Extrinsically Typed Operational Semantics for Functional Languages

Anonymous Author(s)

Abstract

The research line on intrinsic typing has demonstrated that the type soundness of languages can be derived from type checking their implementation in a host language with strong meta-theoretic properties. In this paper, we take a different perspective in type checking language definitions for their soundness.

We present an extrinsic type system in which types are used to classify parts of the operational semantics of a language, and to model a common language design organization. The resulting typing discipline guarantees that the language at hand is automatically type sound.

A benefit of extrinsic typing is that, thanks to the use of types to model language design, our type checker has a high-level view on the language being analyzed and can report messages using the same jargon of language designers.

We have implemented our type system in the LANG-N-CHECK tool, and we have applied it to derive the type soundness of several functional languages, including those with recursive types, polymorphism, exceptions, lists, sums, and several common types and operators.

ACM Reference Format:

1 Introduction

The research line on intrinsic typing has demonstrated that type systems can be an effective tool for the analysis of languages. In this approach, a language is implemented within a host type theory with strong meta-theoretic properties [Church 1940; Harper and Stone 2000; Poulson et al. 2018; Rouvoet et al. 2020]. If the implementation type checks then the language is type sound. For example, if we can make a language definition type check within a type theory with exhaustive pattern-matching that is strongly normalizing then the progress theorem, which is one aspect of type soundness, automatically holds.

In this paper, we take a different perspective in the type checking of languages for their soundness: we intend to use types and type systems in the same way they are used in program analysis [Cardelli 2004; Pierce 2002], i.e., to model a high-level organization and discipline to ensure that some property holds. The most common way to type checking is with an external function that analyzes programs, that is, an extrinsic type system. Our goal is to present an extrinsic type system that accepts or rejects a programming language definition (syntax, type system, operational semantics) and guarantees that accepted languages are type sound.

A Meta Type System for Type Soundness. To avoid naming confusion, we use the term meta type system for a type system that analyzes a language definition rather than a program. Our main contribution is a meta type system that ensures that the components of a language definition (value declarations, typing rules, reduction rules, evaluation contexts, and so on) are all in order so that type soundness automatically holds. Our meta type system models a high-level organization and discipline that are not novel. Conversely, they capture invariants that language designers have been using for a long time, and makes them explicit and formally defined. For example, our meta type system traverses the grammar and typing rules of the language at hand in order to apply the common classification of operators as introduction forms (such as \( \lambda x.e \) in the simply typed \( \lambda \)-calculus (STLC)), elimination forms (such as application), derived operators (such as let and letrec), errors and error handlers (such as try). This classification paves the way for a high-level analysis of a language definition. For example, our meta type system can check the \( \beta \) rule of STLC \((\lambda x:T.e) v \rightarrow e[v/x]\) in the following way. (Below, the Classification meta-variable contains information about the role of operators and characterizes values. The application operator is named as app, and Section 3.2 presents complete details.)

\[
\text{Contexts} \vdash \text{Classification} : \text{funType} \\
\text{Classification} \vdash (\lambda x:T.e) : \text{value funType} \\
\text{Contexts} \vdash \text{app : } \{1,2\}
\]

This is a meta-level typing rule that type checks a reduction rule. We call these typing rules meta typing rules. Here, the first line of premises detects that the application is an elimination form of the function type, previously classified as such in Classification by other parts of the meta type system. It then requires that the argument being eliminated, which is highlighted and is sometimes called the principal argument [Harper 2012], be a value of the function type.

\[1\] The word extrinsic can be confusing when we go in the technical part of the paper, as our meta type system needs to analyze a type system which is itself extrinsic. We therefore use meta type system.
This materializes a common design in programming languages in which elimination forms manipulate values of some data type. Moreover, the arguments at positions 1 and 2 must be evaluation contexts for the application, as those arguments need to be evaluated for the reduction rule to apply. Contexts ⊢ app : {1, 2} enforces that the definition of evaluation contexts for application have E in both the first and second sub-expression positions, as in the grammar: 

$$E ::= (app E e) | (app v E)$$

Analogously, if the language at hand has lists and the reduction rule head (cons v1 v2) → v1, our meta type system would use an instance of the rule above and detect that head is an elimination form for lists. It would then check that (cons v1 v2) is a value for lists, and that the evaluation contexts are defined appropriately.

Our full meta type system captures design principles that are based on the above as well as other common language design invariants. It applies to functional languages with a typing relation of the form Γ ⊢ e : T and a reduction relation of the form e → e based on small-step semantics and evaluation contexts. Ultimately, we have proved that languages that conform to our meta type system are type sound.

**Implementation: LANG-N-CHECK.** We have implemented our meta type system in LANG-N-CHECK. The tool works with language definitions such as that in Figure 1, which contains the operational semantics of System F [Girard 1972; Reynolds 1974] with booleans. Language definitions use a domain-specific language that is close to pen&paper definitions in textbooks [Harper 2012; Pierce 2002] and research papers. The reason for this is that we wish to use the tool in courses on programming languages. Language definitions that type check are type sound. What happens if our meta type system fails? One of the differences between our approach and intrinsic typing is that since we directly model language design with types then the meta type checker has a high-level view on the language being analyzed and can report messages using the same jargon of language designers. For example, were we to forget the context (appT C e) at line 6, LANG-N-CHECK would reject the specification with the error message “The principal argument of the elimination form if is not declared as evaluation context, hence some programs may get stuck”. We are not aware of intrinsic type systems for languages whose type error messages refer to terms such as “principal argument” and “elimination form”.

We have applied LANG-N-CHECK to a plethora of functional languages to check their type soundness, including the simply typed λ-calculus and its variants with integers, booleans, pairs, lists, sums, tuples, fix, let, letrec, unit, universal types, recursive types, option types, exceptions, list operations such as append, map, mapi (also depends on the position in the list of the current element being processed), filter, filteri, range, list length, reverse, and the recursion of natural numbers (natrec). We have also considered different evaluation strategies for the features mentioned above: call-by-value, call-by-name and a parallel reduction strategy (both function and argument can evaluate non-deterministically), as well as lazy pairs, lazy lists, and lazy tuples, and both left-to-right and right-to-left evaluations. LANG-N-CHECK compiles the languages that type check successfully into the Abella theorem prover [Baelde et al. 2014] and automatically generates their machine-checked proof of type soundness. This gives us high confidence that, besides the theoretical guarantees of our paper, also our implementation is reliable.

**Close Related Work.** Cimini reports that LANG-N-CHECK has been used to teach programming languages theory [Cimini 2019]. We would like to clearly distinguish that work from this paper. That work briefly (and incompletely) describes the capabilities of LANG-N-CHECK, makes the point that error messages are informative, sets forth the thesis that a tool such as LANG-N-CHECK can be beneficial in a teaching context, and describes the studies that have been made, and that are to be made in the future. This paper, instead, provides a formal meta type system, gives a complete description of the organization principles that guide the meta type system, provides a proof of correctness, and reports on the many languages that have been checked as type sound using LANG-N-CHECK.

**Intrinsic vs Extrinsic Typing? So Far, Intrinsic Wins.** We have described the difference between intrinsic and extrinsic typing above, and also pointed out the benefits of the latter. Where does extrinsic typing of language stand as compared to intrinsic typing?

