[v/x]$. The instantiations (1) and (2) give us the following rules.

$$\frac{\Gamma \vdash \lambda x.e : T_1 \to T_2 \quad \Gamma \vdash v : T_1}{\Gamma \vdash ((\lambda x.e)\ v) : T_2} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2}$$

In this case, the symbolic type environment is $(\Gamma, x : T_1 \vdash e : T_2), \Gamma \vdash v : T_1$. We finally need to check $\Gamma^s \vdash e[v/x] : T_2$, which can be established using (EQ-SUB).

A Requirement for the Error In a language with the error and error contexts, we enforce that

D-Err *The error must be typed at any type.*

This is necessary because the error travels through contexts with the rule $F[er] \longrightarrow er$, for any context F. For the sake of type preservation, wherever the error lands it must be prepared to match the type of the expression it replaces.

So far, we have spelled out a descriptive methodology for ensuring both the progress and preservation theorem. It is easy to check that it applies to `Fpl` in full. This is a non-trivial language with modern features such as recursive types, polymorphism and exceptions. Now that we have described the methodology in detail we can embark on formalize it as a typing discipline and prove it correct.

6 Typed Languages as Logic Programs

We now proceed to give the methodologies of the previous sections a formal counterpart. To this aim, we first need a formal representation for language specifications that can be manipulated and be the subject of proofs. We represent them as logic programs in the higher-order intuitionistic logic. This logic has a solid theoretical foundation, is executable and is the basis of the λProlog

programming language. Higher-order logic programs turn out to be a convenient medium for our endeavors because they are in close correspondence to pen&paper language definitions.

Logic programs are equipped with a *signature*, which is a set of *declarations* for the entities that are involved in the specifications. For example, the following is a partial signature for `Fpl`.

$$\begin{array}{l}\mathsf{exp}\,,\mathsf{type}\,:\mathsf{kind}\\ \mathsf{arrow}\,:\mathsf{type}\,\to\mathsf{type}\,\to\mathsf{type}\\ \mathsf{abs}\,:\mathsf{type}\,\to(\mathsf{exp}\,\to\mathsf{exp}\,)\to\mathsf{exp}\\ \mathsf{app}\,:\mathsf{exp}\,\to\mathsf{exp}\,\to\mathsf{exp}\end{array}$$

The schema variable Σ ranges over signatures. The constant o is the type of propositions. To help the presentation, we sometimes shall use symbols rather than names and have declarations $\vdash\,:\mathsf{exp}\,\to\mathsf{type}\,\to\mathsf{o}$ and $\to\,:\mathsf{exp}\,\to\mathsf{type}\,\to\mathsf{o}$ for a typing and a reduction predicate, respectively.

When we represent program expressions as types, we shall use the familiar setting of higher order abstract syntax (HOAS) to encode bindings. That is, binders in program expressions will be mapped directly to binders in terms. For example, the declaration of the abstraction abs above takes two parameters, of which the second is an abstraction of the logic. The identity function $\lambda x{:}\mathtt{Bool}.\,x$ is then encoded as $(\mathsf{abs}\ \mathtt{Bool}\ \lambda x.\,x)$.

The *terms* of higher-order logic are based on the usual notion of simply typed λ-terms over a signature. Given a signature Σ, a (higher-order intuitionistic logic) formula P over Σ is any formula built from implications and universal quantifier and atomic formulas. We shall represent logic programming rules ϕ in the form

$$\frac{P_1\ \ldots\ P_n}{P}$$

In higher-order logic programs, the use of universal and implicational formulae in premises enable generic and hypothetical reasoning. Their role in language specification can be described with the example of the following typing rule for the abstraction operator abs .

$$\frac{(\forall x.\vdash x\ T_1\Rightarrow\vdash(E\ x)\ T_2)}{\vdash(\mathsf{abs}\ T_1\ E)\ (\mathsf{arrow}\ T_1\ T_2)}$$

The universal quantification $\forall x$ introduces a new variable x encoding a program term and the implication temporarily augment the logic programs with the fact $\vdash x\ T_1$ while proving $\vdash(E\ x)\ T_2$. Therefore, the explicit type environment Γ, which encodes the typing information assumed along the way, is not necessary.

Our notion of typed languages is based on the following standard definition of logic programs.

Definition 1 (Logic Programs). *A* logic program *is a pair (Σ, D) where Σ is a signature and D is a set of rules over Σ. A query q (which can be any logical formula) follows from a logic program, written $(\Sigma, D) \models P$, if P is provable from D in intuitionistic logic.*

As we have seen in the previous sections, typed languages also rely on evaluation and error contexts. We define *context summaries* as a declarative means for their specification. Intuitively, the contexts: $\texttt{head}\ E \mid \texttt{raise}\ E \mid E\ e \mid v\ E \mid \texttt{cons}\ E\ e \mid \texttt{cons}\ v\ E$ are modeled with a function ctx such that

$$\begin{aligned} ctx(\textsf{head}\ ) = ctx(\textsf{raise}\ ) &= \{(1, \emptyset)\} \\ ctx(\textsf{app}\ ) = ctx(\textsf{cons}\ ) &= \{(1, \emptyset), (2, \{1\})\}. \end{aligned}$$

Here, $(2, \{1\})$ means that the second argument is contextual but requires the first to be a value.

Definition 2 (Context summaries). *Given a signature Σ, a* context summary over Σ *is a function ctx from constants of Σ to $\mathcal{P}(\mathbb{N} \times \mathcal{P}(\mathbb{N}))$.*

Typed languages are logic programs augmented with two context summaries for evaluation and error contexts.

Definition 3 (Typed Languages). *A* typed language *is a tuple $(\Sigma, D, ctx, err\text{–}ctx)$ such that*

- *(Σ, D) is a logic program, such that Σ contains kinds* $\textsf{exp}$ *and* $\textsf{type}$ *and*

$$\begin{aligned} &\vdash : \textsf{exp} \to \textsf{type} \to \textsf{o}\,. \\ &\to : \textsf{exp} \to \textsf{type} \to \textsf{o}\,. \\ &\to^* : \textsf{exp} \to \textsf{type} \to \textsf{o}\,. \\ &\textsf{value} : \textsf{exp} \to \textsf{type} \to \textsf{o}\,. \\ &\textsf{error} : \textsf{exp} \to \textsf{type} \to \textsf{o}\,. \end{aligned}$$

- *ctx and $err\text{–}ctx$ are context summaries over Σ.*
- *D contains the rules that define $\to^*$ as the reflexive and transitive closure of $\to$.*

We let $\mathcal{L}$ range over typed languages. Sometimes we use E for variables of kind $\textsf{exp}$, T for those of kind $\textsf{type}$. Terms of kind $\textsf{exp}$ are ranged over by e and those of kind $\textsf{type}$ by t. We use the notation $D_{|pred}$ to denote the subset of rules in D that define the predicate *pred*. For example, $D_{|\vdash}$ and $D_{|\to}$ are the typing and the reduction rules in D, respectively. Given a signature Σ, we denote by $\Sigma(\textsf{exp}\ )$ and $\Sigma(\textsf{type}\ )$ the sets of constant in Σ that define expressions and types, respectively.

