

A Case for First-Class Environments

ANONYMOUS AUTHOR(S)

Formalizations of programming languages typically adopt the substitution model from the lambda calculus. However, substitution creates notorious complications for reasoning and implementation. Furthermore, it is disconnected from practical implementations, which normally adopt environments and closures.

In this paper we advocate for formalizing programming languages using an environment-based semantics instead, and adding expressive power by making environments first-class. We present a call-by-value statically typed calculus, called λ_E , with an environment semantics. Programming language constructs for first-class environments arise naturally from the environment-based semantics, without creating significant additional complexity. λ_E is a conservative extension of the call-by-value Simply Typed Lambda Calculus (STLC). We show that reasoning about λ_E is simple, and in many cases simpler than reasoning about the traditional STLC. λ_E employs de Bruijn indices for its formalization, which fit naturally with the environment-based semantics. We show an abstract machine that implements the semantics of λ_E , and has an easy correctness proof. We also extend λ_E with references. To illustrate the expressive power, we show that λ_E can model a simple form of first-class modules, and suggest using first-class environments as an alternative to objects for modelling capabilities. λ_E is formalized in the Coq proof assistant, together with numerous technical results. The notorious binding problem in proof assistants, was not a problem in the formalization of λ_E .

1 INTRODUCTION

Formalizations of programming languages have historically been heavily influenced by the foundational concepts of the lambda calculus [Church 1941]. The substitution model, a key element of the lambda calculus, provides a powerful yet elegant mechanism for expressing computation through the process of replacing variables with their corresponding values. This model has been widely adopted in the design and formalization of modern programming languages.

Despite its elegance and widespread adoption, the substitution model, is not without drawbacks. One of the most notable complications, arising from the use of substitution, is the challenge it poses for reasoning and implementation. In particular, the process of substituting variables can lead to issues such as variable capture, where the meaning of a program may be inadvertently altered during substitution. The challenges posed by the substitution model often create complications when attempting to reason about proofs. In particular, working with proof assistants can be notoriously difficult due to the intricacies associated with formalizing binding and substitution [Abel et al. 2019; Aydemir et al. 2005; Charguéraud 2012]. Additionally, the necessity of performing substitutions throughout the entire program can result in significant computational overhead, making it less efficient for practical applications. Consequently, practical implementations of programming languages often diverge from the pure substitution model by adopting techniques such as environments and closures. By employing these concepts, programming languages can be efficiently implemented, and the execution model remains faithful to the conceptual underpinnings of the lambda calculus. However, this creates a gap between substitution-based formalizations and their efficiently implementable execution models. This gap has been noted by several researchers [Abadi et al. 1991; Amin and Rompf 2017; Biernacka and Danvy 2007; Curien et al. 1996; Curien 1991], which have argued for alternatives to the substitution model.

In this paper, we advocate for a paradigm shift in formalizing programming languages by utilizing an environment-based semantics, which offers important advantages over the traditional substitution model. We present a call-by-value statically typed calculus, called λ_E , with an environment semantics. The environment-based semantics is more closely aligned with the practical execution models that rely on environments and closures. We argue that another advantage of adopting an environment-based semantics is simplified reasoning. λ_E employs de Bruijn indices for its formalization, which fit naturally with the environment-based semantics. Formalizations can be done

with a simplified formulations of de Bruijn indices, and without the need for shifting/unshifting operations, which are renowned for complexities when using formalizations based on de Bruijn indices. This simplification is particularly beneficial when working with proof assistants, and it allows for a more efficient and manageable approach to formalizing programming languages.

We formalize a substantial numbers of results in the Coq proof assistant for λ_E . These results include: syntactic type soundness; semantic type soundness and normalization; correspondence between small and big-step semantics; simulation and conservativity results for the (traditional) call-by-value Simply Typed Lambda Calculus (STLC); the design and correctness of an abstract machine; and others. We also extend λ_E with references. In all these results, we only had to reason about substitutions when showing the conservativity to the STLC. This is, of course unavoidable, since the standard formulation of the STLC involves substitution. Overall we never felt that issues with binding were in our way in our Coq formalization. Throughout the paper we will document places in our proofs, where we believe that the environment-based approach made a significant difference to an approach based on substitution.

Programming language constructs for first-class environments arise naturally from the environment-based small-step semantics that we propose, without creating significant additional complexity. Thus, in λ_E , we elevate environments to first-class citizens within the language, while retaining the simplicity of reasoning and binding. This is in contrast with previous work on first-class environments [Nishizaki 1994; Sato et al. 2001, 2002], where issues with binding introduces significant complexity. First-class environments are useful to enhance both the expressive power and practicality of the language, empowering developers to fully harness the potential of an environment-based semantics in their programs. The introduction of first-class environments enables the creation of functions that can accept environments as arguments or return them as results. This paves the way for more powerful and flexible abstractions. The increased expressive power can facilitate the development of sophisticated and modular programs, as developers can now leverage environments to encapsulate and manage variable bindings more effectively. To illustrate the expressive power of λ_E , we show that λ_E can model a simple form of first-class modules, and suggest using first-class environments as an alternative to objects for modelling capabilities [Miller 2006; Miller et al. 2000]. In essence, since we can use first-class environments to model modules, we can use these modules as capabilities that can be passed to clients of the modules.