---

2 That paper refers to a tool with another name because LANG-N-CHECK is used within a larger tool.
A recent result by Poulsen et al have demonstrated that intrinsic typing can be applied to practical languages, and in particular to a large subset of Middleweight Java [Poulsen et al. 2018]. Our work is limited to the functional realm, and we acknowledge that the state of the art of intrinsic typing is substantially ahead.

The research line of intrinsic typing can be seen as occurring in two stages. Stage 1 laid down the idea and the practice of intrinsic typing, Stage 2 demonstrated that intrinsic typing can scale, as was recently shown by Poulsen et al. 2018.

Research in extrinsic typing of languages may need to follow a similar path. We believe that the research in this paper helps to establish Stage 1. It has been challenging to reach Stage 2 for intrinsic typing; we leave scaling up our approach for future work.

**Summary of our Contributions.** This paper makes the following contributions.

1. We devise a new approach to language analysis based on extrinsic typing of operational semantics. Our meta type system models a common high-level organization and discipline on language definitions that guarantee their type soundness (Section 3).

2. We prove that language definitions accepted by our meta type system are type sound (Section 4). In some sense, this result proves that the invariants that language designers have been using for years are correct at some general scale.

3. We implement the meta type system in LANG-N-CHECK (Section 5). We have applied the tool to derive the type soundness of several functional languages, including those with recursive types, polymorphism, exceptions, lists, sums, and several common types and operators.

The tool, the repository of language definitions, and their corresponding machine-checked proofs can be found at [link elided]. Proofs are in the Appendix.

## 2 Language Definitions

We briefly review how languages are defined in small-step operational semantics. This section serves to fix the terminology that we use in the rest of the paper.

Figure 2 shows the definition of $F_{exc}$, which is essentially System F extended with exceptions. The language defines a series of syntactic categories defined by a BNF grammar. The syntactic categories Type and Expression define the types and the expressions of the language. Next, language designers decide which expressions constitute values. These are the possible results of successful computations. Values are defined with the syntactic category Value. Similarly, the language designer may define which expression constitute the error, which drives the possible outcomes when computations fail. The error is defined with the syntactic category

\[
\begin{align*}
\text{Type} & : T ::= \top | T \to T | \forall X. T \\
\text{Expression} & : e ::= x | \lambda x : T . e | e \ e | \Lambda X . e | e [ T ] \\
& | \text{raise} e \ try \ e \ with \ e \\
\text{Value} & : v ::= \lambda x : T . e | \Lambda X . e \\
\text{Error} & : er ::= \text{raise} v \\
\text{Context} & : E ::= E \ e | \emptyset | E [ T ] | \text{raise} E \ try \ e \ with \ e \\
\text{Error Context} & : F ::= F e | v F | \text{raise} F
\end{align*}
\]

**Type System**

\[
\begin{align*}
\Gamma & \vdash \ e : T \\
\end{align*}
\]

**Expression**

\[
\begin{align*}
\Gamma, x : T \vdash e : T_1 & \Rightarrow \Gamma \vdash \lambda x : T . e : T_1 \to T_2 \\
\Gamma & \vdash \Lambda X . e : \forall X . T_2 \\
\end{align*}
\]

**Dynamic Semantics**

\[
\begin{align*}
\text{(T-VAR)} & : (\text{t-raise}) \\
\text{(T-APPL)} & : (\text{t-try}) \\
\text{(T-APPL)} & : (\text{t-tappt}) \\
\text{(T-APP)} & : (\text{t-tappt}) \\
\end{align*}
\]

**Figure 2.** Type system and dynamic semantics of $F_{exc}$.

Error. Context defines the evaluation contexts, which prescibe within what context we allow reduction to take place. Similarly, Error Context defines the error contexts, which are those contexts in which we are allowed to detect that an error has occurred. (They typically differ from evaluation contexts when the language includes error handlers.)

Next, $F_{exc}$ defines its type system with a relation of the form $\Gamma \vdash e : T$, which is inductively defined with a set of inference rules. The dynamic semantics here is of the form $e \rightarrow e'$ and is also inductively defined by a set of inference rules called reduction rules.

In the setting of small-step operational semantics and languages with errors, the following is the statement of type soundness. (\rightarrow^* is the reflexive and transitive closure of \rightarrow).

**Definition 2.1 (Type Soundness).** A language $L$ is type sound whenever for all expressions $e, e'$, and types $T$, if $\emptyset \vdash e : T$ and $e \rightarrow e'$ then either

- $e'$ is a value such that $\emptyset \vdash e' : T$,
- $e'$ is an error, or
- there exists $e''$ such that $e' \rightarrow e''$ and $\emptyset \vdash e'' : T$. 


3 A Meta Type System for Type Soundness

We work with a formal representation for language definitions. We define a language definition \( L \) as an 8-tuple

\[
(\text{Type}, \text{Expression}, \text{Value}, \text{Error}, \text{Context}, \text{Error Context}, \\
\text{Type System}, \text{Dynamic Semantics})
\]

where we place the components of Fig 2 in the tuple \( L \). To make an example, \( F_{\text{exc}} \) is represented as

\[
(\text{Types} = T \mapsto \exists T \Rightarrow T, \\
\text{Expression} = e ::= x | \lambda x : T.e | e e | \text{raise} e | \text{try} e \text{ with } e, \\
\text{Value} = v ::= \lambda x : T.e, \\
\text{Error} = v ::= \text{raise} v, \\
\text{Context} = E ::= E e | v E | \text{raise} E | \text{try} E \text{ with } e, \\
\text{Error Context} = F ::= F e | v F | \text{raise} F, \\
\text{Type System} = \{(\text{t-var}), (\text{t-abs}), (\text{t-tabs}), (\text{t-app}), (\text{t-tapp}), \\
(\text{t-raise}), (\text{t-try}), (\text{t-sub}), \\
(\text{t-beta}), (\text{t-try-success}), \\
(\text{t-try-raise}), (\text{t-ctx}), (\text{t-errCtx})\})
\]

where we simply wrote the name of rules although the tuple contains the actual inference rules.

In this section we define a meta type system with judgment \( \vdash L \) that makes sure that soundness automatically holds.

Our setting must accommodate syntax with some generality. Therefore, we assume that operators are defined in abstract syntax rather than concrete syntax. For example, we have if e e e rather than if e then e else e. We use \( xs \) as variables for expressions and (capitalized) \( Xs \) for types.

Without loss of generality, we assume that the type annotations of an operator all appear first, followed by types with a bound \( X \), followed by expression arguments, followed by expressions with a bound \( x \), and finally followed by expressions with a bound \( X \). We use a general shape that is similar to Harper’s abstract binding trees [Harper 2012] and is op \( T (X).T \overline{\sigma}(x).e(X).e \), where op is an operator, and \( \overline{\sigma} \) denotes sequences.

Our meta type system is an inference system that type checks parts of languages, including inference rules. To avoid confusion, we write the parts of the language being type checked in blue color. To help our presentation, we some times make examples with operators that are not in \( F_{\text{exc}} \).

Figure 3 contains the main judgement for \( \vdash L \), handled by rule [MAIN]. This rule simply checks all the components of the language in the way described in the following sections.