The semantics of a typed language $\mathcal{L}$ is denoted by $[\![\mathcal{L}]\!]$ and it is the straightforward counterpart of $\mathcal{L}$ as logic program in which the information in the context summaries is translated into rules. For example, $ctx(\textsf{cons}\ ) = \{(1, \emptyset), (2, \{1\})\}$ generates the two rules below.

$$\frac{E_1 \to E_1'}{(\textsf{cons}\ E_1\ E_2) \to (\textsf{cons}\ E_1'\ E_2)} \qquad \frac{E \to E'}{(\textsf{cons}\ V\ E) \to (\textsf{cons}\ V\ E')}$$

and $err\text{–}ctx(\textsf{head}\ ) = \{(1, \emptyset)\}$ generates $\dfrac{\textsf{error}\ E}{(\textsf{head}\ E) \to E}$.

We overload $\models$ to typed languages, with the meaning that typed languages are first translated to logic programs.

Syntactic Sugar for Representing Languages In the next sections we develop meta type systems that inspect logic programming based representations of language specifications. To our experience, readers may encounter some difficulty in relating the raw syntax so far introduced with what they are familiar with. To help our presentation, we employ some syntactic sugar.

Typing rules are augmented with a type environment for replacing generic and hypothetical occurrences. The symbol for the type environment is fixed and is $\mathbf{\Gamma}$. To make some examples, the following rules on the left describe the typing rules (T-TAIL) and (T-ABS) and (T-ABST) as logic programming rules. On the right, we have the syntax we adopt. The type declared for **mu** is (**type** $\rightarrow$ **type**) $\rightarrow$ **type**.

$$\frac{\vdash\ e : \mathtt{List}\ T}{\vdash\ \mathtt{tail}\ e : \mathtt{List}\ T} \equiv \frac{\mathbf{\Gamma} \vdash\ e : \mathtt{List}\ T}{\mathbf{\Gamma} \vdash\ \mathtt{tail}\ e : \mathtt{List}\ T}$$

$$\frac{(\forall x. \vdash x\ T_1 \Rightarrow \vdash (E\ x)\ T_2)}{\vdash (\mathbf{abs}\ T_1\ E)\ (\mathbf{arrow}\ T_1\ T_2)} \equiv \frac{\mathbf{\Gamma}, x : T_1 \vdash\ E\ T_2}{\mathbf{\Gamma} \vdash (\mathbf{abs}\ T_1\ E)\ (\mathbf{arrow}\ T_1\ T_2)}$$

$$\frac{\forall x. \vdash\ E : (T\ x)}{\vdash\ \mathtt{absT}\ E : (\mathtt{forall}\ T)} \equiv \frac{\mathbf{\Gamma}, x. \vdash\ E : T}{\mathbf{\Gamma} \vdash\ \mathtt{absT}\ E : (\mathtt{forall}\ T)}$$

The predicate $\rightarrow$ is used in infix notation. Also, we adopt the convention that variables V are treated as *value variables* and entail that the rule implicitly contains the premise **value** V. The reduction rule (BETA), for example, is represented as follows.

$$\frac{\mathbf{value}\ E}{(\mathbf{app}\ (\mathbf{abs}\ R)\ E) \rightarrow (R\ E)} \equiv (\mathbf{app}\ (\mathbf{abs}\ R)\ V) \rightarrow (R\ V)$$

Value and error definitional rules are rewritten in the following style. Notice, below we have also applied the convention on value variables.

$$\frac{\mathbf{value}\ E_1 \quad \mathbf{value}\ E_2}{\mathbf{value}\ (cons\ E_1\ E_2)} \quad \equiv \quad \mathbf{value}\ ::= (cons\ V_1\ V_2)$$

$$\frac{\mathbf{value}\ E}{\mathbf{error}\ (raise\ E)} \quad \equiv \quad \mathbf{error}\ ::= (raise\ V)$$

7 A Type System for Type Soundness

In this section, we devise a type system that applies the methodology described in Section 3 to typed languages. Figure 4 shows the type system. The main typing judgment is $\vdash \mathcal{L}$, for a given typed language $\mathcal{L}$. The first line of (T-MAIN) checks that the contexts are not circularly defined. This covers the **D3** item of the methodology. The second line splits the rules of the languages in three categories: value and error definitions, typing rules and reduction rules. Each of these categories is type checked using an appropriate typing judgement. Value and error definitions are type checked with $\vdash_{\mathrm{def}}$, which produces *bindings* that are collected and used by the next type system. Those bindings simply classify

$\boxed{\vdash \mathcal{L}}$

$$\frac{\begin{array}{c}\texttt{rng}(ctx)\ \textit{is a directed acyclic graph} \\ (D_{|\textsf{value}} \cup D_{|\textsf{error}}) = \{\phi_1^{\mathrm{d}}, \ldots, \phi_n^{\mathrm{d}}\} \quad D_{|\vdash} = \{\phi_1^{\mathrm{t}}, \ldots, \phi_m^{\mathrm{t}}\} \quad D_{|\to} = \{\phi_1^{\mathrm{r}}, \ldots, \phi_l^{\mathrm{r}}\} \\ ctx \vdash_{\mathrm{def}} \phi_1 : B_1^{\mathrm{d}} \ \ldots \ ctx \vdash_{\mathrm{def}} \phi_n : B_n^{\mathrm{d}} \qquad \Gamma^{\mathrm{d}} = B_1^{\mathrm{d}}, \ldots, B_n^{\mathrm{d}} \\ D_{|\to} \mid \Gamma^{\mathrm{d}} \vdash_{\mathrm{typ}} \phi_1^{\mathrm{t}} : B_1^{\mathrm{t}} \ \ldots \ D_{|\to} \mid \Gamma^{\mathrm{d}} \vdash_{\mathrm{typ}} \phi_m^{\mathrm{t}} : B_m^{\mathrm{t}} \qquad \Gamma^{\mathrm{t}} = B_1^{\mathrm{t}}, \ldots, B_m^{\mathrm{t}} \\ op : \mathrm{errHandler} \in \Gamma^{\mathrm{t}} \qquad ctx = err\text{–}ctx \cup \{op \mapsto (1, \emptyset)\} \\ ctx \mid \Gamma_1 \vdash_{\mathrm{red}} \phi_1^{\mathrm{r}} : B_1^{\mathrm{r}} \ \ldots \ ctx \mid \Gamma_l \vdash_{\mathrm{red}} \phi_l^{\mathrm{r}} : B_l^{\mathrm{r}} \qquad \Gamma^{\mathrm{t}} - (B_1^{\mathrm{r}}, \ldots, B_m^{\mathrm{r}}) = \emptyset \\ D_{|\vdash} \vdash^{\mathcal{L}}_{\mathrm{pre}} \phi_1^{\mathrm{r}} \ \ldots \ D_{|\vdash} \vdash^{\mathcal{L}}_{\mathrm{pre}} \phi_l^{\mathrm{r}}\end{array}}{\vdash (\Sigma, D, ctx, err\text{–}ctx)} \ (\textsc{ts-main})$$

$\boxed{ctx \vdash_{\mathrm{def}} \phi : B^{\mathrm{d}}}$

$$\frac{1, \ldots, n \in ctx(op)}{ctx \vdash_{\mathrm{def}} \textsf{value} ::= (op\ V_1\ \ldots\ V_n\ \widetilde{E}) : op : \mathrm{value}\ \{1, \ldots, n\}} \ (\textsc{value})$$

$$\frac{1, \ldots, n \in ctx(op)}{ctx \vdash_{\mathrm{def}} \textsf{error} ::= (op\ V_1\ \ldots\ V_n\ \widetilde{E}) : op : \mathrm{error}\ \{1, \ldots, n\}} \ (\textsc{error})$$