Overall, we believe that there is a synergy of factors that make calculi with an environment-based semantics and first-class environments appealing. Nevertheless we acknowledge that the study in this paper is limited, and that with more advanced programming language features additional complications may arise from binding. Section 6 discusses some expected challenges and limitations, as well as possible variants in the design. The contributions of this paper are:

- **The λ_E calculus**, which is a call-by-value statically typed calculus with first-class environments. λ_E adopts an environment-based semantics and supports lookups by de Bruijn indices or labels. We illustrate its applicability to encodings of first-class modules and capabilities.
- **A small-step style of environment semantics with easy metatheory**. The λ_E calculus is shown to have desirable properties including determinism, type soundness, and normalization. The proofs in the metatheory are simple and do not involve reasoning about substitutions.
- **Operational correspondence** between the λ_E calculus and other models. The results include equivalence between small-step and big-step semantics, completeness and conservativity with respect to the lambda calculus, and correctness of compilation to an abstract machine.
- **An extension of λ_E with references**: We also investigate a type sound and deterministic extension of the λ_E calculus with mutable references.
- **Coq formalization**: We have formalized all the calculi and proofs in this paper in the Coq theorem prover. The proofs are available in the supplementary materials.

2 OVERVIEW

This section provides an overview of our work. We start with some relevant background, and then present the key ideas in our work.

2.1 Background: Substitution-based Semantics

The lambda calculus [Church 1941] is an abstract mathematical model of computation, which can be regarded as the theoretical foundation of functional programming. Beta reduction, which reduces function applications, is the core rule that performs computation. In the typical presentation of the operational semantics for the lambda calculus [Barendregt 1985], a substitution-based semantics is usually employed. A function application reduces by substituting the right-hand component in the body of the function definition, which is formally expressed as the following rule:

$$\frac{}{(\lambda x. e_1) e_2 \hookrightarrow e_1[e_2/x]} \text{BETA}$$

While the substitution model is, by far, the most widely adopted model for formalizing lambda calculi, it does have some important drawbacks, which we discuss next.

Reasoning about binding and substitutions is tricky. Defining a correct capture-avoiding substitution is quite tricky, as the representation of variable names, freshness, and α -conversion need to be considered carefully. The Barendregt convention is often adopted in pencil-and-paper proofs, where all bound variables are assumed to be distinct and their names should be different from any free variables. Nevertheless, nowadays proofs about programming languages tend to be mechanized using proof assistants to enhance the rigor of the work. Therefore, a rigorous representation of variables, such as de Bruijn indices [De Bruijn 1972] (or others, such as locally nameless [Charguéraud 2012]) should be applied in mechanized proofs.

Reasoning about substitutions with de Bruijn indices is however tedious. Several lemmas need to be established to characterize the interaction between shifting and substitution [Aydemir et al. 2005]. The manipulation of terms becomes fragile and error-prone. For example, incrementing the mismatched index can easily disrupt the binding connection and cause issues in the encoding. Proving normalization with a logical relation for the substitution-based lambda calculus requires even stronger reasoning about substitutions, since we need to define simultaneous substitutions for dealing with open terms. As a consequence, the proof relies on many properties about context extensions, renaming, and composition of single and simultaneous substitution [Abel et al. 2019]. For example, the proof of normalization using de Bruijn indices in Abel et al.'s formalization consists of a large amount of lemmas stating properties about composition, decomposition, and associativity of substitutions, which requires development of a sophisticated theory of substitutions.

Gap to implementations. In most practical implementations of programming languages, beta reduction is not implemented with substitution, since substitution is algorithmically inefficient. The essential problem with substitutions is that it requires traversing terms to replace the variables by their corresponding values or expressions. Since applications are pervasive, this means that some expressions need to be traversed multiple times to replace different variables, which is very costly. Since implementations, such as abstract machines, avoid substitutions, this complicates establishing the connection between substitution based calculi and those implementations.

2.2 Background: Environment-based Semantics

In practical implementations, an explicit representation of environments is used. In this way expressions are never traversed or modified. The substitution for a variable is deferred in the sense that the associated value is looked up only if a variable needs to be evaluated. As a result,

many abstract machines, such as Krivine’s machine [Crégut 1990], the CEK machine [Felleisen and Friedman 1987], and the SECD machine [Landin 1964], can be implemented efficiently with environments. In such a setting, de Bruijn indices can be adapted easily by abstract machines and work quite well.

While much less common in formalizations of lambda calculi, there are some alternative presentations of the lambda calculus that employ an environment-based semantics, instead of substitution. Typically, the environment-based evaluation of lambda calculus is presented with a big-step semantics [Kahn 1987]. The evaluation judgment $E \vdash e \Rightarrow v$ means that, with the global environment E , expression e evaluates to the value v . Function closures [Landin 1964] naturally appear as part of the formalization in such a formulation. A function closure is basically a pair $(\lambda x. e)[E]$ containing a function and an environment E associating values to the free variables of the function.

Curien [1991] went one step further to internalize environments and closures into the language in the closure calculus $\lambda\rho$ using de Bruijn indices. A follow-up untyped calculus $\lambda\hat{\rho}$ extends $\lambda\rho$ with one-step reductions [Biernacka and Danvy 2007]. In this way, small-step semantics and weak reduction strategies can be expressed in $\lambda\hat{\rho}$. Closure calculi do not employ a global environment as in the conventional representation of big-step semantics. Instead, an environment is propagated to each component of an expression. Thus, in the $\lambda\hat{\rho}$ calculus with the call-by-value strategy, the application $((\lambda.e)[s])v$ is beta-reduced to $e[v \cdot s]$, in which $v \cdot s$ is the environment by appending s with v . In this way, a *general* notion of closures $e[s]$ is introduced to the syntax of closure calculi, where s denotes the environment for computing e .

