Extrinsically Typed Operational Semantics for Functional Languages

Matteo Cimini
University of Massachusetts Lowell
USA
matteo_cimini@uml.edu

Dale Miller
Inria & LIX, École Polytechnique
France
dale.miller@inria.fr

Jeremy G. Siek
Indiana University
USA
jsiek@indiana.edu

Abstract
We present a type system over language definitions that classifies parts of the operational semantics of a language in input, and models a common language design organization. The resulting typing discipline guarantees that the language at hand is automatically type sound.

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.

CCS Concepts:
• Software and its engineering → Formal language definitions; • Theory of computation → Type theory.

Keywords: Extrinsic type systems, type soundness, functional languages

ACM Reference Format:

1 Introduction
Type systems play a major role in program analysis. Can they play a similar role in language analysis, i.e. the analysis of the meta-theoretic properties of programming languages?

The research line on intrinsic typing has long investigated this question, and 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. 2017; 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 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 perform 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 program1. Our main contribution is a meta type system that ensures that the various parts 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

1The word extrinsic can be confusing when we dive in the technical part of the paper, as our meta type system needs to analyze a type system which is itself extrinsic. We therefore say meta type system.
hand in order to apply the common classification of operators as introduction forms (such as $\lambda x : T. e$ 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, funType is the function type, and Section 3.2 presents complete details.)

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

This is a meta-level typing rule that type checks a reduction rule. Here, the first line of premises says that the type of the operator app is “elim funType”, which means 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, which we check by making sure that the abstraction is of type “value funType”. This materializes a common design in programming languages in which elimination forms manipulate values of some data type. As exemplified above, it is typical to apply this approach by assigning a principal argument to an elimination form of a type and then by defining its reduction rules by case analysis on the values of that type.

The meta-level typing rule above also checks that the arguments at positions 1 and 2 must be evaluation contexts for the application, as those arguments need to be evaluated for the $\beta$-rule to apply. Contexts $\vdash \text{app : \{1, 2\}}$ enforces that the definition of evaluation contexts for application have $E$ in both the first and second sub-expression positions, as in the grammar: $E ::= (\text{app } E E) | (\text{app } v E) | \cdots$

Analogously, if the language at hand had lists and the reduction rule head (cons $v_1 v_2$) $\rightarrow v_1$, 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 $v_1 v_2$) 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 $\Gamma \vdash e : T$ and a reduction relation of the form $e \rightarrow e$ 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 the LANG-N-CHECK tool [Cimini 2015]. 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, if we forgot the context (if C e e) at line 7, 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 and checked their type soundness, including the simply typed $\lambda$-calculus and its variants with integers, booleans, pairs, lists, sums, tuples, fix, let, letrec, unit, universal types, recursive types, option types, exceptions, list

---

**Figure 1.** Example input of LANG-N-CHECK: language definition of System $F$ with booleans. We do not mention binding variable in lines such as 21 and 22 because x is the bound expression variable by default in the tool, and X is the type variable by default.

| 1 | Expression $E ::= x | (\text{abs } T (x)E) | (\text{absT } X(E)) | (\text{app } E E) |
| 2 | $\text{absT } X(E) | (\text{app } E T) | (tt) | (ff) | (if F E E) |
| 3 | Type $T ::= \text{bool} | (arrow T T) | (all X T) |
| 4 | Value $V ::= \text{abs } T (x)E | (\text{absT } X(E)) | (tt) | (ff) |
| 5 | Error ::= |
| 6 | Context $C ::= \{} | (\text{app } C e) | (\text{app } v C) | (\text{appT } C T) |
| 7 | $\text{|if C e e|} |
| 8 | 9 | Gamma $\vdash (\text{abs } T E) : (\text{arrow } T T) ::= |
| 10 | 11 | Gamma, x : T $\vdash E : T. |
| 12 | 13 | Gamma $\vdash (\text{app } E T) : T \equiv Gamma, X \vdash E : T. |
| 14 | 15 | Gamma $\vdash (\text{appT } T E T) : T T (T1/X) \equiv Gamma, X \vdash E : (\text{all } T T). |
| 16 | 17 | Gamma $\vdash (\text{ffx } X) : (\text{bool}). |
| 18 | 19 | Gamma $\vdash (\text{if } E E E) : T \equiv Gamma, X \vdash E : (\text{bool}) |
| 20 | 21 | Gamma $\vdash E : T |
| 22 | 23 | Gamma $\vdash (\text{iff } E E) : T |
| 24 | 25 | (\text{app } (\text{abs } T E) V) \rightarrow E[v/x] |
| 26 | 27 | (\text{appT } E T) \rightarrow E[v/x] |
| 28 | 29 | (\text{if } (tt) E E) \rightarrow E |
| 30 | 31 | (\text{if } (ff) E E) \rightarrow E |

---

SLE '20, November 16–17, 2020, Virtual, USA

Matteo Cimini, Dale Miller, and Jeremy G. Siek
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 recursor 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.

Closely 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 languages stand as compared to intrinsic typing?

A recent result by Poulsen et al has demonstrated that intrinsic typing can be applied to practical languages, and in particular to a large subset of Middleware Java [Poulsen et al. 2017]. 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 2017.

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.

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 Fexc, which essentially is 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 results of successful computations. Values are defined with the syntactic category Value. Similarly, language designers designate which expression constitute the error, which is the outcome when computations fail. The error is defined with the syntactic category Error. Context defines the evaluation contexts, which prescribe within what contexts reduction can 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, Fexc defines its type system with a relation of the form Σ ⊢ e : T, which is inductively defined with a set of inference rules. The dynamic semantics here is of the form e → 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. (→∗ is the reflexive and transitive closure of →).