### 3.1 Meta Type Systems for the Syntactic Categories

The meta type system for type checking the syntactic categories are also in Figure 3. The language design invariants that are at play at this point are:

**Val** Any argument that is required to be evaluated to a value for a definition to apply must be declared as evaluation context. For example, **Val** enforces that if \( \text{cons } v v \) is declared as value then Context must contain productions of the form \( \text{cons } E _{\text{...}} \) and \( \text{cons } _{E} \), where _ can be \( e \) or \( v \) in both occurrences. Indeed, if one of these were missing, say the first, the expression \( \text{cons } (\lambda x . \text{nil}) \text{nil} \) would be stuck. This expression is not a value and does not take any step. Type soundness would then be jeopardized.

**Ctx** Evaluation contexts have no circular dependencies w.r.t. the evaluation of their arguments. For example, we must forbid context declarations such as \( \text{cons } \ E \ v \) and \( \text{cons } \ v \ E \). Such declarations make the expression \( \text{cons } (\lambda x . \text{nil}) \ (\lambda x . \text{nil}) \) stuck because the first argument does not start evaluating until the second is a value, while the second waits for the first to become a value, which won’t start evaluating.

**ErrCtx** Error contexts are evaluation contexts minus the error handler at the principal argument (where the error is expected to appear to be handled). This says that the context try \( F \) with \( e \) should not be an error context. Indeed, the reduction try (raise \( v \)) with \( e \rightarrow \text{raise } v \), which would apply by (err-ctx), should not take place because we expect the semantics of try to handle the error.

**Type Checking** Expression. We use \( \text{exp}^- \) to denote an expression derived by the grammar for Expression. The judgement \( \vdash \text{Expression} \) checks that each grammar production is of the form \( \text{op } T (X).T \overline{\sigma}(x).e(X).e \), which we have discussed previously. Notice that no other syntactic categories are allowed so far. This means that we rule out layered grammars such as expressions vs statements, which we plan to address in the future.

**Type Checking** Value. The judgement Context \( \vdash \text{Value} \) checks the value declarations. We refer to \( \text{exp}^- \) for \( \text{exp} \) in which we can use \( v \). There are two aspects that are checked for values. The first is that they make use of \( ev \), which denotes a variable that can be either \( e \) or \( v \). The second aspect been checked is the language design invariant **Val**, defined above. This check is done with Context \( \vdash \text{op} : \text{contextPositions} \), which informs about the set of argument positions that are declared as evaluation contexts for \( \text{op} \). For example, given Context with context declarations \( E ::= \langle \text{cons } \ E \ e \rangle | \langle \text{cons } v \ E \rangle \), we have Context \( \vdash \text{cons} : \{1,2\} \). Notice that \( E \) cannot appear underneath a binder, so we forbid evaluation underneath binders. Once we have computed the set contextPositions, we then check that the argument
positions in a value declaration that are vs, i.e., that are required to be values, are in that set. To make an example, the value declaration \( v := (\textup{cons} \ v \ v) \) is well-typed because the vs appear in argument positions 1 and 2, which are in \([1, 2]\), contextual positions for cons.

**Type Checking** Error. The judgement \( \Gamma \vdash E \vdash e \) checks the error declaration, if present. The invariant Val is checked here, as well. For example, raise \( E \) must be an evaluation context to prevent expressions such as raise \( (\lambda x : T. e . n i l) \) from being stuck and thereby invalidating type soundness.

**Type Checking** Context. We use \( evE \) to denote a variable that can be \( e \), \( v \) or \( E \), and we refer to \( ctx \) as those expressions where \( Es \) and vs can appear. For example, \( (\textup{cons} \ E \ v) \) is a valid \( ctx \). The judgement \( \Gamma \vdash \text{Context} \) checks that each context declarations have exactly one occurrence of \( E \). (We enforce this with the existential quantification with uniqueness \( \exists ! \).) It also enforces language design invariants Val and Ctx. The latter is ensured with the use of the function acyclic which performs this check through a graph representation of the dependencies at play. To be precise, we have an edge for each context declaration from the index position of \( E \) to the index positions of vs. Context declarations \( (\textup{cons} \ E \ e) \) and \( (\textup{cons} \ v \ E) \), i.e. left-to-right evaluation, induces the graph \([2 \rightarrow 1]\) which is acyclic. The bad context declaration \( (\textup{cons} \ E \ v) \) and \( (\textup{cons} \ v \ E) \), instead, induces the graph \([1 \rightarrow 2, 2 \rightarrow 1]\), which contains a cycle and is rejected.

**Figure 3.** Meta Type system: Main Judgement and Syntactic Categories, with the exception of Error Context, which is defined at the end of Section 3.2. \( ty \) is an expression derived by the grammar for Type. \( exp \) is \( exp^- \) in which we can use \( v \) variables. \( ctx \) is derived by the grammar for Context. \( exp^- \) is derived by the grammar for Expression. \( ev \), is a variable that can be either \( e \) or \( v \). \( evE \), is a variable that can be either \( e \) or \( v \) or \( E \).
The function acyclic performs a standard topological sort and so we omit its definition.

The last syntactic category that must be checked is Error Context. However, in order to check invariant ErrCtx we need to know whether an error handler exists. We therefore postpone this check until after we present the meta type system for Type System.

3.2 Meta Type System for Typing Rules

Figure 4 contains the meta type system for checking the typing rules of the language. The typing rules classify the operators of the language w.r.t. their role. Operators are classified as introduction forms, elimination forms, derived operators, the error and error handlers.

[T-MAIN] (Figure 4) is the main rule. It checks all the typing rules and returns the complete classification of the operators in the structure Classification, which we explain. When checking each typing rule, we return a role map \( B \) which can be of three possible forms: 1) \( op \mapsto R \), where \( op \) is an operator and \( R \) is the role that this operator plays in the language, 2) \( exp \mapsto c \) that denotes that \( exp \) is a value for the type constructor \( c \), or 3) \( exp \mapsto \Theta \), which means that \( exp \) is an error.

The meta type system collects all the role maps \( B_1, \ldots, B_n \), for which we check compatible\((B_1, \ldots, B_n)\). This check ensures that each operator has only one role. For example, cons cannot be both a value and an elimination form. Ultimately, the collection of all the role maps forms Classification.

Anatomy of a Typing Rule. Below we discuss how the meta type system checks each typing rule. Before proceeding we take a closer look at the typing rule for \( \text{let} \) and type abstraction in system \( F \).

\[
\begin{align*}
\text{(T-LET)} & \\
\Gamma, e_1 : T_1 & \quad \Gamma, x : T_1, e_2 : T_2 & \quad \Gamma, X : e : T \\
\Gamma & \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : T_2 & \Gamma & \vdash \lambda X.e : \forall X.T
\end{align*}
\]

We say that \( \text{let} \) is the subject of the typing rule (T-LET), and \( T_2 \) is its assigned type. We impose a common form for type environments: they can be used as \( \Gamma \vdash e : T \) and \( \Gamma \) can have bindings \( x : T \) (as in (T-LET)), and/or type variables \( X \) (as in (T-ABST)). We also impose that typing rules type check all their \( e \) arguments, as done above and in virtually any typing rule we know.