Lambda calculi based on an environment semantics provide solutions to the problems that we identified in Section 2.1:

- **Simplified binding.** Calculi based on environments do not need substitution. Thus they can avoid the issues related to substitution. In particular, with an environment-based semantics de Bruijn indices work quite well. The cumbersome aspects of working with de Bruijn indices, which are related to substitutions, can be avoided.
- **The gap to implementations is smaller.** If a lambda calculus is based on an environment semantics, then it has a smaller gap to implementations, such as abstract machines. Since implementations also use an environment-based approach, then reasoning about the relationship between the lambda calculus and the abstract machine is simpler.

2.3 First-class Environments

We can go one more step further from closure calculi. In closure calculi, environments serve as a tool at the meta-level for providing formal semantics of the languages. However, environments themselves do not appear as syntactic entities within these languages. We can generalize the notion of environments and closures from closure calculi, so that environments are *first-class* expressions, and $e_1 \llbracket e_2 \rrbracket$ denotes the evaluation of an expression e_2 in an environment e_1 . Following this idea, several statically typed calculi with first-class environments have been proposed [Nishizaki 1994; Sato et al. 2001; Tan and Oliveira 2023].

Unfortunately, unlike closure calculi, where binding is very simple, existing calculi with first-class environments have to deal with problems that complicate the binding story again. Some typed calculi supporting environments as a first-class construct have been considered from a theoretical perspective. For example, the $\lambda_{\text{env}}^{\rightarrow}$ calculus proposed by Nishizaki and the λ_e calculus proposed by Sato et al. are both based on explicit substitutions [Abadi et al. 1991]. Moreover, both of them use named presentation for variables and employ record-like environments $\{e_1/x_1, \dots, e_n/x_n\}$. As a result, $(\lambda x. e_1) e_2$ is reduced to $\{e_1/x\} \llbracket e_2 \rrbracket$, where $\{e_1/x\}$ is the environment for evaluating e_2 . Consider the following example:

$$\begin{aligned}
& (\lambda z. ((\lambda y. \lambda x. y) \ z \llbracket x \rrbracket \ 10)) \{e_1/x_1, \dots, e_n/x_n\} \\
& \hookrightarrow \{\{e_1/x_1, \dots, e_n/x_n\}/z\} \llbracket (\lambda y. \lambda x. y) \ z \llbracket x \rrbracket \ 10 \rrbracket \\
& \hookrightarrow \{\{e_1/x_1, \dots, e_n/x_n\}/z\} \llbracket (\{z \llbracket x \rrbracket /y\} \llbracket \lambda x. y \rrbracket) \ 10 \rrbracket
\end{aligned}$$

Since environments are first-class, they can be passed to a function. The initial expression above is an application of a function with an environment. Inside the function body, there is another application, which is expected to return the first argument. Now we evaluate this expression. The first step is performing beta reduction, so that variable z is associated with this expression. The second step is a beta reduction inside the function body, which produces $\{z \llbracket x \rrbracket /y\} \llbracket \lambda x. y \rrbracket$. Now we need to consider how to deal with the evaluation of $\lambda x. y$ in the environment $\{z \llbracket x \rrbracket /y\}$. One way is to push the environment into the lambda abstraction, which gives $\lambda x. (\{z \llbracket x \rrbracket /y\} \llbracket y \rrbracket)$. However, in order to ensure α -equivalence, we need to make sure that x is not free in $z \llbracket x \rrbracket$ before pushing the environment. This introduces a question: how can we determine the free variables of $z \llbracket x \rrbracket$? We know that z is associated with $\{e_1/x_1, \dots, e_n/x_n\}$, but this information cannot be passed to the evaluation of $\{z \llbracket x \rrbracket /y\} \llbracket \lambda x. y \rrbracket$.

The $\lambda_{\text{env}}^{\rightarrow}$ calculus forbids substitution under lambdas to avoid the issue above. In general, the problem of defining free variables for an expression is not considered in the $\lambda_{\text{env}}^{\rightarrow}$ calculus. As a result, the issues caused by α -equivalence and variable renaming are not investigated. In such a setting, the property of conservativity over the lambda calculus is not satisfied in $\lambda_{\text{env}}^{\rightarrow}$. In other words, $\lambda_{\text{env}}^{\rightarrow}$ cannot encode all the behaviour of the lambda calculus. The $\lambda\epsilon$ calculus allows substitution under lambdas but it requires significant machinery. We know from the example above that z does not have enough information about free variables from its syntactic representation. To solve this problem, the definition of freeness in $\lambda\epsilon$ resorts to the information from types: x is free in $e_1 \llbracket e_2 \rrbracket$ if x is free in both e_1 and e_2 , and x does not exist in the set of names collected from the type of e_1 . For example, if the type of z is an environment type $\{x^A, y^B\}$, then x is not free in $z \llbracket x \rrbracket$, since x is “bound” by the type of z even though z is just a variable. On the other hand, x is free in $z \llbracket x \rrbracket$ if z has type $\{u^A, y^B\}$ which does not contain x .