Definition 2.1 (Type Soundness). A language L is type sound whenever for all expressions e, e′, and types T, if Σ ⊢ e : T and e →∗ e′ then either:

- e′ is a value such that Σ ⊢ e′ : T,
- e′ is an error, or
- there exists e″ such that e → e″ and Σ ⊢ e″ : T.
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 $\text{if } e \text{ then } e \text{ else } e$. We use $x$s as variables for expressions and (capitalized) $X$s 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 expressions with a bound $x$, followed by expressions with a bound $X$, and finally followed by (not bound) expression arguments. This leads to a general shape that is similar to Harper’s abstract binding trees [Harper 2012] and is $\text{	exttt{op}} \Gamma \rightarrow (X.T) (x.e) (X.e) \tau$, where $\text{op}$ is an operator, and $\tau$ denotes sequences. To make an example, abstraction fits that general shape as $(\text{abs } T (x).e)$ and a type-annotated $\text{cons}$ fits that shape as $(\text{cons } T e e)$. 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 $\mathcal{E}_\text{exc}$.

Figure 3 contains the main judgement for $\vdash \mathcal{L}$, handled by rule $[\text{MAIN}]$. 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 in Figure 3, as well. 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, $\text{Val}$ enforces that if $(\text{cons } v u)$ is declared as value then Context must contain productions of the form $(\text{cons } E _)$. Indeed, if one of these were missing, say the first, the expression $(\text{cons } (\lambda x.x) n_1 n_1)$ 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 $(\text{cons } E u)$ and $(\text{cons } v E)$. Such declarations make the expression $(\text{cons } ((\lambda x.x) n_1) ((\lambda x.x) n_1))$ 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, $(\text{err-ctx})$ should not apply to derive the reduction $\text{try } (\text{raise } v) \text{ with } e \rightarrow \text{raise } v$, as we expect the semantics of $\text{try}$ to handle the error.
**Type Checking** Type. The judgement $\vdash T$ in Fig. 3 checks some restrictions on the grammar for types. In particular, we allow types to be a constant $c$ applied to $Ts$, or applied to $Ts$ with bound variables $xs$ (for types quantified by types). On one hand, this accommodates universal types and recursive types but also prevents expressions from being used in types, ruling out dependent types and refinement types. These latter features are rather advanced and we leave them for future work.

**Type Checking** Expression. We use $exp$ to denote an expression derived by the grammar for Expression. The judgement $\vdash exp$ 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 is that we check the language design invariant $\text{Val}$, defined above. This check is done with Context $\vdash op : contextPositions$, which informs about the set of argument positions that are declared as evaluation contexts for $op$. For example, if Context has declarations $E ::= (\text{cons } E e)$, 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 $v ::= (\text{cons } v v)$ 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 Context $\vdash e$ checks the error declaration, if present. The invariant $\text{Val}$ is checked here, as well. For example, $\text{raise } E$ must be an evaluation context to prevent expressions such as $\text{raise } (\lambda x : T.e) \text{ nil}$ from being stuck and thereby jeopardizing 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, $(\text{cons } E v)$ is a valid $ctx$. The judgement Context $\vdash Context$ checks that each context declaration has exactly one occurrence of $E$. That is, $(\text{cons } E v)$ is not a valid $ctx$. (We enforce this with the existential quantification with uniqueness $\exists !$.) It also enforces language design invariants $\text{Val}$ and $\text{Ctx}$. The latter is ensured with the use of the function $\text{acyclic}$ 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 $(\text{cons } E e)$ and $(\text{cons } v E)$, i.e., left-to-right evaluation, induce the graph $[2 \rightarrow 1]$, which is acyclic. The bad context declarations $(\text{cons } E v)$ and $(\text{cons } v E)$, instead, induce the graph $[1 \rightarrow 2, 2 \rightarrow 1]$, which contains a cycle and is rejected. The function $\text{acyclic}$ 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 $\text{ErrCtx}$ 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) $op \rightarrow R$, where $op$ is an operator and $R$ is the role that this operator plays in the language, 2) $exp \rightarrow value c$ that denotes that $exp$ is a value for the type constructor $c$, or 3) $exp \rightarrow error$, 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 $F$.

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

We say that $\text{let } x = e_1 \text{ in } e_2 : T_2$ 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-абст)). 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 \((c \overrightarrow{T})\), as highlighted, that is, a type constructor applied to arguments. \([\overline{T-VALUE}]\) also checks that \(op\) forms a value. To make some examples, below \([\overline{T-VALUE}]\) is instantiated to check \((\Lambda x.e)\) and the typing rule for the list operator \(\text{cons}\).

\[
\begin{align*}
\Gamma & \vdash \lambda x : T.e \\
\text{Error} & \quad \text{Dynamic Sem.}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash \lambda x : T_1.e : T_2 \\
\text{Dynamic Sem.} & \quad \text{Error}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash \text{cons} \ e_1 \ e_2 : \text{List} \\
\text{Error} & \quad \text{Dyn. Sem.}
\end{align*}
\]

\text{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 \(v\). \(E\).
Type Environment \( \Gamma^* := \Gamma \mid \Gamma.x : T \mid \Gamma.X \)

Role Maps \( B := op \mapsto R \mid exp \mapsto value \ c \mid exp \mapsto error \)

Roles \( R := \text{elim} \ c \mid \text{derived} \mid \text{errorHandler} \)

**Value | Error | Dynamic Semantics \( \vdash \) Type System : Classification**

![Diagram](image)

**Figure 4.** Meta Type system: Typing Rules. For simplicity, each \( e_i \) in \( e_1, \ldots, e_n \) may be an expression variable or a bound expression such as \((x)_e\). The sequences \( T \) contain types, each of which may also be a bound type such as \((X).T\). Sequences \( \overline{args} \) contain variables, which may be under a binder as well. Also, sequences in this figure have distinct variables.

operator is defined by case analysis on some values. Therefore, \([\text{-elim}]\) expects to find at least a reduction rule of the form \((op_1 \ (op_2 \ \overline{args}_2) \ \overline{args}_1) \rightarrow exp^+ \in \text{Dynamic Semantics} \quad op_2 \in \text{Values}\)