[T-VALUE] detects the introduction forms following the common design principle that introduction forms build values of some data type. To this aim, [T-VALUE] imposes that the assigned type of the typing rule has the form \( c.T \), as highlighted, that is, a type constructor applied to arguments. [T-VALUE] also checks that \( op \) forms a value. To make some examples, below [T-VALUE] is instantiated to check

\[
\begin{align*}
\text{(T-LAMBDA)} & \\
\Gamma, x : T_1 & \quad e : T_2 & \quad \Gamma, \lambda x : T.e & \quad \text{value} \\
\Gamma & \vdash \lambda : T.e & \quad \Gamma, x : T_1, e : T_2 & \quad \Gamma & \vdash \lambda x : T.e
\end{align*}
\]

[T-ELIM] classifies elimination forms following a common pattern:

**Elim** elimination forms manipulate/inspect values of some data type at their principal argument.

Accordingly, [T-ELIM] imposes that the typing rule assigns a complex type \( c.T \) to the principal argument of \( op_1 \). To simplify our presentation, we assume without loss of generality that the principal argument is always the first expression variable \( (e_1) \). Since the operator must manipulate this argument, it is expected to have at least a reduction rule of the form \( \text{op}_1 \ (\text{op}_2 \ \text{args}_2) \ \text{args}_1 \rightarrow \text{exp} \), that is, the principal argument is a complex expression, as highlighted. Reduction rules typically follow this pattern, examples are

\[
\begin{align*}
\text{app} (\lambda x.e) c & \rightarrow e[v/x] \text{ for the elimination form app,} \\
\text{head} (\text{cons} v_1 v_2) & \rightarrow v_1 \text{ for the elimination form head.}
\end{align*}
\]

To distinguish whether the operator is an elimination form rather than an error handler, we check that \( op_2 \) is a value.

To make some examples, below we instantiate [T-ELIM] for checking (T-APP), the typing rule for the list operator head, and (T-TAPP).

\[
\begin{align*}
\text{(T-TABS)} & \\
\Gamma, x : T_1 & \quad e_2 : T_2 & \quad \Gamma, X : e : T \\
\Gamma & \vdash \text{let} x = e_1 \ \text{in} \ e_2 : T_2 & \Gamma & \vdash \lambda X.e : \forall X.T
\end{align*}
\]

\[
\begin{align*}
\text{(T-LIST)} & \\
\Gamma & \vdash \text{cons} v_1 v_2 : \text{List} \\
\Gamma & \vdash \text{head} (\text{cons} v_1 v_2) : v_1 & \Gamma & \vdash \text{List}
\end{align*}
\]

\[
\begin{align*}
\text{(T-ERRHANDLER)} & \\
\Gamma & \vdash \text{error} & \Gamma & \vdash \text{error handler}
\end{align*}
\]

\[
\begin{align*}
\text{app} (\lambda x : T.e) v & \rightarrow e[v/x] \in \text{Dynamic Semantics} \\
\Gamma & \vdash \text{app} e_2 : T_2 & \Gamma & \vdash \text{app} \ e_2 : T_1 & \Gamma & \vdash \text{app} \ e_2 : T_1 & \Gamma & \vdash \text{app} \ e_2 : T_1
\end{align*}
\]

\[
\begin{align*}
\text{head} (\text{cons} v_1 v_2) & \rightarrow v_1 \in \text{Dynamic Semantics} \\
\Gamma & \vdash \text{head} & \Gamma & \vdash \text{head}
\end{align*}
\]
Figure 4. Meta Type system: Typing Rules. In this figure we use the convention that sequences $\overline{T}$ contain types, which may be under a binder. Here, sequences $e_1, \ldots, e_n$ are expressions, which may be under a binder. Sequences $\overline{\text{args}}$ contain variables, be under a binder. Also, sequences in this figure have distinct variables.

To make an example, we instantiate [T-ERRHANDLER] for the typing rule (t-TRY).

\[
\begin{align*}
\text{try (raise v) with e } & \rightarrow (e, v) \in \text{Dynamic Semantics} \\
\text{raise } & \in \text{Error} \\
\text{Value } & \vdash e_1 : \overline{T} \\
\text{Error } & \vdash e_2 : \overline{T} \rightarrow T \\
\text{Dyn. Sem. } & \vdash \text{try } e_1 \text{ with } e_2 : \overline{T} \\
\end{align*}
\]

[t-errHandler] classifies derived operators. One of their characteristics is that derived operators do not primitively manipulate complex data, rather they pass it on to other operators. For this reason, [T-DERIVED] checks that all the reduction rules for the operator are of the form $(op \overline{\text{args}}) \rightarrow exp$, where $\overline{\text{args}}$ are exclusively variables. Therefore, $op$ does not pattern-match its arguments into complex expressions. To make some examples, we instantiate [T-DERIVED] for the
typing rules of fix and letrec.

\[
\text{fix } v \rightarrow v (\text{fix } v) \in \text{Dynamic Sem.}
\]

\[
\begin{array}{c|c|c}
\text{Value} & \Gamma \vdash e : T & \Gamma \vdash \text{fix } e : T \\
\hline
\text{Error} & \Gamma \vdash e : T & \Gamma \vdash \text{fix } e : T \\
\hline
\text{Dynamic Sem.} & & \\
\end{array}
\]

\[
\text{letrec } x = e_1 \text{ in } e_2 \rightarrow e_2 ((\text{fix } (\lambda x.e_1))/x) \in \text{Dyn. Sem.}
\]

\[
\begin{array}{c|c|c}
\text{Value} & \Gamma, x : T_1 \vdash e_1 : T_1 & \Gamma \vdash \text{letrec } x = e_1 \text{ in } e_2 : T_2 \\
\hline
\text{Error} & \Gamma \vdash e_1 : T_1 & \Gamma \vdash \text{letrec } x = e_1 \text{ in } e_2 : T_2 \\
\hline
\text{Dynamic Sem.} & & \\
\end{array}
\]

\[
| \text{T-ERROR} | \Gamma \vdash e : T \rightarrow \text{letrec } x = e_1 \text{ in } e_2 : T_2 \\
\]

\[
\text{Preserv} \quad \text{Naturally, reductions must be type preserving, i.e., in a reduction } e \rightarrow e' \text{, } e \text{ and } e' \text{ have the same type.}
\]

\[
\text{ErrSucc} \quad \text{Error handlers must handle both the error and the case of success.}
\]

\[
\text{ValRed} \quad \text{Principal arguments, and any argument that is required to be evaluated to a value for a reduction rule to fire, must be declared as evaluation context.}
\]

3.3 Meta Type System for Dynamic Semantics

Figure 5 contains the meta type system for checking the dynamic semantics. The language design invariants that we enforce at this point are:

\[
\text{AllVal} \quad \text{Each elimination form of some type manipulates all the values of that type.}
\]

\[
\text{Preserv} \quad \text{Naturally, reductions must be type preserving, i.e., in a reduction } e \rightarrow e', e \text{ and } e' \text{ have the same type.}
\]

\[
\text{Preserv} \quad \text{Naturally, reductions must be type preserving, i.e., in a reduction } e \rightarrow e', e \text{ and } e' \text{ have the same type.}
\]
Move Maps \( C \ ::= \ op \mapsto M \)