In this setting, the definition of freeness is entwined with environment types in $\lambda\epsilon$. The correctness of reduction rules requires carefully considering the interaction between variables, environments and types. Moreover, we need devices and constraints, such as positions and occurrences, that identify the scope of variables within the context of first-class environments. Consequently, the reasoning involved about binding is complex, and it is not clear whether such a formulation can be easily adapted to theorem provers. Although lambda abstractions allow local renaming of lambda-bound variables to fresh names, α -conversion is not permitted on variables bound by the environment as they resemble labels. Otherwise the freeness would be broken. For example, y is free in $\{1/x\} \llbracket x \rrbracket$, but it is not free in $\{1/y\} \llbracket y \rrbracket$ which is α -converted from $\{1/x\} \llbracket x \rrbracket$.

The E_i calculus introduced by Tan and Oliveira is another typed calculus with first-class environments. In E_i , variables are encoded as label names. The name bound by the abstraction $\{e\}^\bullet$ needs to match with the label name from the argument. For example, $\{e\}^\bullet : \{x : \text{Int}\} \rightarrow \text{Int}$ would only accept an argument labelled with x such as $\{x = 1\}$. Thus, α -conversion is not allowed on variables bound by an abstraction, since $\{e\}^\bullet : \{y : \text{Int}\} \rightarrow \text{Int}$ cannot accept $\{x = 1\}$. To model conventional lambda abstractions, a special kind of abstractions $\{e\}^\circ : \{x : A\} \rightarrow B$ is proposed where x is only used internally in the function body. Thus, $\{e\}^\circ : \{x : \text{Int}\} \rightarrow \text{Int}$ can accept 1 as an argument. However, the concise connection between such abstractions and standard lambda abstractions is not clearly defined.

Complications of first-class environments. In addition to the complications introduced by binding, as far as we are aware none of the calculi with first-class environments comes with a corresponding

abstract machine. The complexities of binding may partly explain this, as the abstract machine would need to deal with such complexities itself. For closure calculi (or calculi with second class environments) an environment-based semantics has very appealing properties. However, these good properties seem to be lost in calculi with first-class environments. This seems unfortunate, as first-class environments introduce useful expressive power, and the added expressiveness, together with the other nicer attributes of closure calculi would make a powerful argument for the use of calculi with first-class environments.

2.4 Key Ideas

In this paper, we propose a typed, call-by-value calculus, called λ_E , with first-class environments. λ_E recovers the simplicity of binding found in closure calculi. Moreover, we provide an abstract machine that implements λ_E and is shown to correctly model the semantics of λ_E . We show that λ_E has numerous desirable properties, such as being conservative over the call-by-value lambda calculus, and being normalizing. Next we introduce the key ideas of λ_E .

Lookups by de Bruijn indices and labels. As mentioned above, the named representation can be burdensome and inefficient, especially in the context of first-class environments. To address this problem, like in closure calculi, de Bruijn indices (denoted by \underline{n}) are employed for variables in the λ_E calculus. Hence, α -equivalence coincides with syntactic equality. Thus, standard lambda abstractions are definable in λ_E . For example, the named lambda abstraction $\lambda f : \text{Int} \rightarrow \text{Int}. \lambda x : \text{Int}. f \ x$ is represented as

$$\lambda \text{Int} \rightarrow \text{Int}. \lambda \text{Int}. \underline{1} \ 0$$

Moreover, α -equivalence is also respected for those variables that are bound by the environment. In the λ_E calculus, $e_1 \triangleright e_2$, where \triangleright is called a box, denotes the evaluation of e_2 in the environment e_1 . The variables in e_2 can be a de Bruijn index bound by e_1 . For example, in the expression

$$(\cdots \circlearrowleft \text{succ}(1) \circlearrowleft 42 \circlearrowleft \text{true}) \triangleright \underline{1}$$

the environment consists of some expressions composed by the merge operator (\circlearrowleft). In the environment, the rightmost expression is indexed by 0, and the index increments from right to left. Thus, index $\underline{1}$ refers to 42 in the environment. Note that counting of indices in environments coincides with counting in typing contexts. This is achieved by the unification of typing contexts and types. In contrast, α -equivalence is not respected under an environment type in λ_E [Sato et al. 2001].

Besides de Bruijn indices, λ_E also provides label-based lookups. A label, unlike a name or a de Bruijn index, does not admit α -conversion. Labels fetch values from records. For example,

$$\lambda\{l : \text{Int} \rightarrow \text{Int}\}. \lambda \text{Int}. \underline{l} \ 0$$

is a well-typed abstraction, where type $\text{Int} \rightarrow \text{Int}$ is looked up by \underline{l} . This is also true for looking up a value in an environment:

$$(\cdots \circlearrowleft \text{succ}(1) \circlearrowleft \{l = 42\} \circlearrowleft \text{true}) \triangleright \underline{l}$$

In λ_E , ambiguous lookups by labels are forbidden: ambiguous lookups are type errors. For example,

$$(\cdots \circlearrowleft \text{succ}(1) \circlearrowleft \{l = 42\} \circlearrowleft \{l = \text{true}\}) \triangleright \underline{l}$$

is not allowed since the lookup would result in either 42 or true. However, the environment itself is well-typed even though l appears twice. This treatment of environments is similar to how Haskell, and some other languages, deal with ambiguous occurrences of names imported from multiple modules. Note that this is also a more liberal treatment than the one in [Nishizaki 1994; Sato et al. 2001; Tan and Oliveira 2023], where the names in an environment should be distinct.

Moreover, in λ_E we have a query construct $?$ to reify the environment, and $e.n$ and $e.l$ for index-based and label-based lookups for an expression respectively. In this way, a de Bruijn index \underline{n} is desugared as $?n$, and a label \underline{l} is desugared as $?l$.