Values | Error | Dynamic Semantics \( \vdash \)

\[
\Gamma \vdash e_1 : ty_1 \quad \ldots \quad \Gamma \vdash e_n : ty_n \\
\Gamma \vdash \langle \overline{arg} \rangle : \langle \\cdot \overline{T} \rangle : \langle \text{value} \rangle \\
\Gamma \vdash \langle \text{elim} \rangle \langle \overline{arg} \rangle : \langle \text{value} \rangle \\
\Gamma \vdash \langle \text{errorHandler} \rangle \langle \overline{arg} \rangle : \langle \text{value} \rangle \\
\Gamma \vdash \langle \text{derived} \rangle \langle \overline{arg} \rangle : \langle \text{value} \rangle \\
\Gamma \vdash \langle \text{error} \rangle \langle \overline{arg} \rangle : \langle \text{value} \rangle
\]

\( \forall (op \ \overline{term}) \rightarrow exp^+ \in \text{Dynamic Semantics}, \ \overline{term} \) contains only variables

Values | Error | Dynamic Semantics \( \vdash \)

\[
\Gamma \vdash e_1 : ty_1 \quad \ldots \quad \Gamma \vdash e_n : ty_n \\
\Gamma \vdash \langle \overline{arg} \rangle : \langle \text{value} \rangle \\
\Gamma \vdash \langle \text{error} \rangle \langle \overline{arg} \rangle : \langle \text{value} \rangle
\]

expression \( exp^+ \), which denotes an expression \( exp \) in which substitutions of the form \( exp[exp/x] \) and \( exp[ty/x] \) can occur.

Some of the reduction rules of error handlers have the same pattern of that for elimination forms, but an error appears as principal argument. Therefore, 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), and the typing rule for head.

\[
\text{app } (\lambda x : T. e) \, v \rightarrow e[v/x] \in \text{Dynamic Semantics}
\]

\[
\begin{array}{c|c|c}
\text{Value} & \Gamma \vdash e_1 : T_1 & \Gamma \vdash e_2 : T_2 \\Rightarrow \quad \text{app} \\
\hline
\text{Value} & \Gamma \vdash e_1 : T_1 \rightarrow T_2 & \Gamma \vdash e_2 : T_2 \\Rightarrow \quad \text{elim} \\
\hline
\text{Dynamic Sem.} & \Gamma \vdash \text{app } e_1 : e_2 : T_2 & \Rightarrow \quad \text{t-app}
\end{array}
\]

\[
\text{head } (\text{cons } v_1 \, v_2) \rightarrow v_1 \in \text{Dynamic Semantics}
\]

\[
\begin{array}{c|c|c}
\text{Value} & \Gamma \vdash e : \text{List } T \Rightarrow \quad \text{head} \\
\hline
\text{Error} & \Gamma \vdash e : T \Rightarrow \quad \text{elim} \\
\hline
\text{Dynamic Sem.} & \Gamma \vdash \text{head } e : T \Rightarrow \quad \text{t-elim}
\end{array}
\]

Imposing that the principal argument be a value and typed at a complex type helps satisfy the progress theorem for the language being type checked. In particular, when we analyze an operator whose typing rule has been type checked with [T-ELIM] we are guaranteed to be able to apply the appropriate canonical form lemma. The meta type system for the dynamic semantics of Section 3.3 then ensures the existence of reduction rules to prove some steps, and therefore that the progress theorem would hold for the operator.

[T-ERRHANDLER] classifies the error handlers. In doing so, we make sure that error handlers handle the error at their principal argument. [T-ERRHANDLER] is similar to [T-ELIM]. There are two differences: it does not impose the principal argument to be typed at some precise type, and it expects a reduction rule of the form \((\text{op}_1 \, (\text{op}_2 \, \text{args}) \, \text{args}) \rightarrow \text{exp}^*)\), where the operator \text{op}_2 is an error rather than a value.

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

\[
\text{try } (\text{raise } v) \, w \rightarrow e(v) \in \text{Dynamic Semantics}
\]

\[
\begin{array}{c|c|c}
\text{Value} & \Gamma \vdash e_1 : T \Rightarrow \quad \text{try} \\
\hline
\text{Error} & \Gamma \vdash e_2 : T \Rightarrow T \Rightarrow \quad \text{errorHandler} \\
\hline
\text{Dynamic Sem.} & \Gamma \vdash \text{try } e_1 : e_2 : T \Rightarrow \quad \text{t-errHandler}
\end{array}
\]

[T-DERIVED] 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 \((\text{op} \, \text{args}) \rightarrow \text{exp}^*)\), where \text{args} are exclusively variables. Therefore, \text{op} does not pattern-match its arguments into complex expressions. To make some examples, we instantiate [T-DERIVED] for the typing rules of \text{fix} and \text{letrec}.

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

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

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

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

[T-ERROR] handles the typing rule for the error. We enforce that the assigned type is a fresh variable \(T\). This ensures that the error can be typed at any type. This is necessary to establish the type preservation of the language being type checked. Indeed, the error travels through contexts thanks to the rule (err-CTX), and it must be prepared to match the type of the expression it replaces. It is easy to see that [T-ERROR] checks (t-error) successfully.

For \(F_{\text{exec}}\), all typing rules check successfully and we end up with the classification below.

\[
\lambda x : T. e \rightarrow \text{value} \\
\Lambda x e \rightarrow \text{value} \\
\text{app} \rightarrow \text{elim} \\
\text{tapp} \rightarrow \text{elim} \\
\text{try} \rightarrow \text{errorHandler}, \\
\text{raise } v \rightarrow \text{error}
\]

We write Classification \(\vdash \text{app} : \text{elim} \rightarrow \) for retrieving information about operators.