Moves \( M \ ::= \) eliminates \( \op \) | handlesError | handlesSuccess | moves

Moves exhausts Classification whenever

\[
\{ \op_1 : \text{elim } c, \op_2 : \text{intro } c \} \subseteq \text{Classification implies } \op_1 : \text{eliminates } \op_2 \in \text{Moves}, \text{and}
\]

\[
\{ \op_1 : \text{errorHandler}, \op_2 : \text{error} \} \subseteq \text{Classif. implies } \{ \op_1 : \text{handlesError}, \op_1 : \text{handlesSuccess} \} \subseteq \text{Moves}.
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash \text{Dynamic Semantics}
\]

\[
\text{ErrCtx} = \begin{cases} ((\text{Err-ctx})], & \text{if Classification } \vdash \op : \text{error} \\ \emptyset, & \text{otherwise} \end{cases}
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash \phi_1 : M_1
\]

\[
\ldots
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash \phi_n : M_n
\]

\[
M_1, \ldots, M_n \text{ exhausts Classification}
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash \text{ErrCtx} \cup \{(\text{ctx}), \phi_1, \ldots, \phi_n\}
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op_1 (\op_2 \ args^1) \ args^2) : ty \Rightarrow \Gamma^s \text{ Type System} | \Gamma^s \vdash_{\text{rhs}} \exp : ty
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op_1 (\op_2 \ args^1) \ args^2) \rightarrow \exp : \op_1 \rightarrow \text{eliminates } \op_2
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op \ u \ args^1) : ty \Rightarrow \Gamma^s \text{ Type System} | \Gamma^s \vdash_{\text{rhs}} \exp : ty
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op \ u \ args^1) \rightarrow \exp : \op \rightarrow \text{handlesError}
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op \ u \ args^1) \rightarrow \exp : \op \rightarrow \text{handlesSuccess}
\]

\[
\text{Context} | \text{Classification} | \text{Type System} \vdash (\op \ u \ args^1) \rightarrow \exp : \op \rightarrow \text{moves}
\]

\[
\text{Figure 5. Meta Type system: Dynamic Semantics}
\]

check to ensure that AllVal and ErrSucc are fulfilled. This is done with Moves exhausts Classification.

\[\text{[r-elimination]}\]

handles the case of reduction rules for elimination forms. The first aspect that \[\text{[r-elimination]}\] enforces is the language design pattern Elim. To this aim, we check that \(\op_1\) is an elimination form for a type \(c\) and that \(\op_2\) is a value of the same type \(c\). Next, \[\text{[r-elimination]}\] covers ValRed. In particular, we check that the principal argument (position 1) is an evaluation context. Not only that, we check that any other argument of \(\op_1\) that is required to be a value has an evaluation context defined. Lastly, \[\text{[r-elimination]}\] and all other meta typing rules of Figure 5 check that the overall reduction is type preserving. This is done with the last line of the premises. We postpone the discussion of type preservation until Section 3.4.

To make an example, we instantiate [r-elimination] for (beta) and one of the reduction rules for head. (Below and
throughout this subsection we have elided the type preservation check, which we discuss later).

We first detect that the operator at hand is an error handler, specifically, the following way.

We observe that a call-by-name STLC with reduction rule (r-try-error) handles errors similar to the way that the reduction rules for fix and let handles operators such as fix and let. To make an example, the reduction rule for fix and let are handled as follows. (Typically, evaluation contexts are address the reduction rule that specifies the behavior of the error handler when an error did not occur. We first detect that the operator is an error handler. Then, we check that a value v appears in principal argument position. To make an example, [r-success] checks (r-try-success) in the following way.

3.4 Meta Type Checking for Type Preservation

A mandatory part of type soundness is that reduction rules must be type preserving. Given a reduction rule exp → exp', we have to ensure that the types of exp and exp' coincide. This is done with the last line of premises in each rule of Figure 5. Differently from the rest of the meta type system, where organizational principles play a role, we are unaware of language organization methods that ensure type preservation. Language designers typically simply write reduction rules making sure that the type on the left-hand side and that on the right-hand side coincide. Works such as [Pfenning and Schürmann 1999; Schürmann 2000] and [Grewe et al. 2015] have demonstrated that type preservation checks can be automated with theorem provers. Our approach adapts that method and integrates it within the formalism of our meta type system.

Ideally, we need to check that for all Γ, Γ ⊢ exp : T implies Γ ⊢ exp' : T, but checking all possible type environments is prohibitive. Therefore, we approximate such a check with the use of a symbolic type environment, which contains the assumptions on the types of the variables. Our first check is Type System ⊢bs exp : ty ↝ Γ⁺, which means that according to the type system we have that exp has type ty under assumptions Γ⁺, which are computed as output. The way we build Γ⁺ for exp is by inverting the typing rules used to type check exp. As we essentially use the method in [Grewe et al. 2015; Pfenning and Schürmann 1999; Schürmann 2000] we omit this part here, though it can be found in Appendix B.

To make an example, head (cons v₁ v₂) → v₁ gives Γ⁺ = Γ + v₁ : T, Γ + v₂ : List T. We also obtain its type T. Then, we check Type System ⊢Γ⁺ + exp' : ty, which means that exp' has type ty according to the type system and Γ⁺, which is an input this step. This check solves a provability problem: Type System U Lemmas ⊢ Γ + v₁ : T, Γ + v₂ : List T ⊨ T that is, the formula Γ + v₁ : T can be proved within the inference system formed by the type system, substitution lemmas (Lemmas, see below), and the formulae in Γ⁺.

Lemmas contains the well-known substitution lemmas:
We have implemented our meta type system in Extrinsically Typed Operational Semantics for Functional Languages. We have proved the correctness of our meta type system: Language definitions that are well-typed are guaranteed to be type sound.

Theorem 4.1 (Correctness of $\vdash L$).

Given a language definition $L$, if $\vdash L$ then $L$ is type sound.

The proof is in Appendix A. Once the language definition is type checked, it has a predictable structure, and we can automate the proof of Wright and Felleisen’s [Wright and Felleisen 1994]. The proof replays canonical form lemmas, the progress theorem, type preservation theorem and, ultimately, the type soundness theorem.

5 Evaluation

We have implemented our meta type system in LANG-N-CHECK. This tool is written in OCaml and type checks language definitions that are written in a rather intuitive domain-specific language. An example of language definition is in Figure 1 of the introduction section.

The type preservation checks for reduction rules are done by compiling languages into the Abella theorem prover [Baelde et al. 2014] and by running the queries described in Section 3.4.

Deriving Type Soundness of Languages. We have used LANG-N-CHECK to derive the type soundness of several languages: STLC, STLC with integers, booleans, pairs, lists, sums, tuples, fix, let, letrec, unit, universal types, recursive types, option types, exceptions, list operations such as append, map, mapi (also depends on the position in the list of the current element being processed), filter, filteri, range, list length, reverse, and the recursion of natural numbers (natrec).

We have also considered different evaluation strategies for the features mentioned above: call-by-value, call-by-name and a parallel reduction strategy (both function and argument can evaluate non-deterministically), as well as lazy pairs, lazy lists, and lazy tuples, and both left-to-right and right-to-left evaluations.

As we said above, LANG-N-CHECK compiles languages into the Abella theorem prover. If they type check successfully LANG-N-CHECK also generates their machine-checked proof of type soundness. We therefore have a mechanized proof for all the languages that we have tested. This gives us high confidence that, besides the theoretical guarantees of our paper, also our implementation is reliable.

Informative Error Messages. LANG-N-CHECK has been used to teach programming languages theory [Cimini 2019]. That paper points out that LANG-N-CHECK can report informative error messages to the user. We provided an example in the introduction section. To show other examples, consider the listing in Figure 1 of the introduction.