A small-step style for environment-based semantics. While de Bruijn indices are used for representing variables, substitution is not used. Instead, λ_E employs a style of environment semantics in which function closures are set to be values. In the lexical environment for a function closure, each expression is a value. The reduction relation $v \vdash e_1 \hookrightarrow e_2$ is read as: with the global environment v , expression e_1 reduces to e_2 in one step. Moreover, λ_E adopts call-by-value evaluation. In particular, an expression $e_1 \triangleright e_2$ with the box construct is evaluated using the call-by-value strategy. The environment e_1 , which possibly contains open expressions, evaluates to a value environment v_1 within the global environment first, and then e_2 evaluates under v_1 . For example,

$$\begin{aligned} \{l_1 = 42\} &\vdash \{l_2 = \underline{l_1}\} \triangleright \underline{l_2} \\ &\hookrightarrow^* \{l_2 = 42\} \triangleright \underline{l_2} \\ &\hookrightarrow^* 42 \end{aligned}$$

This setting gives rise to simple reasoning about the evaluation of boxes, especially in the case of nested evaluations. Every $e_1 \triangleright e_2$ needs to evaluate to a value. In this way, the evaluation is simplified, and the need for determining free variables is eliminated. For example, suppose that the initial environment is empty (ε) and v is $\{l_1 = 3\}, \{l_2 = 4\}$, and we have the evaluation:

$$\begin{aligned} \varepsilon &\vdash (\lambda\{l_1 : \text{Int}\} \& \{l_2 : \text{Int}\}. ((\lambda \text{Int}. \lambda \text{Int}. \underline{1}) (\underline{0} \triangleright \underline{l_1}) 10)) \ v \\ &\hookrightarrow^* (\varepsilon, v) \triangleright (\lambda \text{Int}. \lambda \text{Int}. \underline{1}) (\underline{0} \triangleright \underline{l_1}) 10 \\ &\hookrightarrow^* (\varepsilon, v) \triangleright (\varepsilon, v \triangleright \lambda \text{Int}. \lambda \text{Int}. \underline{1}) (\underline{0} \triangleright \underline{l_1}) 10 \\ &\hookrightarrow^* (\varepsilon, v) \triangleright (\varepsilon, v \triangleright \lambda \text{Int}. \lambda \text{Int}. \underline{1}) 3 10 \\ &\hookrightarrow^* (\varepsilon, v) \triangleright (\varepsilon, v, 3, 10 \triangleright \underline{1}) \\ &\hookrightarrow^* 3 \end{aligned}$$

This is similar to the problematic example shown in Section 2.3. A function is applied with environment v , and $\underline{0} \triangleright \underline{l_1}$ in the function body that expects to fetch the value of l_1 . However, $\underline{0} \triangleright \underline{l_1}$ is not pushed into a lambda abstraction. Instead, it evaluates directly to 3 in environment ε, v . As a result, the unevaluated $\underline{0} \triangleright \underline{l_1}$ is never added into the lexical environment of the function closure.

Easy proofs of metatheory and operational correspondence. Proving type safety and termination of λ_E is easy in our setting of de Bruijn indices and environment-based semantics. In particular, the proof of termination is easier than in $\lambda_{\text{env}}^{\rightarrow}$ and λ_E . We do not need to translate the calculus to another calculus (like in $\lambda_{\text{env}}^{\rightarrow}$), or define some advanced concepts to record the flow of environments (like in λ_E). The proof of termination is even easier than in substitution-based lambda calculi. No reasoning of substitutions is needed given that the global environment in λ_E contains enough information. Furthermore, the use of de Bruijn indices also paves the way to connect λ_E with other conventional computational models using de Bruijn indices, which is not done for E_i .

2.5 Applications of the λ_E Calculus

First-class environments provide interesting expressiveness over conventional lambda calculi. Several authors, have already argued for the ability of first-class environments to model a form of first-class modules and/or declarations [Gelernter et al. 1987; Jagannathan 1994; Queinnec and Roue 1996]. We revisit this ability in λ_E . Furthermore, we argue that first-class environments provide an alternative to model *capabilities* [Miller 2006; Miller et al. 2000], which are commonly

modelled via objects. We propose using first-class environments instead, as an alternative to objects to model capabilities.

First-class modules. Cardelli [1997] proposed a simply-typed calculus with high-level abstractions for modules and interfaces, where a module consists of abstract interfaces and a list of function declarations that may require the interfaces. To illustrate that a simple form of first-class modules in Cardelli's framework can be modelled with first-class environments using λ_E , consider two libraries represented as modules:

```

module IOUtils where
  print : String → Void = ...
  ...
module ListUtils where
  require print : String → Void

  sort : [Int] → [Int] = ...
  printLst : [Int] → Void = ...

```

IOUtils is a module that consists of several IO-related operations such as a print function. Module ListUtils requires an external print function. Moreover, it contains a sort function that sorts a list of integers and printLst that calls the imported function print for printing a list. A user can also create a program as follows:

```

module Main where
  require sort : [Int] → [Int]
  require printLst : [Int] → Void

  printSorted : [Int] → Void = printLst . sort
  main : Void = printSorted([2, 1, 3])

```

Here, Main requires functions sort and printLst. In the module, printSorted uses these two functionalities to print a sorted list, and main applies printSorted with [2, 1, 3]. The sorted list [1, 2, 3] should be printed after the execution of main.