**Back to Error Context.** After having meta type checked the typing rules, we can check Error Context because we have acquired knowledge on the error handler, if present. Indeed, we need to ensure \text{ErrCtx}, and we do so with the typing judgement Classification \(\vdash \text{Error Context}\), defined below.

\[
\begin{array}{c|c|c}
\text{Classification} & \text{Context} & \vdash \text{Error Context} \\
\hline
\text{Classification} & \text{Context} & \vdash \, \text{err} : \equiv e \\
\hline
\text{Classification} & \text{Context} & \vdash \text{error} \\
\hline
\text{Classification} & \text{Context} & \vdash \text{errorHandler}
\end{array}
\]

If the language does not contain an error, Error Context is the empty derivation \(\varepsilon\), which we consider well-typed by default. If an error is present but there are no error handlers, then we simply impose that error contexts are exactly the evaluation contexts. We recall that evaluation contexts are defined with the variable \(E\) and error contexts are defined with the variable \(F\). With the notation \text{Contexts}[F/E] we denote the grammar of Context in which variables \(E\) are replaced with \(F\). Ultimately, if an operator is classified as an error handler, we check that error contexts are evaluation contexts minus the error handler at the principal argument \(F\) as the first argument of \text{op}_2).
Type checking error contexts make them synch with evaluation contexts, and helps proving the progress theorem for the language being type checked because it ensures that operators have a defined behavior if an error appears as one of their arguments.

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:

- **ValRed**: 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. For example, without the context declaration \((\varepsilon e)\) the expression \(((\lambda x. x) (\lambda x. x)) 111\) would be stuck because (beta) would not fire and jeopardize type soundness. Similarly, since (beta) requires that also the second argument of app be a value, missing the evaluation context would jeopardize type soundness.

- **AllVal**: Each elimination form of some type manipulates all the values of that type. For example, were we to miss to specify the reduction rule for the predecessor operation \((\text{pred} \space \text{zero}) \rightarrow \text{error}\), the expression \((\text{pred} \space \text{zero})\) would be stuck, jeopardizing type soundness.

- **ErrSucc**: Error handlers must handle both the error and the case of success. For example, if we were to define \((\text{r-try-error})\) as the sole behavior of the error handler, the expression \(\text{try} \space \text{true}\) with \(e\) would be stuck, jeopardizing type soundness. A reduction rule such as \((\text{r-try-success})\), which expects a value at principal argument position, must be specified.

- **Preserv**: Naturally, reductions must be type preserving, i.e., in a reduction \(e \rightarrow e'\), \(e\) and \(e'\) must have the same type.

[r-main] is the main typing rule. First, it imposes that (ctx) and (err-ctx) are present. Next, it makes sure that the rest of the reduction rules \(\phi_1, \ldots, \phi_n\) are well-typed.

Meta type checking a reduction rule returns a move map (C), which is a description of the computational step that the reduction rule enables. A move map assigns a move \((M)\) to the operator that is the subject of the reduction rule. Moves can be:

- eliminates \(op\): for reduction rules of elimination forms, which manipulate some specific value.
- handlesError: for the reduction rule that handles the error.
- handlesSuccess: for the error handler to handle the case when an error does not occur.
- moves: for reduction rules of derived operators.

When all the reduction rules have been checked, [r-main] collects all the move maps and we perform an exhaustiveness check to ensure that AllVal and ErrSucc are fulfilled. This is done with Moves exhausts Classification.

[r-elimination] handles the case of reduction rules for elimination forms. The first aspect that [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, [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. These checks help satisfy the progress theorem of the language being type checked. If some arguments need to become a value for the rule to fire, the existence of the corresponding evaluation contexts let them evaluate indeed. Also, [r-elimination] ensures that when the principal argument becomes the value \(op_2\) then we can take a step with this reduction rule, satisfying the progress theorem.

Lastly, [r-elimination] and all other 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).

\[
\begin{align*}
\text{Classification} & 
\vdash \text{app} : \text{elim} \rightarrow \\
\text{Classification} & 
\vdash \lambda x : T.e : \text{value} \rightarrow \\
\text{Context} & 
\vdash \text{app} : \{1, 2\}
\end{align*}
\]

\[
\begin{align*}
\text{Context} & \vdash (\lambda x : T.e) \vdash e[v/x] : \text{T} \\
\text{Type System} & \quad \text{eliminates } \lambda
\end{align*}
\]

\[
\begin{align*}
\text{Classification} & \vdash \text{head} : \text{elim} \text{List} \\
\text{Classification} & \vdash \text{cons} v_1 v_2 : \text{value} \text{List} \\
\text{Context} & \vdash \text{head} : \{1\}
\end{align*}
\]

\[
\begin{align*}
\text{Context} & \vdash (\text{cons} v_1 v_2) \vdash v_1 : \text{List} \\
\text{Type System} & \quad \text{eliminates cons}
\end{align*}
\]

Notice that a call-by-name application with reduction rule \((\text{app} (\lambda x : T.e_1)) e_2 \rightarrow e_1[e_2/x]\) would be handled similarly, with the difference that the check on evaluation contexts would be Context \vdash \text{app} : \{1\}. This is because, roughly speaking, the second argument of the application is an \(e\) rather than a \(v\).

[r-errHandler] addresses the reduction rule for error handling. This rule is similar to [r-elimination]. We first detect that the operator at hand is an error handler, and then we check that the principal argument forms an error. To make an example, [r-errHandler] is instantiated for
Move Maps $C ::= op \mapsto M$

Moves $M ::= \text{eliminates } op \mid \text{handlesError} \mid \text{handlesSuccess} \mid \text{moves}$

Moves exhausts Classification whenever $[op_1 : \text{elim c}, \ op_2 : \text{intro c}] \subseteq \text{Classification}$ implies $op_1 : \text{eliminates } op_2 \in \text{Moves}$, and $[op_1 : \text{errorHandler}, \ op_2 : \text{error}] \subseteq \text{Classification}$ implies $\{op_1 : \text{handlesError}, op_1 : \text{handlesSuccess}\} \subseteq \text{Moves}$.