$\boxed{D \mid \Gamma^{\mathrm{d}} \vdash_{\mathrm{typ}} \phi : B^{\mathrm{t}}}$

$$D^{\mathrm{r}} \mid \Gamma^{\mathrm{d}}, op : \mathrm{value}\ N \vdash \frac{\Gamma_1 \vdash E_1\ t_1 \quad \ldots \quad \Gamma_n \vdash E_n\ t_n}{\Gamma \vdash (op\ E_1\ \ldots\ E_n)\ (c\ \widetilde{T})} : op : \mathrm{value}\ c\ N \quad (\textsc{t-value})$$

$$\frac{\begin{array}{c}D^{\mathrm{r}}.\mathrm{rules}(op_1) = \{\phi_1, \ldots, \phi_m\} \\ \forall i, 1 \le i \le m, \exists op_2,\ \phi_i = (op_1\ (op_2\ \widetilde{E'})\ \widetilde{E''}) \to e \qquad \Gamma^{\mathrm{d}}(op_2) = \mathrm{value}\ N\end{array}}{D^{\mathrm{r}} \mid \Gamma^{\mathrm{d}} \vdash \frac{\Gamma_1 \vdash E_1\ (c\ \widetilde{T}) \quad \ldots \quad \Gamma_n \vdash E_n\ t_n}{\Gamma \vdash (op_1\ E_1\ \ldots\ E_n)\ t} : op_1 : \mathrm{elim}\ c} \ (\textsc{t-elim})$$

$$\frac{\begin{array}{c}D^{\mathrm{r}}.\mathrm{rules}(op_1) = \{\phi_1, \phi_2\} \\ \phi_1 = (op_1\ (op_2\ \widetilde{E_2})\ \widetilde{E_3}) \to e_1 \qquad \Gamma^{\mathrm{d}}(op_2) = \mathrm{error}\ N \\ \phi_2 = (op_1\ V\ \widetilde{E_4}) \to e_2\end{array}}{D^{\mathrm{r}} \mid \Gamma^{\mathrm{d}} \vdash \frac{\Gamma_1 \vdash E_1\ t_1 \quad \ldots \quad \Gamma_n \vdash E_n\ t_n}{\Gamma \vdash (op_1\ E_1\ \ldots\ E_n)\ t} : op_1 : \mathrm{errHandler}} \ (\textsc{t-errHandler})$$

$$\frac{\begin{array}{c}D^{\mathrm{r}}.\mathrm{rules}(op) = \{\phi_1, \ldots, \phi_m\} \\ \forall i, 1 \le i \le m, \quad \phi_i = (op\ \widetilde{V_i}\ \widetilde{E_i'}) \to e_i\end{array}}{D^{\mathrm{r}} \mid \Gamma^{\mathrm{d}} \vdash \frac{\Gamma_1 \vdash E_1\ t_1 \quad \ldots \quad \Gamma_n \vdash E_n\ t_n}{\Gamma \vdash (op\ E_1\ \ldots\ E_n)\ t} : op : \mathrm{derived}} \ (\textsc{t-derived})$$

$$\frac{T \notin \texttt{vars}(t_1) \cup \ldots \cup \texttt{vars}(t_n)}{D^{\mathrm{r}} \mid \Gamma^{\mathrm{d}}, op : \mathrm{error}\ N \vdash \frac{\Gamma_1 \vdash E_1\ t_1 \quad \ldots \quad \Gamma_n \vdash E_n\ t_n}{\Gamma \vdash (op\ E_1\ \ldots\ E_n)\ T} : op : \mathrm{error}\ N} \ (\textsc{t-error})$$

Fig. 4. Type system for ensuring progress

$\boxed{ctx \mid \Gamma^{\mathrm{t}} \vdash_{\mathrm{red}} \phi : B^{\mathrm{r}}}$

$$\frac{\begin{array}{cc}\Gamma^{\mathrm{t}}(op_1) = \text{elim } c & \Gamma^{\mathrm{t}}(op_2) = \text{value } c\ N \\ \widetilde{V} = V_1\ \ldots\ V_n & N = \{1, \ldots n\} \\ \widetilde{V'} = V'_2\ \ldots\ V'_m & \{1, \ldots m\} \subseteq ctx(op_1)\end{array}}{ctx \mid \Gamma^{\mathrm{t}} \vdash (op_1\ (op_2\ \widetilde{V}\ \widetilde{E})\ \widetilde{V'}\ \widetilde{E'}) \to e : op_1 : \text{eliminates } op_2} \quad \text{(R-ELIM)}$$

$$\frac{\begin{array}{cc}\Gamma^{\mathrm{t}}(op_1) = \text{errHandler} & \Gamma^{\mathrm{t}}(op_2) = \text{error } N \\ \widetilde{V} = V_1\ \ldots\ V_n & N = \{1, \ldots n\} \\ \widetilde{V'} = V'_2\ \ldots\ V'_m & \{1, \ldots m\} \subseteq ctx(op_1)\end{array}}{ctx \mid \Gamma^{\mathrm{t}} \vdash (op_1\ (op_2\ \widetilde{V}\ \widetilde{E})\ \widetilde{V'}\ \widetilde{E'}) \to e : op_1 : \text{eliminates } op_2} \quad \text{(R-ERRHANDLER)}$$

$$\frac{\widetilde{V} = V_2\ \ldots\ V_n \qquad \{1, \ldots n\} \subseteq ctx(op)}{ctx \mid \Gamma^{\mathrm{t}}, op : \text{errHandler} \vdash (op\ V_1\ \widetilde{V}\ \widetilde{E}) \to e : \text{plain}} \quad \text{(R-ERRHANDLER-VALUE)}$$

$$\frac{\{1, \ldots n\} \subseteq ctx(op)}{ctx \mid \Gamma^{\mathrm{t}}, op : \text{derived} \vdash (op\ V_1\ \ldots\ V_n\ \widetilde{E}) \to e : \text{plain}} \quad \text{(R-DERIVED)}$$

Fig. 5. Type system for ensuring progress, continued: reduction rules

values and errors. Typing rules are type checked by $\vdash_{\mathrm{typ}}$, which again produces bindings that are collected. This time the bindings fully classify the operators according to the classes of Section 3. After this classification, we can know about the error handler, therefore the 5-th line of (T-MAIN) check that error contexts respect **D2**, i.e. they are the evaluation contexts minus the error handler at the eliminating argument. $\vdash_{\mathrm{red}}$ type checks the reduction rules and produces bindings that keep track of the operators that are actually eliminated by other operator with a reduction rule. We use these bindings against the classification (in Γ^{t}) with $\Gamma^{\mathrm{t}} - (B^{\mathrm{r}}_1, \ldots, B^{\mathrm{r}}_m) = \emptyset$, which is a slight abuse of notation to denote the check whether *all* values are eliminated by *all* eliminators and also that the error is. The difference operator that we denote with $-$ is straightforward and we omit its definition here. Additionally the last line employs the type system $\vdash^{\mathcal{L}}_{\mathrm{pre}}$ for checking whether all the reduction rules are type preserving.

Below, we explain our type systems in detail. The grammar that we employ in our type systems is the following. Below, Γs are type environment as usual, and Bs stand for bindings.

$$\begin{aligned}\mathcal{X} \in \{\mathrm{d}, \mathrm{t}, \mathrm{r}\}, c \in \Sigma(\textsf{type}), op \in \Sigma(\textsf{exp}), N \subseteq \mathbb{N} \\ \Gamma^{\mathcal{X}} &::= \emptyset \mid B^{\mathcal{X}} \\ B^{\mathcal{X}} &::= op : role^{\mathcal{X}} \\ role^{\mathrm{d}} &::= \text{value } N \mid \text{error } N \\ role^{\mathrm{t}} &::= \text{value } c\ N \mid \text{error } N \mid \text{elim } c \mid \text{derived} \mid \text{errorHandler} \\ role^{\mathrm{r}} &::= \text{plain} \mid \text{eliminates } op\end{aligned}$$

A Type System for Definitions The type system for definitions has a judgement of the form $ctx \vdash_{\mathrm{def}} \phi : B^{\mathrm{d}}$. The context ctx is necessary for checking that progress-dependent arguments of values and the error are contextual.