The modules above can be modelled as first-class entities in λ_E :

```

{IOUtils : {print : String → Void} & ...}
{IOUtils = {print = ...} , ...}

{ListUtils : {print : String → Void} →
  {sort : [Int] → [Int]} & {printLst : [Int] → Void}}
{ListUtils = λ{print: String → Void}.
  ({sort = ...} , {printLst = ...}) }

```

Modules and declarations are modelled as single-field records. The merge operator is used to compose declarations of functions in a module. A module is not fully implemented if a module requires external functions. For instance ListUtils is not fully implemented. Nevertheless, functions defined in such a module can rely on the abstract interfaces. In this case, a lambda abstraction is used for modelling the required functions, and the imported interfaces are encoded as input types. Thus, to use the module we must *apply* it with some concrete implementations of the required functions. The user-defined module can be modelled in a similar manner:

```

{Main : {sort : [Int] → [Int]} & {printLst : [Int] → Void} →
  {printSorted : [Int] → Void} & {main : Void}}
{Main = λ{sort : [Int] → [Int]} & {printLst : [Int] → Void}.
  ({printSorted = printLst . sort} , {main = printSorted([2, 1, 3])) }

```

The merge operator in λ_E is dependent, thus printSorted can be referred by main. Now we can implement the ListUtils module and then pass the implementation of sort and printLst to Main:


```

393   ({ListUtilsImp = ListUtils({print = IOUtils.print}}) , {Main = Main}) ▷
394   Main({sort = ListUtilsImp.sort} , {printLst = ListUtilsImp.printLst})

```

The expression above can be compiled in an environment containing `IOUtils`, `ListUtils` and `Main`. Module `ListUtilsImp` is implemented by accepting the `print` function from `IOUtils`. With this implementation, we have a first-class environment for module `Main` that is concretized with `sort` and `printLst` coming from `ListUtilsImp`. The advantage of module systems is that a user-defined program is decoupled from concrete implementations of libraries. Hence, users are supposed not to know the concrete implementation of modules. For example, `sort` could be reimplemented later using a more efficient algorithm, and `printLst` can be updated to print a list in a prettier format. Thus we could have a different implementation of `ListUtils`, to obtain these alternative functions.

Capabilities as first-class environments. Object-capability programming [Miller 2006; Miller et al. 2000] is a programming paradigm that provides a mechanism for controlling access to resources. Several object-capability languages have been developed, such as Joe-E [Mettler et al. 2010], Caja [Miller et al. 2008], and ADsafe [Crockford 2008]. With these languages, one can grant limited privileges to untrusted code by selectively passing it relevant capabilities.

The concept of sandboxing, as an explicit authorization mechanism, is often applied in the implementation of capabilities-based languages. To ensure security, the environments are supposed not to expose security-sensitive code to developers. In other words, with sandboxing, only a specific set of APIs is exposed to users as mediated access to certain functionalities is performed. Alternatively access to the APIs can be restricted. In λ_E , we can use first-class environments/modules to model capabilities, and the `box` construct to model mediated access to capabilities. For example, it is possible that a user creates a program:

```

416  module Untrusted where
417    require sort : [Int] → [Int]
418    require print : String → Void
419
419    sorted : [Int] = sort([2, 1, 3])
420    main : Void = print("hacked")

```

This module imports a `sort` and a `print` function. Suppose that this is a module from an untrusted party. Hence, we want to restrict this module so that it is not implemented with the real `print` function and thus “hacked” will not be printed. In λ_E , we can use first-class environments and the `box` construct to achieve the goal:

```

425   ({printReal = IOUtils.print} ,
426   {ListUtilsImp = ListUtils({print = printReal}}) ,
427   {Untrusted = Untrusted} ,
428   {printImp = error "cannot access this function"}) ▷
429   Untrusted({sort = ListUtilsImp.sort} , {print = printImp})

```

Here, `printReal` is the real implementation of the `print` function. We can use it to implement `ListUtilsImp` to obtain the implementation of a `sort` function. Moreover, `printImp` that is passed to `Untrusted` is a *mock* implementation of the `print` function. In this way, `Untrusted` is not able to use the `print` capability. The `box` construct enables fine-grained control over access of capabilities. Now the execution of the expression above will output an error message instead of “hacked”.

3 THE λ_E CALCULUS: SYNTAX, TYPING AND SEMANTICS

In this section, we will introduce the λ_E calculus, which is a calculus with first-class environments. The design of first-class environments in λ_E is partly inspired by Tan and Oliveira [2023]. In particular we borrow the following ideas: typing contexts are modelled as types; runtime environments

Types	$A, B, \Gamma ::= \text{Int} \mid \varepsilon \mid A \& B \mid A \rightarrow B \mid \{l : A\}$
Expressions	$e ::= ? \mid e.n \mid i \mid \varepsilon \mid \lambda A. e \mid e_1 \triangleright e_2 \mid \langle v, \lambda A. e \rangle \mid e_1 e_2 \mid e_1 \circ e_2 \mid \{l = e\} \mid e.l$
Values	$v ::= i \mid \varepsilon \mid \langle v, \lambda A. e \rangle \mid v_1 \circ v_2 \mid \{l = v\}$
Frames	$F ::= [] . n \mid [] \circ e \mid [] e \mid [] \triangleright e \mid v [] \mid \{l = []\} \mid [] . l$
Syntactic Sugar	$\underline{n} \triangleq ?.n$ $\underline{l} \triangleq ?.l$

Fig. 1. λ_E syntax

as values; and compositions of environments as merges. However, our approach to binding is based on de Bruijn indices, making modelling standard lambda abstractions simple and straightforward.