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

Context $\vdash \text{Classification}$ $\vdash \text{Type System}$ $\vdash \text{Dynamic Semantics}$

$[\text{r-main}]$

\[
\begin{align*}
\text{Type System} & \vdash_{\text{lhs}} (op_1 (op_2 \ arg{\bar{s}}) \ \overline{\arg{\bar{s}}}) : ty \Rightarrow \Gamma^s & \text{Type System } \vdash \Gamma^s \vdash_{\text{exp}^+} : ty \\
\end{align*}
\]

$[\text{r-elimination}]$

\[
\begin{align*}
\text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op_1 : elim c} & \text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op_1 : contextPositions} \\
\text{Classification} & \vdash (op_2 \ arg{s}) : \text{value c} & \forall i \in [1] \cup \{i \mid \ arg{s}_i = v\}, i \in \text{contextPositions} \\
\text{Type System} & \vdash_{\text{lhs}} (op_1 (op_2 \ arg{s}) \ \overline{\arg{s}}) : ty \Rightarrow \Gamma^s & \text{Type System } \vdash \Gamma^s \vdash_{\text{exp}^+} : ty \\
\end{align*}
\]

$[\text{r-errorHandler}]$

\[
\begin{align*}
\text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op_1 : errorHandler} & \text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op_1 : contextPositions} \\
\text{Classification} & \vdash (op_2 \ arg{s}) : \text{error} & \forall i \in [1] \cup \{i \mid \ arg{s}_i = v\}, i \in \text{contextPositions} \\
\text{Type System} & \vdash_{\text{lhs}} (op_1 (op_2 \ arg{s}) \ \overline{\arg{s}}) : ty \Rightarrow \Gamma^s & \text{Type System } \vdash \Gamma^s \vdash_{\text{exp}^+} : ty \\
\end{align*}
\]

$[\text{r-success}]$

\[
\begin{align*}
\text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op_1 : errorHandler} & \text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op : contextPositions} \\
\text{Type System} & \vdash_{\text{lhs}} (op \ u \ \overline{\arg{s}}) : ty \Rightarrow \Gamma^s & \text{Type System } \vdash \Gamma^s \vdash_{\text{exp}^+} : ty \\
\end{align*}
\]

$[\text{r-moves}]$

\[
\begin{align*}
\text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op : derived} & \text{Context} & \vdash \text{Classification} \vdash \text{Type System} \vdash \text{op : contextPositions} \\
\text{Type System} & \vdash_{\text{lhs}} (op \ \overline{\arg{s}}) : ty \Rightarrow \Gamma^s & \text{Type System } \vdash \Gamma^s \vdash_{\text{exp}^+} : ty \\
\end{align*}
\]

$[\text{r-try-error}]$ in the following way.

\[
\begin{align*}
\text{Classification} \vdash \text{try : errorHandler} \\
\text{Classification} \vdash \text{raise v : error} \\
\text{try : } \{1\}
\end{align*}
\]

$\text{try \ raise \ v} \mapsto (e \ v) \mapsto \text{handlesError}$

$[\text{r-success}]$ addresses the reduction rule that specifies the behavior of the error handler when an error does 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, $[\text{r-success}]$ checks $[\text{r-try-success}]$ in

\[
\begin{align*}
\vdash \text{Classification} \vdash \text{Type System} \vdash \text{try : errorHandler} \\
\vdash \text{Classification} \vdash \text{Type System} \vdash \text{try : raise v} \\
\vdash \text{try : with e handlesError}
\end{align*}
\]
the following way.

\[
\begin{array}{c}
\text{Classification} \vdash \text{try} : \text{errorHandler} \\
\text{Context} \vdash \text{try} : [1] \\
\end{array}
\]

\[
\begin{array}{c}
\text{Context} \vdash \text{try} \\
| \text{Classification} \vdash \text{try} \nu \text{ with } e \rightarrow v : \leftrightarrow \rightarrow \rightarrow \\
| \text{Type System} \vdash \text{handlesSuccess} \\
\end{array}
\]

\[\text{[R-moves]} \] handles the reduction rule of derived operators. To make an example, the reduction rules for \text{fix} and \text{let} are handled as follows. (Evaluation contexts are \text{fix} \ E \mid \text{let} x = E \ in \ e.)

\[
\begin{array}{c}
\text{Classification} \vdash \text{fix} : \text{derived} \\
\text{Context} \vdash \text{fix} : [1] \\
\end{array}
\]

\[
\begin{array}{c}
\text{Context} \vdash \text{fix} \nu \rightarrow v(\text{fix} \nu) : \text{fix} \rightarrow \text{moves} \\
| \text{Type System} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Classification} \vdash \text{let} : \text{derived} \\
\text{Context} \vdash \text{let} : [1] \\
\end{array}
\]

\[
\begin{array}{c}
\text{Context} \vdash \text{let} x = v \ in \ e \rightarrow e[v/x] : \text{let} \rightarrow \text{moves} \\
| \text{Type System} \\
\end{array}
\]

### 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 \( \text{exp} \rightarrow \text{exp}' \), we have to ensure that the types of \( \text{exp} \) and \( \text{exp}' \) 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 and 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 \( \Gamma, \Gamma \vdash \text{exp} : T \) implies \( \Gamma \vdash \text{exp}': T \), 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 \( \Gamma_{\text{hs}} \vdash \text{exp} : t y \Rightarrow \Gamma' \), which means that according to the type system we have that \( \text{exp} \) has type \( t y \) under assumptions \( \Gamma' \), which are computed as output. The way we build \( \Gamma' \) for \( \text{exp} \) is by inverting the typing rules used to type check \( \text{exp} \). 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 \( (\text{cons} v_1 v_2) \rightarrow v_1 \) gives \( \Gamma' = \Gamma \vdash v_1 : T, \Gamma \vdash v_2 : \text{List } T \). We also obtain its type \( T \). Then, we check Type System \( | \Gamma' \vdash_{\text{hs}} \text{exp}' : t y \), which means that \( \text{exp}' \) has type \( t y \) according to the type system and \( \Gamma' \), which is an input this time. This check solves a provability problem:

\[
\begin{array}{c}
\text{Type System} \cup \text{Lemmas} \cup \Gamma \vdash v_1 : T, \Gamma \vdash v_2 : \text{List } T \Rightarrow \Gamma \vdash v_1 : T. \\
\end{array}
\]

That is, the formula \( \Gamma \vdash v_1 : T \) can be proved with the inference system formed by the type system, substitution lemmas (Lemmas, see below), and the formulae in \( \Gamma' \).