(VALUE) processes a value definition and classifies the operator as value. Notice that at this point, we do not know which type the operator builds a value of. This information is stored in typing rules and will be added later.

(ERROR) has the same role as (VALUE) for the error definition. Notice that for both, N keeps track of the argument positions that need to be values for the definition to apply. This information is needed when type checking the reduction rules of eliminators, as explained later.

A Type System for Typing Rules The type system for typing rules has a judgement of the form $D \mid \Gamma^{\mathrm{d}} \vdash_{\mathrm{typ}} \phi : B^{\mathrm{t}}$. The argument D is the set of reduction rules of the language. This argument is needed for distinguishing the role of some operators. For example, we can distinguish an eliminator from an error handler by retrieving its reduction rules and checking whether they eliminate values or the error.

(T-VALUE) applies only for typing rule of operators that Γ^{d} classifies as values. The shape of the typing rule deserves some attention. This shape imposes that the assigned type has the form $(c\ \widetilde{T})$, that is, a constructed type. It also imposes that all the arguments of the operator are the subject of a typing premise, as prescribed by **P-Typ**. Throughout this type system, we fix the convention that $\mathbf{\Gamma}_1, \ldots \mathbf{\Gamma}_n$ are build with $\mathbf{\Gamma}$ and they exclusively can be of the form $\mathbf{\Gamma}_i ::= \mathbf{\Gamma} \mid \mathbf{\Gamma}, x \mid \mathbf{\Gamma}, x : T$. This means that (T-VALUE) allows for ordinary typing premises as well as generic and hypothetical premises.

(T-ELIM) classifies eliminators at the encounter of their typing rule. The shape of the typing rule imposes that the type of the eliminating argument has the form $(c\ \widetilde{T})$, that is, a constructed type. Next, we retrieve the reduction rules of the operator and check that these rules all eliminate some value. This is done by matching each reduction rule with the form $(op_1\ (op_2\ \widetilde{E'})\ \widetilde{E''}) \to e$. We check then that op_2 is a value for each of these rules.

(T-ERRHANDLER) classifies the error handler. We retrieve the reduction rules for this operator. There must be exactly two such rules: one that eliminates the error, and the other that fires for values at the eliminating type, as prescribed by **D5**.

(T-DERIVED) classifies derived operators. We check that all the reduction rules for the operator are of the form $(op\ \widetilde{V_i}\ \widetilde{E'_i}) \to e_i$. This means that the arguments of op are variables, whether value or expression variables, and there is no pattern-matching of constructed expressions.

(T-ERROR) handles the typing rule for the error. We enforce that the assigned type is a free variable T. This ensures that the error can be typed at any type, as prescribed by **D-Err**.

A Type System for Reduction Rules The type system for type checking reduction rules has a judgement of the form $ctx \mid \Gamma^{\mathrm{t}} \vdash_{\mathrm{red}} \phi : B^{\mathrm{r}}$. The binding produced by this judgement records whenever an operators eliminates another one with a binding of the form (op_1 : eliminates op_2). This happens for reduction rules of eliminators and for the reduction rule that handles the error. All other reduction

rules produce a binding with the label (plain), meaning that no elimination takes place when firing the rule. We show this type system in Figure 5.

(R-ELIM) type checks a reduction rule for an eliminator. The shape of this rule must be of the form $(op_1\ (op_2\ \widetilde{V}\ \widetilde{E})\ \ \widetilde{V'}\ \widetilde{E'}) \to e$. In particular, notice the complex expression at the eliminating argument. We check that op_1 is an eliminator for some type constructor c and that op_2 is indeed a value for that type. We impose that the rule fires exactly when op_2 forms a value. To this aim, $(op_2\ \widetilde{V}\ \widetilde{E})$ must be such that $\widetilde{V}$ contains precisely those positions prescribed by N. With the check $\{1, \dots m\} \subseteq ctx(op_1)$ we impose that the eliminating argument (index 1) is contextual, and that also its sibling arguments that are tested for valuehood are. This is prescribed by **D1**.

(R-ERRHANDLER) type checks the reduction rule that handles the error. The way we handle this case is very similar to that for (R-ELIM). It differs from (R-ELIM) in that it makes sure that op_1 is the error handler and that op_2 is the error.

(R-ERRHANDLER-VALUE) type checks the reduction rule that defines the step of the error handler for values. The form of the rule must be $(op\ V_1\ \widetilde{V}\ \widetilde{E}) \to e$, which imposes the eliminating argument to be a value variable. We then check that the evaluation contexts are properly defined.

(R-DERIVED) type checks the reduction rules for derived operators. The shape of these rules imposes that no pattern-matching would take place. As for the previous cases, we then check that evaluation contexts are properly defined.

A Type System for Type Preservation We now explain the type system that ensures that reduction rules are type preserving. The judgement for this task takes the form $D \vdash^{\mathcal{L}}_{\text{pre}} \phi$. The argument D is the set of typing rules of the language. Typing rules are necessary because we build symbolic type environments out of them. Figure 6 shows the type system for $\vdash^{\mathcal{L}}_{\text{pre}}$. For this type system, we fix the notation that $e[\widetilde{E}]$ is an expression that can use only variables in $\widetilde{E}$. Similarly, in $e[\widetilde{E'}, \widetilde{E''}]$ we have that e uses only variables in $\widetilde{E'}$ and $\widetilde{E''}$. Since typing rules are unique for an operator we use the notation $D^{\text{t}}(op)_{.\text{output}}$ for retrieving the assigned type of the typing rule for op. For example, $D^{\text{t}}(\textsf{cons})_{.\text{output}} = \textsf{List}\ \ T$. Similarly, the notation $D^{\text{t}}(op)_{.\text{premises}}$ retrieves the set of premises of the typing rule for op.

Rule (PRE-MAIN) treats a rule of the form $(op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''}) \to e_2[\widetilde{E'}, \widetilde{E''}]$. Recall that, virtually, we need to establish that $(op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''})$ and $e_2[\widetilde{E'}, \widetilde{E''}]$ have the same type. To do this, we compute the symbolic type environment with the call $D^{\text{t}} \vdash_{\text{symb}} (op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''}) : \Gamma^s$. The type judgement $\vdash_{\text{symb}}$ takes an expression and returns a symbolic type environment, that is simply a set of typing formulae. Rule (SYMB-ONE) handles the case where $(e_1[\widetilde{E'}])$ is not a complex expression. This happens for reduction rules for derived operators, for example. In this case, we build the symbolic type environment with the premises of the typing rule for op_1. Reduction rules for eliminators and for handling the error are such that $(e_1[\widetilde{E'}])$ is built with a top level operator op_2. This case is handled by (SYMB-TWO), which builds the symbolic type environment with the

typing premises of both op_1 and op_2. Once we have computed the symbolic type environment Γ^s, we check that the source and the target of the reduction rule are typed at the same type when Γ^s is used. This type is the type assigned by the typing rule of op_1. We check this with $\vdash^{\mathcal{L}}_{\text{ent}}$ which builds the appropriate query that we check for entailment. The function $(\cdot)^{\forall}$ simply quantifies universally over all the variables of the query. Notice that the query is checked in the language augmented with the axiom for (EQ-SUB), which translates in our setting as

$$\begin{aligned}(\text{EQ-SUB})^* = \forall E_1, E_2, T_1, T_2,\\ (\forall x, \vdash\ x\ T_1 \Rightarrow \vdash\ (E_1\ x)\ T_2) \wedge \vdash\ E_2\ T_1 \Rightarrow \vdash\ (E_1\ E_2)\ T_2.\end{aligned}$$

As E_1 is an abstraction, $(E_1\ E_2)$ encodes the substitution $E_1[E_2/x]$ in HOAS.