- We were to miss one of the reduction rules for the if operator, say the rule at line 22, LANG-N-CHECK would print the error message "Operator if is elimination form for the type bool but does not have a reduction rule for handling one of the values of type bool: value tt”.
- We were to miss the context declaration (app C v) at line 6, LANG-N-CHECK would reject the language definition and print the error message "Reduction rule for elimination form app requires argument 2 to be a value but that argument is not declared as evaluation context, hence some programs may get stuck”.
- If the language definition had contexts declarations (app C v) and (app v C), LANG-N-CHECK would reject the definition and print the error message "Evaluation contexts have cyclic dependencies, hence some programs may get stuck”.

Limitations and Future Work. We shall discuss some limitations of our meta type system. The shape of the typing relation and reduction rules are enforced to be $\Gamma \vdash e : T$ and $e \rightarrow e$, respectively, which is standard for purely functional languages, and does not capture languages with effects, typestate, and stores, to name a few. We plan to address those features in the future. Similarly, the grammar for types forbids dependent types and refinement types. These are advanced features that we leave for future work. We also plan to extend our work to give the user the possibility to define layered grammars such as expressions vs statements.

Languages are restricted to an organization that is based on introduction forms and elimination forms, as well as other roles. Some calculi may not fit this schema even within the realm of functional languages. For example, the gradually typed $\lambda$-calculus has a cast operator that is sometimes a value and some times performs computational steps [Siek and Taha 2006], and therefore defies this schema.

Our setting only allows unary binding [Cheney 2005]. This has been sufficient for the languages that we have checked. However, such restricted approach leaves out some languages, especially those that are non-lexically scoped. We plan to adopt more sophisticated binding structures. In this regard, we are looking at integrating scopes and frames [Néron et al. 2015; Poulsen et al. 2016], which have been successfully employed in other systems [Poulsen et al. 2018].

6 Related Work and Future Work

Intrinsically Typed Languages. Poulsen et al have demonstrated that intrinsic typing can derive the soundness of a large subset of Middleweight Java [Poulsen et al. 2018]. The...
applicability of their work is much more general than that of this paper in at least two aspects. First, they capture a rather complex language with a store, which is out of the scope of our meta type system. Second, they employ a more general binding structure to accommodate classes. We make use of a less expressive unary binding approach.

The title of our paper is inspired by theirs\(^3\). 1) We use extrinsic typing rather than intrinsic typing. As we have explained in the introduction, the two are conceptually different approaches. Also, our meta type system can print error messages that use the jargon of language designers, which is not the case for [Poulsen et al. 2018].

2) We work with operational semantics formulations rather than definitions of interpreters. The latter are Agda implementations of languages, whose evaluator must be augmented with the so-called fuel to help the compiler derive termination [Owens et al. 2016]. In contrast, our meta type system may appeal a part of the community for working with language definitions that are in 1-1 correspondence with pen&paper formulations. Cimini [Cimini 2019] observes that students could use lang-n-check after having been taught the type system and operational semantics notation with the TAPL textbook [Pierce 2002]. Moreover, Poulsen et al. target a big-step semantics, which does not need evaluation contexts. In contrast, we use small-step semantics and our meta type system contributes a formal treatment for evaluation contexts.

3) Our title makes it clear that the scope of our paper is that of functional languages rather than imperative languages. In this regard, we acknowledge that the state of the art of intrinsic typing is substantially ahead.

Automated Theorem Proving. The seminal work of Schürmann and Pfennig shows that the type soundness of non-trivial functional languages can be automatically established in the LF-based theorem prover Twelf [Pfenning and Schürmann 1999; Schürmann 2006]. The Veritas tool automates the verification of the type soundness of languages by compiling them into a first order theorem prover, and by checking that suitable formulae hold [Grewe 2019; Grewe et al. 2016, 2017, 2015]. The way our meta type system checks type preservation stems directly from these works, but our approach to progress is different. Differently from these works, we establish progress by encoding language invariants as a typing discipline rather than querying a theorem prover with formulae. To our knowledge, those tools also do not report error messages that refer to language design terminology, such as the messages in Section 5.

Type Sound Language Extensions. Schwaab and Siek [Schwaab and Siek 2013], and Delaware et al [Delaware et al. 2013] provide solutions to the composition of already existing proofs of type soundness. Lorenzen and Erdweg’s work on SoundX [Lorenzen and Erdweg 2016] proposes a method to establish the soundness of language extensions, making sure that their desugaring is correct. In contrast to these works, our meta type system can check that a language definition given from scratch is type sound in the first place. These works are somehow orthogonal to ours, and we plan to integrate their insights in our context.

Model Checking. Roberson et al. [Roberson et al. 2008] propose a model checking approach to type soundness in which configurations are generated, steps are computed, and then type checked. Similarly to testing, this approach can detect bugs but cannot guarantee type soundness.

Language Workbenches and Semantics Engineering Tools. There are several tools that support the specification of languages, such as Ott [Sewell et al. 2007], Lem [Mulligan et al. 2014], the K framework [Roșu and Serbanuță 2010], and PLT Redex [Felleisen et al. 2009]. Furthermore, language workbenches also assist language designers with a plurality of language services [Erdweg et al. 2013, 2015; Fowler 2005], and some have extensive support for implementing type systems, such as MPS [Voelter and Solomatov 2010], Xtext [Bettini 2011; Efftinge and Spönenmann [n.d.]], and Sugar [Erdweg et al. 2011], to name a few. However, these systems do not establish the type soundness of the languages being defined\(^4\).

7 Conclusions

In this paper, we have presented an extrinsic type system that establishes the type soundness of functional languages defined in operational semantics. Our meta type system is based on a typing discipline that models a high-level language design organization. We have proved that our type system is correct. Furthermore, we have implemented our work in the lang-n-check tool and we have applied it to a plethora of functional languages.

Intrinsic typing has certainly demonstrated that it can scale well, thanks to recent advances in its theory and practice [Poulsen et al. 2018]. We have informally referred to such advances as Stage 2 in the introduction. This paper introduces the extrinsic typing approach to type soundness and we believe that it helps to establish Stage 1 for this research line. In the future, we plan to tackle sophisticated languages with extrinsic typing.

References


\(^3\)For easy of reference, the title is: “Intrinsically Typed Definition Interpreters for Imperative Languages”.

\(^4\)A plan to equip Spoofax [Kats and Visser 2010] with verification capabilities has been reported [Visser et al. 2014] but we are not aware of any integrated component for it yet.
A Proof: Correctness of the Meta Type System

We follow the syntactic approach to type soundness of Wright and Felleisen. We generate canonical form lemmas, progress, type preservation and, ultimately, type soundness.

A.1 Progress Theorem

**Progress Theorem:**

\[
\text{If } \vdash \mathcal{L} \text{ then } \\
\text{if } \emptyset \vdash e : T \text{ then } e \text{ progresses.}
\]

An expression \(e\) progresses whenever either \(e\) is a value, \(e\) is an error, or there exists \(e'\) such that \(e \rightarrow e'\).

**The Main Progress Theorem.** Assume \(\vdash \mathcal{L}\) and \(\vdash e : T\). The proof is by induction on \(\vdash e : T\). As \(\vdash e : T\) is provable, it means that there exists a typing rule \(\phi\) of the form 

\[
\frac{f_1, \ldots, f_n}{\text{conclusion}}
\]

that is ‘satisfied’, that is, there exists a substitution \(\gamma\) from logical variables (of the rule) to terms such that \(f_i\gamma\) are all satisfied, and \(\text{conclusion} \vdash e : T\).