Lemmas contains the well-known substitution lemmas:

\[
\begin{array}{c}
(\text{typ-sub1}) \\
\Gamma, x : T_1 \vdash e : T_2 \Gamma \vdash e' : T_1 \Gamma, X \vdash e : T \\
\Gamma \vdash e[e'/x] : T_2 \quad \Gamma \vdash e[T'/X] : T'[T'/X] \\
\end{array}
\]

We have proved that these lemmas automatically hold for any given language that is restricted to our setting.

### 4 Correctness of the Meta Type System

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 [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 a tool called lang-n-check [Cimini 2015]. 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 [Baeld 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 recur sor 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.

Some remarks on the type preservation checks and the generation of the machine-checked proof are in order. These are two different processes. During type checking, we use Abella to run queries that check for type preservation. After type checking, LANG-N-CHECK generates the proof code that describes the proof of type preservation. This code contains instructions to do case analysis on hypothesis at the correct order, apply the inductive hypothesis, and so on. If the mentioned queries succeed during type checking, then the proof code succeeds, too, as the proof is guaranteed to find what it needs during its execution.

Informative Error Messages. LANG-N-CHECK has been used to teach programming languages theory [Cimini 2019]. That paper remarks that LANG-N-CHECK can report informative error messages to the user.

We provided an example in the introduction section. To show other examples³, consider the listing in Figure 1 of the introduction.

- Were we to miss one of the reduction rules for the if operator, say the rule at line 22, LANG-N-CHECK would reject the language definition and 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”.
- Were we to miss the context declaration (app v C) 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. There are type sound languages whose type soundness cannot be verified with our system. 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. We do not have a general treatment for subtyping and type-based access, therefore calculi such as Featherweight Java [Igarashi et al. 2001] are currently out of the scope of LANG-N-CHECK. 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 some times 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. 2017].

6 Related 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. 2017]. 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 theirs⁴. 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. 2017].

2) We work with operational semantics formulations rather than definitional 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. The syntax of LANG-N-CHECK is indeed inspired by that of the much earlier work of Ott [Sewell et al. 2007]. Cimini [Cimini 2019] observes that students

³Similar examples are made in Cimini 2019.

⁴For ease of reference, the title is "Intrinsically Typed Definitional Interpreters for Imperative Languages".
could use **lang-n-check** after having been taught type systems and operational semantics 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. This is further exemplified by the existence of work on linear and session types with intrinsic typing [Rouvoet et al. 2020], which also are out of our scope.

**Automated Theorem Proving.** The seminal work of Schürmann and Pfenning 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 2000]. 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 checks 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.

Marino and Millstein propose a generic type-and-effect system [Marino and Millstein 2009] that language designers can instantiate with effects. The resulting languages are guaranteed to be type sound. However, the underlying language of the framework is fixed and only the effects of the language are parametrized. Therefore, adding new operators does not come with a guarantee of type soundness. In contrast, our work allows language designers to define the operations of languages and check for type soundness. In the future, we plan on integrating the approach of Marino and Millstein into **lang-n-check** and add a now missing treatment for effects.

**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], Lemi [Mulligan et al. 2014], the K framework [Roșu and Șerbănuță 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; Efttinge and Spönenmann 2006], and Sugar [Erdweg et al. 2011], to name a few. However, these systems do not establish the type soundness of the languages being defined.

A related work in the context of tools for teaching is SA-SyLF [Aldrich et al. 2008], a proof assistant that has been used in the classroom. Students can write derivations and proofs in readable proof code and SA-SyLF provides feedback on students’ mistakes. Differently from SA-SyLF, students do not write proofs in **lang-n-check**, they receive feedback directly on the language definitions they provide.

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. 2017; Rouvoet et al. 2020]. 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.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1763922.

---

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 Correctness of the Meta Type System

We follow the syntactic approach to type soundness of Wright and Felleisen [Wright and Felleisen 1994]. We generate canonical form lemmas, progress, type preservation and, ultimately, type soundness theorems and proofs.

A.1 Progress Theorem

Progress Theorem:

If $\Gamma \vdash L$ then

if $\emptyset \vdash e : T$ then $e$ 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 $\Gamma \vdash L$ 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 for all $f_i$, and $(\text{conclusion})\gamma = \vdash e : T$.

For each case of the induction on $\vdash e : T$, the proof goes as follows. Since $\vdash L$ and $\phi$ is a typing rule of $L$ then $\phi$ has been type checked. Therefore, $e$ has a top level operator $\text{op}$ and arguments, as this is the shape imposed for typing rules. It also means that all the arguments of the operator, say $\overline{e}$, are 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 $\overline{e}$. In particular, we apply the inductive hypothesis for those arguments $e_i$ with index $i \in \text{contextPosition}$ from Context $\vdash \text{op} : \text{contextPosition}$, 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 $\Gamma \vdash e_i : t_i\gamma$ (and not with $\Gamma, x : T$). Therefore we can apply the inductive hypothesis to $\vdash e_i\gamma : t_i\gamma$ for all of these, and we derive that $e_i\gamma$ progresses. We call the Progress Lemma for $\text{op}$ (defined below), which expects exactly those progress assumptions $e_i\gamma$ progresses. The Progress Lemma for $\text{op}$ satisfies the Main Progress Theorem, as discussed below.