$\boxed{D \vdash^{\mathcal{L}}_{\text{pre}} \phi}$

$$\frac{\begin{array}{c} D^{\text{t}} \vdash_{\text{symb}} (op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''}) : \Gamma^s \\ \Gamma^s \vdash^{\mathcal{L}}_{\text{ent}} (op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''}) : D^{\text{t}}(op_1).\text{output} \\ \Gamma^s \vdash^{\mathcal{L}}_{\text{ent}} e_2[\widetilde{E'}, \widetilde{E''}] : D^{\text{t}}(op_1).\text{output}\end{array}}{D^{\text{t}} \vdash^{\mathcal{L}}_{\text{pre}} (op_1\ (e_1[\widetilde{E'}])\ \widetilde{E''}) \rightarrow e_2[\widetilde{E'}, \widetilde{E''}]}\ (\text{PRE-MAIN})$$

$$\Gamma^s \vdash^{\mathcal{L}}_{\text{ent}} e : t \equiv (\mathcal{L} \cup (\text{EQ-SUB})^*) \models (\Gamma^s \Rightarrow \vdash\ e\ t)^{\forall}$$

$\boxed{D \vdash_{\text{symb}} e : \Gamma^s}$

$$D^{\text{t}} \vdash_{\text{symb}} (op_1\ \widetilde{E}) : \bigwedge D^{\text{t}}(op_1).\text{premises}\quad (\text{SYMB-ONE})$$

$$\frac{\begin{array}{c}\Gamma^s_1 = \bigwedge D^{\text{t}}(op_1).\text{premises} \\ \Gamma^s_2 = \bigwedge D^{\text{t}}(op_2).\text{premises}\end{array}}{D^{\text{t}} \vdash_{\text{symb}} (op_1\ \ (op_2\ \widetilde{E_1})\ \widetilde{E_2}) : \Gamma^s_1 \wedge \Gamma^s_2}\ (\text{SYMB-TWO})$$

Fig. 6. Type System for ensuring Type Preservation

8 Well-typed Languages are Sound

We are now ready to establish our main results. We rely on the type system of logic programs in the sense of Church (see [9]). This type system is denoted with $\vdash_{\text{lp}}$ and rejects ill-typed logic programs with mistakes such as $\vdash\ T\ T$ and **app** *arrow arrow*. Thanks to this, our type system $\vdash \mathcal{L}$ does not check for those errors and could focus on its higher level task. Below, we use $\vdash_{\text{lp}}$ lifted to typed languages.

$$\vdash_{\text{ts}} \mathcal{L} \equiv \vdash_{\text{lp}} \mathcal{L} \text{ and } \vdash \mathcal{L}.$$

Theorem 1 (Well-typed languages afford progress). *For all typed languages $\mathcal{L}$ and for all e and T, if $\vdash_{ts} \mathcal{L}$ and $\mathcal{L} \models \vdash\ e\ T$ then either $\mathcal{L} \models$ **value** e, $\mathcal{L} \models$ **error** e, or there exists e' such that $\mathcal{L} \models e \rightarrow e'$.*

Theorem 2 (Well-typed languages are type preserving). *For all typed languages $\mathcal{L}$ and for all e, e' and T, if $\vdash_{ts} \mathcal{L}$, $\mathcal{L} \models \vdash\ e\ T$ and $\mathcal{L} \models e \rightarrow e'$ then $\mathcal{L} \models \vdash\ e'\ T$.*

Type soundness follows from the progress and preservation theorem in the usual way.

Theorem 3 (Well-typed languages are sound). *For all typed languages $\mathcal{L}$ and for all e, e' and T, if $\vdash_{ts} \mathcal{L}$, $\mathcal{L} \models \vdash\ e\ T$ and $\mathcal{L} \models e \rightarrow^* e'$ then either*

- $\mathcal{L} \models$ **value** e',
- $\mathcal{L} \models$ **error** e', *or*
- *there exists e'' such that $\mathcal{L} \models e' \rightarrow e''$.*

The proofs of the theorems above can be found in the appendix.

9 Implementation: the *TypeSoundnessCertifier*

Based on the work of this paper, we have implemented a tool that we have called *TypeSoundnessCertifier*. The tool is written in Ocaml and reads Abella specifications (basically λProlog specifications) augmented with special tags for declaratively specifying evaluation contexts. The tool implements a type-checker based on the type system of Sections 7. We have realized the type system for type preservation by automatically generating queries to the Abella theorem prover.

We have applied our tool to several variants of the simply typed lambda calculus with various subsets of the following features: pairs, `if`–`then`–`else`, lists, sums, unit, tuples, `fix`, `let`, `letrec`, universal types, recursive types and exceptions. We have also considered different strategies such as call-by-value, call-by-name and a parallel reduction strategy as well as lazy pairs, lazy lists and lazy tuples. We have type checked a total of 103 type sound languages, including a rich language such as `Fpl`.

Remarkably, *TypeSoundnessCertifier* spots design mistakes that hinder type soundness. Among other kind of errors, the tool reports an error whenever

- Some eliminator does not eliminate all the values it is supposed to eliminate.
- Some relevant evaluation context is not declared.
- Context declarations have circular dependencies such as `cons` $E\ v$ | `cons` $v\ E$, mentioned in Section 4.3.
- Some reduction rules are not type preserving. For example, If we mistake the operational semantics of `fst` and define it to return the second component of a pair, the type-checker points out the bad rule.

In general, thanks to our type system setting the tool can algorithmically detect departures from the methodology of Section 4 and report them to the user.

Certified languages: For those language specifications that we have type checked, *TypeSoundnessCertifier* can automatically produce a formal proof of type soundness and related theorems that can be checked by the Abella theorem prover [2]. In this paper, we simply report on this aspect of the tool: In addition

to our proofs in the appendix, these machine-checked proofs give us strong confidence that our type system does guarantee the type soundness of languages. A serious investigation on the automatic certification algorithms of the tool is part of our future work.

10 Related Work

The meta-theory set forth in this paper is inspired by a line of research on the meta-theory of operational semantics, and especially on results on *rule formats* [12]. These results typically offer templates and restrictions to operational semantics specifications that can guarantee that some property holds. Typical work from this line of research have been used for establishing various results for process algebras and mostly in the context of equations modulo bisimilarity and congruence [3, 5, 11, 1]. This paper shares the same spirit with those results, though it targets programming languages with types, ensures type soundness, and aims at offering a typing discipline rather than syntactic restrictions.

A close work to ours is that of Schwaab and Siek [16], and that of Delaware et al [6]. In both, the authors offer a solution to the expression problem [18]. These results offer two (very different) solutions to the safe composition of already existing proofs. In this regard, their results are orthogonal to ours and we plan to accommodate their insights in our context.