3.1 Syntax

The syntax of the λ_E calculus is shown in Figure 1. In our presentation we use colors to highlight the constructs, and corresponding rules, that are related to first-class environments. Blue is used to highlight the constructs related to standard environment lookup. Red is used to highlight constructs that can be used to create labelled expressions and lookup by label. We also use yellow, to highlight boxes which, in a calculus with first-class environments, can have a very general form. Though λ_E supports first-class environments, we can restrict the expressiveness of λ_E such that first-class environments are disabled while we still have the notion of closures. To do so we remove or change the colored constructs. We show how this alternative design works in Section 6.

Types and contexts. Meta-variables A, B, Γ range over types and typing contexts. Note that in our calculus there is no distinction between contexts and types. Contexts are just types and any type can act as a context. Types/contexts include base types (Int), a unit type used to denote empty environments (ε), a limited form of intersection types ($A \& B$), arrow types ($A \rightarrow B$), and single-field record types ($\{l : A\}$). A multi-field record type $\{l_1 : A_1, \dots, l_n : A_n\}$ can be obtained by intersecting several single-field records: $\{l_1 : A_1\} \& \dots \& \{l_n : A_n\}$ [Reynolds 1997].

Expressions. Meta-variables e range over expressions. Expressions include several standard constructs: integers (i), the unit expression (ε), applications ($e_1 e_2$), merges ($e_1 \circ e_2$) and lambda abstractions ($\lambda A. e$). Merges are left associative. Thus, $e_1 \circ e_2 \circ e_3$ is interpreted as $(e_1 \circ e_2) \circ e_3$. Note that de Bruijn notation is applied in our calculus, and the variable bound by a lambda abstraction is nameless. In addition, we have boxes ($e_1 \triangleright e_2$) for running computations under a given environment, and closures ($\langle v, \lambda A. e \rangle$) that record the lexical environment of a function. The query construct (?) reifies the current environment, and the projection ($e.n$) construct returns the n -th element of an expression. In this way, a variable, denoted by a de Bruijn index \underline{n} , can be regarded as $?.n$, which is a compound expression with a query and a projection. Single-field records are denoted by $\{l = e\}$. A multi-field record $\{l_1 = e_1, \dots, l_n = e_n\}$ can be represented as a merge of single-field records $\{l_1 = e_1\} \circ \dots \circ \{l_n = e_n\}$. Finally, besides being able to lookup an environment by a de Bruijn index, it is also possible to have labelled entries in the environment and lookup by label. Using the selection construct ($e.l$), the field labelled by l from a multi-field record can be fetched. Similarly to lookup by de Bruijn indices, we can also have syntactic sugar for looking up a labelled entry in the environment by its label \underline{l} .

Values and frames. Meta-variables v range over values. A value is either an integer (i), the unit expression (ε), a closure ($\langle v, \lambda A. e \rangle$) where the lexical environment is also a value, a merge ($v_1 \circ v_2$) of two values, or a record whose field is also a value ($\{l = v\}$). Frames (F) include applications, merges, boxes, projections, records, and selections.

491	$\boxed{l : A \in B}$							(Containment)
492								
493	CTM-RCD	CTM-ANDL	CTM-ANDR					
494	$\frac{}{l : A \in \{l : A\}}$	$\frac{l : A \in B \quad l \notin \text{label}(C)}{l : A \in B \& C}$	$\frac{l : A \in C \quad l \notin \text{label}(B)}{l : A \in B \& C}$					
495								
496	$\boxed{\Gamma \vdash e : A}$							(Typing)
497								
498	TYP-CTX	TYP-PROJ	TYP-LIT	TYP-TOP	TYP-BOX			
499	$\frac{}{\Gamma \vdash ? : \Gamma}$	$\frac{\Gamma \vdash e : A}{\Gamma \vdash e.n : \text{lookup}(A, n)}$	$\frac{}{\Gamma \vdash i : \text{Int}}$	$\frac{}{\Gamma \vdash \varepsilon : \varepsilon}$	$\frac{\Gamma \vdash e_1 : \Gamma_1 \quad \Gamma_1 \vdash e_2 : A}{\Gamma \vdash e_1 \triangleright e_2 : A}$			
500								
501	TYP-MERGE	TYP-APP			TYP-LAM			
502	$\frac{\Gamma \vdash e_1 : A \quad \Gamma \& A \vdash e_2 : B}{\Gamma \vdash e_1, e_2 : A \& B}$	$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : B}$			$\frac{}{\Gamma \& A \vdash e : B}$			
503								
504								
505	TYP-CLOS	TYP-RCD	TYP-SEL					
506	$\frac{\varepsilon \vdash v : \Gamma_1 \quad \Gamma_1 \& A \vdash e : B}{\Gamma \vdash \langle v, \lambda A. e \rangle : A \rightarrow B}$	$\frac{\Gamma \vdash e : A}{\Gamma \vdash \{l = e\} : \{l : A\}}$	$\frac{\Gamma \vdash e : B \quad l : A \in B}{\Gamma \vdash e.l : A}$					
507								
508								
509								

Fig. 2. Type System.