Progress Lemma for $\text{op}$. Fix an operator $\text{op}$ in $L$, and let Context $\vdash \text{op} : \{1, \ldots, n\} \in \text{Context}$. Without loss of generality, here contextual arguments appear first as a presentation aid. The progress theorem for $\text{op}$ is: (We mark one of the assumption with (H)).

If $\vdash L$ and (H)$\vdash (\text{op } e_1 \ldots e_n \overline{e}) : T$, and progress $e_1, \ldots, progress\ e_n$ then progress (\text{op } e_1 \ldots e_n \overline{e})

The proof is by case analysis on (H). If $\vdash (\text{op } e_1 \ldots e_n \overline{e}) : T$ has been proved, then it has been proved with a typing rule $\phi$. We do case analysis on all progress $e_1, \ldots, progress\ e_n$, but in a suitable order. Since $\vdash L$, we have that acyclic(c(Context)). Therefore we can choose an order for argument positions for which the following invariant (invariant) holds: we do case analysis on progress $e_i$ before the case analysis on progress $e_j$ if the context for $i$-th argument of $\text{op}$ does not depend on the valuehood of the argument $j$ of $\text{op}$. 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.

Example: If we have two arguments, the first case analysis on progress $e_1$ opens three cases: 1) value $e_1 \land progress\ e_2$, 2) step $e_1 \land progress\ e_2$, 3) error $e_1 \land progress\ e_2$, where step $e$ means that $e$ can take a step, and error $e$ means that $e$ is an error. After we open this case analysis we are at the left child 1). 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 $e_1 \land progress\ e_2$, 2) value $e_1 \land step\ e_2$, 3) value $e_1 \land error\ e_2$. Afterwards, we are still at the leftmost child: value $e_1$ and value $e_2$, thanks to the invariant.

We show only the case for this leftmost child here, as the further cases analysis generated follow a similar reasoning.

The proof is by case analysis on the role that $\text{op}$ has in Classification. This role exists because $\vdash L$, and therefore a typing rule for $\text{op}$ has been type checked, then Classification has been produced.

* $\text{op}$ is (value $e$): As Classification $\vdash \text{op}\ args : value\ c$ then $\text{op}\ args \in Values$. There are three cases. The first is when we strive to apply the value definition for $\text{op}$, however, this definitions may use a $\nu$ variable if some argument is required to be a value for the definition to apply. As we have type checked $\text{op}$, these arguments must be in contextPosition and therefore they all are involved in a progress $e_1, \ldots, progress\ e_n$ assumptions. We are also in the case in which we have done a case analysis on all of them, and we are in the case where all of them are values, and so the definition applies. We have

- **STEP**: case $e_j \rightarrow e'_j$. Moreover, we have that all arguments of $\text{op}$ before $j$ are all values. Therefore we can apply the evaluation context (op $\nu E_j \overline{e}$) (recall that we assume left-to-right evaluation but the argument generalizes for any topological sort on the dependencies of contexts) to prove a step (op $\nu e_j \overline{e}$) $\rightarrow$ (op $\nu e'_j \overline{e}$), and so we progress.

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

* $\text{op}$ is (elim $c$): Then [$\tau\text{-ELIM}$] has meta type checked $\phi$. This means that the $\phi$ has a typing premise for
the principal argument of the form \( \vdash e_1 : (c \overline{\mathcal{T}}) \). Also, we are exploring the case where \( e_1 \) is a value. Therefore, we apply the Canonical Forms Lemma for \( c \) (described in the following paragraph). This means that \( e_1 = t_1 \lor ... \lor t_m \) (this is notation from the next paragraph) with all \( t_k \) be with form \( (op_2 \ args) \) and \( op_2 \in \text{Values} \). Then, since Dynamic Semantics has been meta type checked, it must have produced Moves that passed the exhaustiveness check exhaust. Therefore we have \( op : \text{elim} \ c \) and Classification\( (op_2) = \text{value} \ c \), which enforces that a reduction rule of the form \( r = (op \ (op_2 \ \overline{\mathcal{V}_2}) \overline{\mathcal{V}_1}) \rightarrow \text{exp} \) exists. As we are in the case where the principal argument is a value, and also all other argument in \( \overline{\mathcal{V}_1} \) are values, we have that this reduction rule fires and takes a step, and so we progress.

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

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

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

• \( op \) is (derived): Then \( \phi \) has been meta type checked by \( [\overline{Tlast}] \). Therefore, a rule \( (op \overline{\mathcal{V}_1} e'_1) \rightarrow e_1 \) exists. Also, all those \( e_i \) 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 not pattern-match arguments for the derived operator \( op \), but imposes that it requires the arguments to be \( \forall s \), and we are in the case where those are all values indeed. So the rule fires, and we progress.

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

• \( op \) is (errHandler): Then \( \phi \) has been meta type checked by \( [\overline{\text{ERR}}] \). Then, since Dynamic Semantics has been meta type checked, it must have produced Moves, which by our exhaustiveness check it contains \( op : \text{handleSuccess} \). This means that it exists a reduction rule of the form \( (op \overline{\mathcal{V}}) \rightarrow \text{exp} \). We are in the case where the arguments are values, and so they satisfy that \( \overline{\mathcal{V}} \) in first argument position. So this rule fires, and we progress. Next, we need to prove the cases for step and error. The former is handled with STEP. The latter is handled differently from the previous cases as type checking of Error Context excluded an error context for that position. However, exhaustiveness of Moves imposes that we have \( op : \text{handleError} \), which in turn, implies that there exists a reduction rule that fires when the error is at that position. Therefore, this reduction rule can apply and prove a step, and so we progress.

Canonical Forms Lemma for \( c \). In the theorem below, we have marked an assumption with (H) and another with (V).

Theorem A.1. If \( \vdash \mathcal{L} \) and (H) \( \vdash e : (c \overline{T}) \) and (V) value \( e \) then \( e = \text{OR}((op \ args) | \text{Classification} \vdash (op \ args) : \text{value} \ c) \).