The specific use of logic programs for encoding operational semantics and typing rules dates back to Kahn's *natural semantics* [8] and its machine implementation [4]. The particular use of higher-order logic programming as a specification language dates back to Schürmann and Pfenning [15].

Finally, there are several tools that are tailored to particular specification of languages, such as Ott [17], Lem [13], the K framework [14], and PLT Redex [7], among others. In many ways, *TypeSoundnessCertifier* shares the same spirit in assisting language designers with their designs. To our knowledge, the way our tool ensures type soundness and the way it informs language designers of design mistakes is a novelty in tools for language design since *TypeSoundnessCertifier* presents features that are orthogonal to those of the mentioned tools. Of course, these other mature tools offer remarkable help to language designers in multiple aspects, including features for executing, evaluating, testing and exporting language specifications.

11 Conclusions and Future Work

In this paper, we somehow treated language specifications as expressions and we have demonstrated that the appropriate typing discipline over these specifications guarantees that the language is type sound: that is, *well-typed languages are sound*.

We have demonstrated this idea with a class of languages based on constructors, eliminators and errors: features that are common in programming language

design. This class is fairly expressive and comprises languages with modern features such as recursive types, polymorphism and exceptions.

Are there programming languages that are out of the reach of our methodology? Yes, definitely many. This is our first paper on the topic and we have only scratched the surface of this research area. Perhaps, the two most natural extensions to the present work are to languages with stores/references and languages with subtyping. These extensions are not as trivial as they might seem. For example, languages with stores/reference carry a heap, and a notion of safety must be systematically derived for the heap. These languages also impose adjustments to the preservation theorem statement for accommodating a location environment, that might grow over time.

Similarly, languages with subtyping bring their own difficulties. For example, as both the language and the subtyping relation are provided by the language designer, we would need principled ways to enforce that object subtyping is rejected when it is covariant in calculi with updates (unsound), and when references are covariant (unsound), and all similar scenarios. We leave an investigation of these classes of languages as future work.

Other classes of languages, such as linear types, dependent types, type-effect systems and typestate, to name a few, are out of the scope of our applicability and they seem to come with their own specific research challenges. We leave these extensions to future work. Similarly, we plan to investigate whether we can translate our results to the style of big step operational semantics.

In this paper, we conjecture that *well-typed languages are sound* is a perspective that, just like *well-typed programs cannot go wrong*, applies across several classes of languages. We will be eager to work with the community to explore this research area further.

References

[1] Luca Aceto, Anna Ingolfsdottir, Mohammadreza Mousavi, and Michel A. Reniers. A rule format for unit elements. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 141–152, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[3] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, January 1995.

[4] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*, pages 14–24, Boston, 1988.

[5] Sjoerd Cranen, MohammadReza Mousavi, and Michel A. Reniers. A rule format for associativity. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 2008.

[6] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM.

[7] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
[8] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, March 1987.
[9] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.
[10] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.
[11] M. R. Mousavi, M. A. Reniers, and J. F. Groote. A Syntactic Commutativity Format For SOS. *Information Processing Letters*, 93(5):217 – 223, 2005.
[12] MohammadReza Mousavi, Michel A. Reniers, and Jan Friso Groote. SOS formats and meta-theory: 20 years after. *Theor. Comput. Sci.*, 373(3):238–272, March 2007.
[13] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, 2014. ACM.
[14] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
[15] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *15th Conf. on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1998.
[16] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 3–12, New York, NY, USA, 2013. ACM.
[17] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM.
[18] Philip Wadler. The Expression Problem. Discussion on the Java Genericity mailing list., November 1998. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.
[19] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Progress Theorem

The Main Progress Theorem Assume $\vdash_{\text{ts}} \mathcal{L}$ and $\mathcal{L} \models \vdash\ e\ T$. The proof is by induction on $\mathcal{L} \models \vdash\ e\ T$. Being $\mathcal{L} \models \vdash\ e\ T$ provable, it means that there exist a typing rule ϕ that of them form $\dfrac{p_1, \ldots p_n}{\vdash\ (op\ \widetilde{E}) : t}$ that is 'is satisfied'. The rule 'is satisfied' in the sense that there exists a substitution γ from logical variables (of the rule) to logical terms such that $\mathcal{L} \models p_i\gamma$ for $\{1, \ldots, n\}$ and $\vdash\ (op\ \widetilde{E})\gamma : t\gamma = \vdash\ e\ T$.

Since $\phi \in \mathcal{L}$ and ϕ is a typing rule, then it has been type checked with $\vdash_{\text{typ}}$. This means that *all* variables in $\widetilde{E}$ are the subject to a typing premise (**P-Typ** common pattern), i.e. $\mathcal{L} \models \vdash\ E_i\gamma\ T\gamma$ for $E_i \in \widetilde{E}$. This means that we can apply

the inductive hypothesis to each E_i if we wish. Of course, it matters to apply the inductive hypothesis to progress-dependent arguments only, if we were to be optimal. The paper does not set a notation for extracting progress-depending arguments, we simply apply the inductive hypothesis to contextual arguments of op. This is suboptimal (only slightly) but correct.

Notice also that in HOAS some variables might be abstractions and might be subject to typing premises p_i that might be hypothetical or generic. However, the shape of the value premises, value definitions and error definitions is simple, for example **value** V: this implicitly forbids evaluation under a binder because for that we need another generic premise, but we use simple premises. (evaluation under binders is not common in programming languages). In short, the contextual variables are of simple expression variables and we can apply IH as usual. We retrieve the contextual in the following way.

By definition of $\mathcal{L}$, $\mathcal{L}$ has the function *ctx* for contexts. Given the operator *op* above, and given $\{i_1, \ldots, i_n\} \in \texttt{fst}(ctx(\textbf{op}))$, we apply the inductive hypothesis to $\mathcal{L} \models \vdash E_{i_j}\gamma\, T\gamma$, for all $1 \leq j \leq n$. Now we have that those $E_{i_j}\gamma$ progress. We call the Progress Lemma for *op* (defined below) passing the assumptions that $E_{i_j}\gamma$ progress. Notice that such lemma expects exactly those progress assumptions and in that number (the number of contextual arguments), as explained below.

Progress Lemma for all op Given an operator *op* of kind $(\ldots \to \textbf{term})$ in $\mathcal{L}$, we prove the following theorem.

Theorem 4. *if* $\vdash\vdash$*, it holds that for all* $op \in \Sigma(exps)$*, for all* $\{i_1, \ldots, i_n\} \in \texttt{fst}(ctx(\textbf{op}))$*, if progress* e_1*, ... progress* e_n*, then progress* $(op\ e_1 \ldots\ e_n\ \widetilde{e})$ *for all* e_1*, ...,* e_n *and* $\widetilde{e}$ *(here* $\widetilde{e}$ *are the rest of the arguments, respecting the arity of* **op** *, that are not contextual).*

The proof is by case analysis on all progress e_1, ... progress e_n, but in a suitable order. Since $\vdash\vdash$ we have that $ctx(op)$ is acyclic, therefore we can choose an order such that (**invariant:**) we do case analysis on progress e_i before the case analysis on progress e_j if the context for i-th argument of *op* does not depend on the valuehood of the argument j of *op*.

After the series of cases analysis on progress e_i, we are at the leftmost child of the leftmost tree of the cases.

Before continuing: an example. If we have two arguments, after the first case analysis on *progress* e_1 we open three cases: *value* e_1 and *progress* e_2, *step* e_1 and *progress* e_2, *error* e_1 and *progress* e_2. And we are at the left child. We now do case analysis on *progress* e_2 and we open other three cases only on the left child: the leftmost subtree is *value* e_1 and *value* e_2, *value* e_1 and *step* e_2, *value* e_1 and *error* e_2. And we are at the leftmost child: *value* e_1 and *value* e_2.