Since \(\vdash \mathcal{L}\) and \(\phi\) is a typing rule of \(\mathcal{L}\) then \(\phi\) has been type checked. Therefore, \(e\) has a top level operator \(op\) and arguments, as this is the shape imposed for typing rules.

It also means that all the arguments of the operator, say \(\vec{v}\), are the subject of a typing premise, because our type checker imposes that shape for typing rules. Therefore we can apply the inductive hypothesis on the arguments \(\vec{v}\), but in particular, we apply the inductive hypothesis for those arguments \(e_i\) with index \(i \in \text{contextPosition}\) from Context \(\vdash op : \text{contextPosition}\), i.e., only to the contextual arguments. As Context has been meta type checked, \(e_i\) is not under a binder. Therefore the typing premise for \(e_i\) is of the form \(\Gamma, \forall e_i : ty_i \vdash \gamma\), and not with \(\Gamma, x : T\). Therefore we can apply the inductive hypothesis to \(\vdash e_i\gamma : ty_i\) for all of them, and we derive that \(e_i\gamma\) progresses.

We call the Progress Lemma for \(op\) (defined below), which expects exactly those progress assumptions \(e_i\gamma\) progresses.

**Progress Lemma for \(op\).** Given an operator \(op\) in \(\mathcal{L}\), and given Context \(\Gamma \vdash op : \{1, \ldots, n\} \in \text{Context}\). Without loss of generality, here contextual arguments appear first as a presentational aid. The progress theorem for \(op\) is:

\[
\text{if } \vdash \mathcal{L} \text{ then if } (H) \vdash (op e_1 \ldots e_n \vec{e}) : T, \text{ and progress } e_1, \ldots, \text{ progress } e_n \text{ then progress } (op e_1 \ldots e_n \vec{e})
\]

The proof is by case analysis on \((H)\) above. If \(\vdash (op e_1 \ldots e_n \vec{e})\) has been proved, than it has been proved with a typing rule \(\phi\). We do case analysis on all progress \(e_1, \ldots, \text{ progress } e_n\), but in a suitable order. Since \(\vdash \mathcal{L}\) we have that acyclicContext. Therefore we can choose an order for argument positions 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 value of the argument \(j\) of \(op\). As this order must exist, without loss of generality, to aid the proof we fix that the evaluation contexts are left-to-right. 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: 1) value \(e_1\) \& progress \(e_2\), 2) step \(e_1\) \& progress \(e_2\), 3) error \(e_1\) \& progress \(e_2\). We then 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 1) value \(e_1\) \& value \(e_2\), 2) value \(e_1\) \& step \(e_2\), 3) value \(e_1\) \& error \(e_2\). And we are at the leftmost child: value \(e_1\) and value \(e_2\), thanks to the invariant.

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 case, all arguments that are requested to be value by the evaluation context of \(op\) are indeed values.
The proof is by case analysis on role such that Classification \(\vdash \text{op} : \text{role}\), which exists because \(\vdash L\) and \(\cdots \vdash \text{Type System}\): Classification and \(\text{op} \ \text{must}\) have had a typing rule.

- \(\text{op} \ \text{args} : \text{value} \ c\): As Classification \(\vdash \text{op} \ \text{args} : \text{value} \ c\) then \(\text{op} \ \text{args} \in \text{Values}\). We strive to apply the value definition for \(\text{op}\), however, this definitions may use \(\nu_j\), i.e. those arguments are required to be values for the value definition to apply. As we have type checked \(\text{op}\), these arguments must be in contextPosition and therefore they all are involved in a \text{progress} \(e_1, \ldots\), \text{progress} \(e_n\) assumptions. Therefore, we have done a case analysis on all of them, we are in the case where all of them are values, and so the definition applies.

Next, we are left with two cases: 1) all arguments are values except for one which is \text{step} \(e_n\) and 2) all arguments are values except for one which is \text{error} \(e_n\). Now, we devise a generalized argument: when we are done with 1) and 2), we find ourself in the context of the case analysis of \text{progress} \(e_{n-1}\) because we have just proved the \text{value} \(e_n\) case of \text{progress} \(e_{n-1}\). Therefore, we need now to prove the cases 1) and 2) of \text{progress} \(e_{n-1}\). Once we prove these two, we have proved the value case of the case \text{progress} \(e_{n-2}\) and we will need to prove cases 1) and 2) of \text{progress} \(e_{n-2}\) and so on. All of these cases are treated uniformly, thanks to the fact that we have opened the case analysis on the progress of arguments in the order that is dictated by the topological sort.

We have

- \(\text{STEP}\): case \(e_j \rightarrow e_j'\). Moreover, we have that all arguments of \(\text{op}\) before \(j\) are all values. Therefore we have apply the evaluation context \((\text{op} \ \nu \ E_j \ \nu)\) to prove a step \((\text{op} \ \nu \ E_j \ \nu)j \rightarrow (\text{op} \nu \ E_j \ \nu)j'\), and so we progress.

- \(\text{ERR}\): \(e_j\) is an error. Moreover, we have that all arguments of \(\text{op}\) before \(j\) are all values. Therefore we have apply the evaluation context \((\text{op} \ \nu \ E_j \ \nu)\). Since we have type checked Error Context(op) and we are in the context of a value \((\text{op} \ \text{args} : \text{value} \ c)\) (rather than an error handler), we have that \((\text{op} \ \nu \ E_j \ \nu)\) is an error context. Therefore, prove a step \((\text{op} \ \nu \ E_j \ \nu)j \rightarrow e_j\), i.e. a step to the error, with \text{ctx-err}. So, we progress.

- \(\text{op} : \text{elim} \ c\): Then \([\text{T-elim}]\) has meta type checked \(\phi\). This means that the \(\phi\) has a typing premise for the principal argument of the form \(\text{e}_1 : (c \ \text{T})\). Also, we are exploring the case where \(e_1\) is an error. Therefore, we apply the Canonical Forms Lemma for \(c\) (described in the following paragraph). This means that \(e_1 \rightarrow t_1 \lor \ldots \lor t_m\) (this is notation from next paragraph) with all \(t_k\) be with form \((\text{op} \ \text{arg} \)s) and \(\text{op}_2 \in \text{Values}\). Then, since Dynamic Semantics has been meta type checked, it must have produced Moves that passed the exhaustiveness check exhaust. Therefore we have \(\text{op} : \text{elim} \ c\) and Classification\((\text{op}_2) = \text{value} \ c\), which enforces that a reduction rule of the form \(r = (\text{op} \ (\text{op}_2 \ \nu \ E_j \ \nu)) \rightarrow \text{exp}\) exists. As we are in the case where the principal argument is a value, and also all other argument in \((\nu \ E_j \ \nu)\) are value, we have that this reduction rule fires and takes a step, and so we progress.

The other cases for \(\text{op} : \text{elim} \ c\) are proved as in \text{STEP} and \text{ERR} above.

The other operators are easier to handle. Below we only handle the progress case \text{value} because the other cases are handled as in \text{STEP} and \text{ERR} above. Below, we make an exception for the error handler, for which the cases \text{error} and \text{step} are proved differently.

- \(\text{op} : \text{error}\): This is handled similarly as to values, though it satisfies progress because the error definition applies rather than a value definitions.