We use OR to denote the disjunction of a set of formulae. value \( e \) means that \( e \) can be derived with the grammar for Value in \( \mathcal{L} \).

The proof is by case analysis on (H). If \( \vdash e : (c \overline{T}) \) has been proved, than 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 \( op \).

We reason by cases analysis on the classification of \( op \). If Classification \( \vdash (op \ args) : \text{value} \ c \) (notice the same \( c \)), then it is one of the values in \( \text{OR}((op \ args) | \text{Classification} \vdash (op \ args) : \text{value} \ c) \), which proves this case. The case Classification \( \vdash (op \ args) : \text{value} \ c' \)) with \( c' \neq c \) cannot happen because \( T \)-value imposes that \( (op \ args) \) is typed at a type \( (c' \overline{T}) \) but we are doing a case analysis on assumption (H) which is \( \vdash e : (c \overline{T}) \), hence the case analysis cannot give a case with \( c' \neq c \).

If the operator \( op \) is such that Classification \( \vdash (op \ args) : \text{elim} \ c \), or derived \( c \), or error \( c \), or errorHandler \( c \), then there is a contradiction with (V), so these cases are discarded, and are proved successfully.

A.2 Type Preservation

**Type Preservation Theorem:**

\[
\begin{align*}
\text{if} \ & \vdash \mathcal{L} \text{ then} \\
\text{for all expressions} & \ e, e' \text{ and types} \ T, \\
\text{if} & \ \emptyset \vdash e : T \text{ and } e \rightarrow e' \text{ then } \emptyset \vdash e' : T
\end{align*}
\]

The proof is by induction on \( e \rightarrow e' \).

Since \( e \rightarrow e' \) is provable, then there exist a reduction rule \( \phi \) with conclusion \( \text{concl} = \text{exp} \rightarrow \text{exp'} \) and a substitution \( \gamma \) such that \( \gamma \text{concl} = e \rightarrow e' \).

We have that \( \phi \) has been type checked, therefore our type checker has classified a move for \( \phi \) such as eliminates \( op \), or some other move. We show the case when the role of \( \phi \) is 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 \( (\text{exp'})\gamma \) is of type \( T \).

As we have type checked \( \phi \) as eliminates \( op \), we know that \( e = (op \ (op_2 \ args) \ args) \), as this is the shape required by [R-Elim]. We have \( \emptyset \vdash e : T \) from the proviso of the theorem,
which must have been proved with a typing rule of the form
\[ \Gamma \vdash (op \ldots) \]
Here, \( ps \) are the premises that contribute to populate \( \Gamma^s \) for the type preservation check. As the typing rule has been type checked, too, its shape must be such that all the arguments that are expressions are recursively type checked. We therefore have a premise in \( ps \) that type checks \((op_2 \ arg_1)\), and this formula has been proved, too. Therefore there is another typing rule that has type checked that formula, and has the form
\[ \Gamma \vdash (op_2 \ldots) \]
where \( ps' \), too, contributes to populate \( \Gamma^s \). Now, since \([R-ELIM]\) has been applied, also its type preservation checks have been satisfied, and we have that \((\#)\) Type System \( \cup \) Lemmas \( \cup \Gamma^s \vdash \exp : T \) and that \((\#')\) Type System \( \cup \) Lemmas \( \cup \Gamma^s \vdash \exp' : T \). The variables of \( \exp \) and \( \exp' \) have been assigned a type in \( \Gamma^s \). The rule meta-variables of \( \exp \) and \( \exp' \) are also in the domain of \( \gamma \) (the substitution that has proved \([R-ELIM]\)). As \((\#)\) and \((\#')\) have been proved with those variables universally quantified, any expression that we substitute to those still makes \((\#)\) and \((\#')\) provable. Therefore, we have \((\#)\) Type System \( \cup \) Lemmas \( \cup \Gamma^s \vdash (\exp) \gamma : T \) and \((\#')\) Type System \( \cup \) Lemmas \( \cup \Gamma^s \vdash (\exp') \gamma : T \). As \( e = (\exp) \gamma \) and \( e' = (\exp') \gamma \), 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

\[ \begin{align*}
\text{Type System}_{\text{premises}}((op \overline{arg_1}) & ) = \Gamma^s \\
\text{Type System} \cup \text{Lemmas} \cup \Gamma^s & \vdash (op_1 \ (op_2 \ arg_1 \ arg_2)) : ty \\
\text{Type System} \vdash_{\text{rhs}} (op \overline{arg_1}) : ty & \Rightarrow \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System}_{\text{premises}}((op_2 \overline{arg_1}) & ) = \Gamma^s \\
\text{Type System} \cup \text{Lemmas} \cup \Gamma^s & \vdash (op_1 \ (op_2 \ arg_1 \ arg_2)) : ty \\
\text{Type System} \vdash_{\text{rhs}} (op_1 \ (op_2 \ arg_1 \ arg_2)) : ty & \Rightarrow \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System}_{\text{premises}}((op \overline{arg_1}) & ) = ps' \\
\text{Type System} \vdash (op \overline{arg_1}) & = \Gamma^s \\
\text{Type System} \vdash_{\text{rhs}} (op \overline{arg_1}) : ty & \Rightarrow \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System} \vdash (op \overline{arg_1}) & = \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System}_{\text{premises}}((op \overline{arg_1}) & ) = \Gamma^s \\
\text{Type System} \vdash (op \overline{arg_1}) & = \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System} \vdash (op \overline{arg_1}) & = \Gamma^s \\
\end{align*} \]

\[ \begin{align*}
\text{Type System} \vdash (op \overline{arg_1}) & = \Gamma^s \\
\end{align*} \]

References


Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshtout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto
with MPS. In Third International Conference on Software Language Engineering (SLE 2010) (Lecture Notes in Computer Science), Mark van den Brand, Brian Malloy, and Steffen Staab (Eds.). Springer.