Now we continue the proof. After the series of cases analysis on progress e_i, we are at the leftmost child of the leftmost tree of the cases. In this cases, *all arguments are values.*

The proof is by case analysis on how *op* has been classified with $\vdash_{\text{typ}}$.

– op : value c N: We dismiss the leftmost child in the following way: Since the typing rule ϕ has been typed, then Γ_{def} contains op : value N. Which means that there exists a value definition ϕ^d for op. Since ϕ^d has been typed with Γ_{def}, it means that the shape of the value deifinition is such that it is restricted only by value premises, i.e. by the valuehood of its arguments (this realizes the common pattern **P-Val**). As we are in the case where all the arguments are values, the definition applies and this case progresses.
Now, we are left with two cases *step* e_1 and *error* e_1. However, notice that once we dismiss them we have dismissed the whole case *value* of the subtree immediately above. Therefore, we go straight to prove the cases *step* and *error* of the subtree immediately above. Thanks to the invariants on the dependency on the valuehood, we can do a uniform proof for all the level of the tree. We have
  - **STEP:** *step* e_j i.e. for some j. As $j \in ctx(\textsf{op})$ by the semantics of $\mathcal{L}$ (translation to logic programs), this means that there exists a rule $\dfrac{\textsf{value}\ E_1 \dots\ E_j \to E'_j}{(\textsf{op}\ \dots E_j \dots) \to (\textsf{op}\ \dots E'_j \dots)}$. Notice that we have ordered the arguments by dependency on valuehood, therefore value premises can be applied, if any, only to $E_1 \dots$ previous to E_j. However, we deal with the case $\to e_j$ only when the previous arguments are values. So we can instantiate and prove a step
$$\mathcal{L} \models (\textsf{op}\ \dots e_j \dots) \to (\textsf{op}\ \dots e'_j \dots)$$
So $(op\ \widetilde{E})$ progresses.
  - **ERR:** *error* e_j for some j, i.e. e_j is an error. Since $j \in ctx(\textsf{op})$ and since op is not an error-handler then $j \in \textsf{err–ctx}(\textsf{op})$. By by the semantics of $\mathcal{L}$ (translation to logic programs) this means that there exists a rule $\dfrac{\textsf{value}\ E_1 \dots \textsf{error}\ E_j}{(\textsf{op}\ \dots E_j \dots) \to E_i}$. Again, as we have ordered the argument by dependency on valuehood, the arguments $e_1 \dots$ are values and the rule can be applied to prove the step $\mathcal{L} \models (op \dots\ e_j\ \dots) \to e_j$. So $(op\ \widetilde{E})$ progresses.

– op : elim c: As ϕ is a typing rule of $\mathcal{L}$ and $\vdash \mathcal{L}$, we have that ϕ has been type checked by $\vdash_{\text{typ}}$. This means that ϕ is of the following shape.
$$r = \frac{\vdash E_1\ (c\ \widetilde{T})}{\vdash (\textsf{op}\ \widetilde{E})}$$
Since rule r has been satisfied, so are its premises. Then, we have $\mathcal{L} \models \vdash e_1\ (c\ \widetilde{T})$. Since we are in the leftmost case, where all arguments are values, we have $\mathcal{L} \models \textsf{value}\ e_1$. Therefore, we apply the Canonical Forms Lemma for c (described in the following paragraph). This means that $e_1 = \{(t_1, V_1) \vee \dots \vee (t_m, V_m)\}$ (this is notation from the next paragraph). Let us fix one such pair (t_k, V_k). By $\vdash \mathcal{L}$, we have $t_k = (op_2\ \widetilde{E'})$. This means that $\vdash_{\text{typ}}$ has type checked op_2 as op_2 : value c V_k. Since $\vdash \mathcal{L}$ succeded also $\vdash_{\text{red}}$ succeded,

which means that the exhaustiveness check $\Gamma^{\mathrm{t}} - (B^{\mathrm{r}}_1, \ldots, B^{\mathrm{r}}_m) = \emptyset$ succeded, and means that since $op_2 : \text{value } c\ V_k \in \Gamma_{\mathrm{typ}}$ then we had a reduction rule r_{step} such that has been type checked by $\vdash_{\mathrm{red}}$ as $op :$ eliminates op_2, because $op :$ elim c.
Since r_{step} has been type checked by $\vdash_{\mathrm{red}}$ with (R-ELIM) it is of them form: $r_{\mathrm{step}} = \dfrac{ps}{\to (op\ (op_2\ \widetilde{E'})\ \widetilde{E''}\ \ t)}$. Therefore we could apply this rule, only provided that premises ps are satisfied. The shape of the rule also imposes that ps are only value premises. These premises are of two kind:

- **value** E_u where $E_u \in \widetilde{E''}$. Then we are in the case where all of those Es are values. Indeed, those arguments are progress-depending arguments for needing valuehood. Also, we are in the leftmost case of the case analysis on all progresses e_j on progress-depending arguments. Thus, we have $\mathcal{L} \models$ **value** e_u for each of them, which satisfies the premise.
- **value** E_u where $E_u \in \widetilde{E'}$. Then $\vdash_{\mathrm{prg}} \mathcal{L}$ imposes that the index $u \in V_k$, which means $\mathcal{L} \models$ **value** e_u (as defined in the next paragraph), therefore also this premise is satisfied.

We can therefore apply the rule r_{step} above and prove $\mathcal{L} \models (op \ldots\ e_j\ \ldots) \to e'$ for some e'. So this cases progresses.
Cases **STEP** and **ERR** are proved as in the previous case for constructors.

The other operators are easier to handle.

- $op :$ error N: The leftmost leaf of errors is handled similarly as to values and so are **STEP** and **ERR**.
- $op :$ derived: Then ϕ is typed by $\vdash_{\mathrm{typ}}$ by (T-DERIVED). Therefore, it exists a rule $(op\ \widetilde{V_i}\ \widetilde{E'_i}) \to e_i$. As those V_i are tested for valuehood they are progress-dependent arguments, so we are in the case analysis of their progress and in particular, they are all values because we are in the leftmost case. Therefore that reduction rule applies and this case progresses. For derived operators, **STEP** and **ERR** also follow the same line as in the previous cases.
- $op_1 :$ errHandler: Then ϕ is typed by $\vdash_{\mathrm{typ}}$ by (T-ERRHANDLER). Therefore, it exists a rule $\phi_2 = (op_1\ V\ \widetilde{E_4}) \to e_2$. As we are in the leftmost case, the eliminating argument, i.e. the first argument, is a value and thus we can apply the reduction rule. So this case progresses. For error handlers, **STEP** follows the line as in the other case, while **ERR** is different: since ϕ is typed by $\vdash_{\mathrm{typ}}$ by (T-ERRHANDLER), it means it exists a rule $\phi_1 = (op_1\ (op_2\ \widetilde{E_2})\ \widetilde{E_3}) \to e_1$ and $\Gamma^{\mathrm{d}}(op_2) = \text{error } N$ .Also, ϕ_1 has been typechecked by $\vdash_{\mathrm{red}}$ which ensures that it fires when the first argument is an error. Therefore we can apply this rule. So this case progresses.