- \(\text{op} : \text{derived}\): Then \(\phi\) has been meta type checked by \([\text{T-derived}]\). Therefore, it exists a rule \((\text{op} \ \nu \ \nu') \rightarrow e_j\). Also, all those \(\nu_i\) are in evaluation contexts, and so have been involved in the progress hypothesis above and a case analysis on all of them is being performed. So we are in the case where they are all values and thus the reduction rule can apply because our type checker imposes that the shape of the reduction rule does \text{not} pattern-matches arguments for the derived operator \(\text{op}\), but imposes that it requires the arguments to be \text{vs}, and we are in the case where those are all values indeed. So the rule fires, and we progress.

- \(\text{op}_1 : \text{errHandler}\): Then \(\phi\) has been meta type checked by \([\text{t-errHandler}]\). Then, since Dynamic Semantics has been meta type checked, it must have produced Moves, which by our exhaustiveness check it contains \(\text{op}_1\), \text{handlesSuccess}. This means that it exists a reduction rule of the form \((\text{op}_1 \ \nu \ \nu) \rightarrow \text{exp}\). We are in the case where the arguments are values, and so they satisfy that \(\nu\). So this rule fires, and we progress. Next, we need to prove \text{error} and \text{step}, which differently from the previous cases they are not handled with \text{STEP} and \text{ERR}. \text{step}: Here we have that the argument at error position must be an evaluation context and so it has a step, \text{error}: This case cannot be satisfied by appealing to error contexts as in \text{ERR} because the type checking of Error Context excluded that context. However, exhaustiveness of Moves imposes that it has \(\text{op}_1\), \text{handlesError}, which in turn, implies that there exists a reduction rule that fires when the error is at error position. Therefore, this reduction rule can apply, prove a step, and we progress.
Canonical Forms Lemma for $c$.

Theorem A.1. if $\vdash \rho$ and $(H) \vdash e : (c \overrightarrow{T})$ and $(V)$ value $e$ then $e = OR((\rho \text{args}) \cup \text{Classification} \vdash (\text{op args}) : \text{value c})$.

The proof is by case analysis on $(H)$ above. If $\vdash e : (c \overrightarrow{T})$ has been proved, then it has been proved with a typing rule $\phi$.

As Type System has been meta type checked, the conclusion of the typing rule has a form of an expression with a top level operator $\text{op}$.

We reason by cases analysis on the classification of $\text{op}$, if Classification $\vdash (\text{op args}) : \text{value c})$ then it is one of the values in $OR((\rho \text{args}) \cup \text{Classification} \vdash (\text{op args}) : \text{value c})$. Classification $\vdash (\text{op args}) : \text{value c}')$ with $c' \neq c$ is not a case that can happen because $T$-VALUE would impose that $(\text{op args})$ is typed at a type $(c' \overrightarrow{T})$. Instead by assumption $H$ we have $\vdash e : (c \overrightarrow{T})$. The other cases: Classification $\vdash (\text{op args}) : \text{elim c}), \text{derived c}), \text{error c}), \text{errorHandler c})$, are such to contradict the assumption $(V)$.

A.2 Type Preservation

Type Preservation Theorem:

if $\vdash \rho$ then

for all expressions $e, e'$ and types $T$,
if $\emptyset \vdash e : T$ and $e \rightarrow e'$ then $\emptyset \vdash e' : T$

The proof is by induction on $e \rightarrow e'$.

$e \rightarrow e'$ is provable, then there exist a reduction rule $\phi$ with conclusion concl $= exp \rightarrow exp'$ and a substitution $\gamma$ such that concl $= e \rightarrow e'$.

Reduction rules: Then $\phi$ has been type checked and we have dots $\vdash \phi : \text{eliminates op}$, or some other move. We show the case dots $\vdash \phi : \text{eliminates op}$ because the cases for the other moves are subsumed by this. In particular, error handlers follow the same line, and derived operators are even simpler as they do not have a nested expression that is pattern-matched.

We have to prove that $(\text{exp} y)$ is of type $T$.

As dots $\vdash \phi : \text{eliminates op}, \text{we know that}$

$e = (\text{op} \overrightarrow{\text{args}}) \rightarrow \text{args2})$. As $\emptyset \vdash e : T$, then this typeability formula must have been proved with a typing rule of the form $\Gamma \vdash \text{op}$...

Here, $ps$ are the inverted premises that contribute to populate $\Gamma'$ for the type preservation check. As the typing rule has been type checked, too, the shape must be such that all the arguments that are expressions are recursively type checked. We therefore have a premises in $\text{ps}$ that type checks $(\text{op args})$, and this formula has been proved, too. Therefore there is another typing rule that has type checked that formula, and has the form $\Gamma \vdash \text{op}_{2}$...

where $ps'$, too, contributes to populate $\Gamma'$. Now we have that $(\#)$ Type System $\cup \text{Lemmas} \cup \Gamma' \vdash \text{exp} : T$ and that $(\star)$ Type System $\cup \text{Lemmas} \cup \Gamma' \vdash \text{exp}' : T$. The variables of $\text{exp}$ and $\text{exp}'$ have been assigned a type in $\Gamma'$. The variables of $\text{exp}$ and $\text{exp}'$ are also the domain of $\gamma$, which assign an expression to these variables. As $(\#)$ and $(\star)$ have been proved with those variables universally quantified, any expression that we substitute to those we still have $(\#)$ and $(\star)$ provable. Therefore, we have $(\#)$ Type System $\cup \text{Lemmas} \cup \Gamma' \vdash \text{exp} : T$ and that $(\star)$ Type System $\cup \text{Lemmas} \cup \Gamma' \vdash \text{exp}' = \text{exp}' : T$. As $e = \text{exp} y$ and $e' = \text{exp}' y$, then the LHS and RHS have the same type.

Contextual rules and Error context rules follow standard reasoning (recall that errors are always typed at any type).

A.3 Type Soundness

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

B Inverting Typing Rules for Type Preservation

\begin{align*}
\text{INVERT-ONE} & : \quad \text{Type System} \cup \text{Lemmas} \cup \Gamma' \vdash (\text{op} \overrightarrow{\text{args}}) : ty \\
& \quad \text{Type System} \vdash_{\text{lhs}} (\text{op args}) : ty \Rightarrow \Gamma' \\
\text{INVERT-TWO} & : \quad \text{Type System} \cup \text{Lemmas} \cup \Gamma' \vdash (\text{op} \overrightarrow{\text{args}}) : ty \\
& \quad \text{Type System} \cup \text{Lemmas} \cup \Gamma' \vdash (\text{op} \overrightarrow{\text{args}}) : ty \\
\text{RHS-CHECK} & : \quad \text{Type System} \cup \text{Lemmas} \cup \Gamma' \vdash \text{exp} : ty \\
& \quad \text{Type System} \cup \Gamma' \vdash_{\text{rhs}} \text{exp} : ty \\
\text{SELF} & : \quad \text{Type System} \cup \Gamma' \vdash \text{exp} : ty \\
& \quad \text{if } r = \text{ps} \quad \text{ps'} \\
& \quad \Gamma \vdash (\text{op args}) \in \text{Type System} \\
& \quad \frac{\text{ps'}}{\Gamma \vdash (\text{op args})} = r \text{ instantiated with (op args)}
\end{align*}