Canonical Forms Lemma for c

Theorem 5. *For all e, c, if $\vdash \mathcal{L}$ and $\mathcal{L} \models \vdash e\ (c\ \widetilde{T})$ and $\mathcal{L} \models$* **value** *e then $e = \{(t_1, V_1) \vee \ldots \vee (t_m, V_m)\}$ where for all $1 \leq j \leq m$, $t_j = op\ \widetilde{e}$ and*

- *(Part 1) $op :$ value $c\ V_j \in \Gamma_{typ}$.*

– *(Part 2)* $\mathcal{L} \models \textsf{value}\ e_i$ *when* $i \in V_j$.

Proof. Assume the hypothesis. As $\mathcal{L} \models \vdash\ e\ (c\ \widetilde{T})$ and $\vdash \mathcal{L}$, then it means that e is typed with a typing rule r whose input is $(\textsf{op}\ \widetilde{E})$, that is, $e = (\textsf{op}\ \widetilde{e})$.

Part 1: Since $\mathcal{L} \models \textsf{value}\ e$, op : value $V_j \in \Gamma_{\text{def}}$, therefore T-ELIM finds op : value $V_j \in \Gamma_{\text{def}}$ and the typing rule r and classifies op : value c , which means op : value $V_j \in \Gamma_{\text{typ}}$.

Part 2: Since we have $\mathcal{L} \models \textsf{value}\ e$ (recall $e = t_j = op\ \widetilde{e}$), we have a rule of form $\dfrac{\textsf{value}\ E_1 \ldots \textsf{value}\ E_n}{\textsf{value}\ (op\ E_1\ \ldots\ E_n\ \ldots)}$. Therefore, all e_i, ..., $e_n \in \widetilde{e}$ are such that $\mathcal{L} \models \textsf{value}\ e_i$ for $1 \leq j \leq n$. Now, by (VALUE), all indexes in V_j are exactly those indexes of the arguments tested for valuehood in that rule.

B Type Preservation

The proof is by induction on $\mathcal{L} \models e \to e'$.

As the formula $\mathcal{L} \models e \to e'$ is provable, it means that there exists a rule of $\mathcal{L}$ that is satisfied and proves the conclusion $e \to e'$. This rule can have three different shapes:

– **contextual rule:** $e = (op\ \widetilde{e})$ and the rule is of the form

$$\frac{E_i \to E_i'}{(\textsf{op}\ \ldots E_i \ldots) \to (\textsf{op}\ \ldots E_i' \ldots)}$$

By the assumptions of the preservation theorem, we have $\mathcal{L} \models \vdash\ (op\ \widetilde{e})\ T$, and so we have typing rule ϕ in $\mathbb{T}$ that proves this typeability fact. Also, since $\vdash \mathcal{L}$, we have that ϕ is typed by $\vdash_{\text{typ}}$. This means that the shape of the rule is such that all arguments $\widetilde{e}$ are typed, including e_i, that is $\mathcal{L} \models \vdash\ e_i\ T'$. As in the proof for progress, since E_i is a contextual argument it cannot be an abstraction but is a simple expression variable. Then we can apply the inductive hypothesis on it and obtain that $\mathcal{L} \models \vdash\ e_i' : T'$. It is easy to see that if $\mathcal{L} \models \vdash\ (op\ \widetilde{e})\ T$ then $\mathcal{L} \models \vdash\ (op\ \widetilde{e}[e_i'/e_i])\ T$. That is, the reduction is type preserving.

– **error steps:** $e = (op\ \widetilde{e})$, $i \in \textsf{err–ctx}\ (\textsf{op}\ )$ and the step has been proved with a rule of the form $\dfrac{\textsf{error}\ E_i}{(\textsf{op}\ \widetilde{E}) \to E_i}$ The fact that $\vdash \mathcal{L}$ imposes that there exists a typing rule that types the error. This is because op : errHandler $\in \Gamma^{\text{t}}$ and $ctx = err\text{–}ctx \cup \{op \mapsto (1, \emptyset)\}$. And successively, we have that $\vdash_{\text{red}}$ imposes that the a reduction rule for op : errHandler is well-typed and that consumes the error in Γ_{typ}, which exists only when a typing rule for the error has been type checked. Now, as this rule has been type checked by Γ_{typ} and by (T-ERROR), we have that the shape of the rule is such that the assigned type is a fresh new variable. So we can prove $\mathcal{L} \models \vdash\ e_i : T$. That is, The reduction is type preserving.

– **by reduction rules:** We see solely the case for a step of an eliminator. This proof case subsumes that of other reducers (derived operators and error handlers). Assume $e = (op\ (op_2\ \widetilde{e'})\ \widetilde{e''})$ of type T and the following reduction rule by which the step has been proved.

$$\frac{ps}{(op\ (op_2\ \widetilde{E'}) \to \widetilde{E''})\ \ t)}$$

(As $\vdash \mathcal{L}$, the rule above has been type cheked by $\vdash_{\text{red}}$, so ps contains only value premises.) As $\vdash \mathcal{L}$ and by the assumptions of the preservation we have that $\mathcal{L} \models \vdash\ (op\ (op_2\ \widetilde{e'})\ \widetilde{e''})\ T$, then this latter fact is proved with a rule for which there exists a substitution γ that satisfies the rule and such that $(op\ (op_2\ \widetilde{E'})\ \widetilde{E''})\gamma = e$. We have to prove that $t\gamma$ is of type T. Since this is a typing rule of $\mathcal{L}$, we have that it has been typechecked with $\vdash_{\text{typ}}$, which means all arguments $\widetilde{e''}$ are well-typed, and also $(op_2\ \widetilde{e'})$ is well-typed. Now, these expressions are well-typed with a corresponding typing formula $\mathcal{L} \models \vdash\ E'_i$ for $E'_i \in \widetilde{E'}$ or $\mathcal{L} \models \vdash\ E''_i$ for $E''_i \in \widetilde{E''}$ (we will consider abstractions later). Since we have that $\vdash_{\text{pre}}\ \mathcal{L}$, we have that those facts have been put in a conjunction Γ^s and succeed to prove a query that $\Gamma^s \vdash (op\ (op_2\ \widetilde{E'})\ \widetilde{E''}) : T$ and also $\Gamma^s \vdash t : T$. Since this query has been checked with universal quantifications over the variables of the query, any instantiations can be concluded. Therefore, we sure had $\mathcal{L} \models \vdash\ (op\ (op_2\ \widetilde{E'})\ \widetilde{E''})\gamma\ T$, that is $\mathcal{L} \models \vdash\ (op\ (op_2\ \widetilde{E'})\ \widetilde{E''})\gamma\ T$, but we can also conclude $\mathcal{L} \models \vdash\ t\gamma\ T$. That is, the reduction is type preserving. In our setting of logic programs, some arguments of op might be abstraction. In that case the query is hypothetical of the form $\vdash\ x : T_1 \Rightarrow \vdash\ R : T_2$, for some R argument of op. Now, there are two cases: either 1) t contains R simply as a variable, i.e. the step simply inherits R as it is, or 2) t contains $(R\ t')$ for some term t', i.e. the step applies a substitution. In case 1) we have that R will have the same type as in $(op\ (op_2\ \widetilde{E'})\ \widetilde{E''})$ and the query has checked that the whole resulting term turns out to be type preserving. In the second case, the fact that the query has been checked with the axiom (EQ-SUB), guarantees us that $(R\ t')$ matches the expected type as well, and, again, any instantiations of R and E will do as well.

C Type soundness

Type soundness of well-typed languages follows from progress and preservation in the usual way. If we have immediately a value or an error we are done. If it takes a step to some e'' then type preservation ensures that e'' is typeable and so the inductive hypothesis applies and we can conclude.