3.2 Type System

Figure 2 shows the type system. The typing rules for integers, the unit expression, applications and lambda abstractions are standard. Rule **TYP-LAM** is the typing rule for lambda abstractions. Since de Bruijn notation is used in our calculus, the variable bound by the current λ is represented as 0 . In rule **TYP-LAM**, the type of 0 , which is the input type A , is appended to the context Γ by intersecting Γ and A , such that it is the rightmost type in the context. This matches the conventional right-to-left counting of λ binders. The query construct $?$ can synthesize the context (rule **TYP-CTX**). Since a context is simply a type in our setting, the type of $?$ is the current context. Rule **TYP-PROJ** is the typing rule for projections. The type of a query $e.n$ is the n -th element of the type of e that is obtained by the lookup function:

$$\text{lookup}(A \& B, 0) = B$$

$$\text{lookup}(A \& B, n + 1) = \text{lookup}(A, n)$$

Consequently, the type of variable \underline{n} (encoded as $?n$) is the n -th element in the typing context. Rule **TYP-MERGE** is the typing rule for merges. The right branch e_2 may depend on the left branch e_1 in merge e_1, e_2 , since the context for typing e_2 is the type $\Gamma \& A$, which is the intersection of the typing context for e_1, e_2 and the type of e_1 . Rule **TYP-BOX** is the rule for boxes. In order to typecheck a box $e_1 \triangleright e_2$, we first check that e_1 is well-typed under the global context and then e_2 is checked under the type of e_1 . Rule **TYP-CLOS** is the rule for closures. For a closure $\langle v, \lambda A. e \rangle$, the body e is checked under the type of the lexical environment v and the input type A . Since v is a value that is closed, the typing context for it can be ε in the rule. Rule **TYP-SEL** is the rule for selection. The type of $e.l$ is A if the type of e contains the entry $l : A$. In order to ensure determinism, when there are multiple entries for l in B , the selection fails according to the containment judgment $l : A \in B$.

3.3 Dynamic Semantics

Figure 3 shows the small-step call-by-value operational semantics for λ_E . The reduction has the form of $v \vdash e_1 \hookrightarrow e_2$. The value v denotes the runtime environment. We define multi-step reduction \hookrightarrow^* as the reflexive and transitive closure of \hookrightarrow . Rule **STEP-CTX** reduces a query $?$ to the current runtime environment. To evaluate a projection $e.n$, expression e is evaluated to a value v first, after

$\boxed{v.l \rightsquigarrow v'}$			(Selection)
SEL-REC	SEL-MRGL	SEL-MRGR	
$\frac{}{\{l = v\}.l \rightsquigarrow v}$	$\frac{v_1.l \rightsquigarrow v}{(v_1 \text{ } v_2).l \rightsquigarrow v}$	$\frac{v_2.l \rightsquigarrow v}{(v_1 \text{ } v_2).l \rightsquigarrow v}$	
$\boxed{v \vdash e \hookrightarrow e'}$			(Small-step Operational Semantics)
STEP-EVAL	STEP-CTX	STEP-PROJ	
$\frac{v \vdash e \hookrightarrow e'}{v \vdash F[e] \hookrightarrow F[e']}$	$\frac{}{v \vdash ? \hookrightarrow v}$	$\frac{}{v \vdash v_1.n \hookrightarrow \text{lookupv}(v_1, n)}$	
STEP-MERGER	STEP-BOX	STEP-BOXV	
$\frac{v \text{ } v_1 \vdash e_2 \hookrightarrow e'_2}{v \vdash v_1 \text{ } e_2 \hookrightarrow v_1 \text{ } e'_2}$	$\frac{v_1 \vdash e \hookrightarrow e'}{v \vdash v_1 \triangleright e \hookrightarrow v_1 \triangleright e'}$	$\frac{}{v \vdash v_1 \triangleright v_2 \hookrightarrow v_2}$	
STEP-CLOSURE	STEP-BETA	STEP-SEL	
$\frac{}{v \vdash \lambda A. e \hookrightarrow \langle v, \lambda A. e \rangle}$	$\frac{}{v \vdash \langle v_1, \lambda A. e \rangle v_2 \hookrightarrow (v_1 \text{ } v_2) \triangleright e}$	$\frac{v_1.l \rightsquigarrow v_2}{v \vdash v_1.l \hookrightarrow v_2}$	

Fig. 3. Dynamic Semantics.

which rule **STEP-PROJ** is triggered. The n -th element of v is fetched by lookupv :

$$\text{lookupv}(v_1 \text{ } v_2, 0) = v_2$$

$$\text{lookupv}(v_1 \text{ } v_2, n + 1) = \text{lookupv}(v_1, n)$$

A merge $e_1 \text{ } e_2$ is evaluated from left to right. After e_1 is evaluated to a value v_1 , v_1 is added to the runtime environment by rule **STEP-MERGER**, and e_2 can refer to both v and v_1 . An application $e_1 e_2$ is also evaluated from left to right. We have closures, instead of lambda abstractions, as values in our calculus. Hence, e_1 in the application evaluates to a closure $\langle v_1, \lambda A. e \rangle$. Moreover, beta reduction is substitution-free: argument v_2 is not substituted into the function body. Instead, rule **STEP-BETA** adds v_2 into the runtime environment by merging v_1 , the lexical environment for the closure, with v_2 . In our small-step reduction, this action is represented as a single step, and the result of beta reduction is $(v_1 \text{ } v_2) \triangleright e$, i.e., e boxed with $v_1 \text{ } v_2$. After that, the function body e can be evaluated further under environment $v_1 \text{ } v_2$. To evaluate a box $e_1 \triangleright e_2$, we first evaluate e_1 to a value v_1 , and then e_2 is reduced under environment v_1 by rule **STEP-BOX**. Once the evaluation of e_2 is done, rule **STEP-BOXV** extracts the value from the box. For a selection $e.l$, e is evaluated to a value v and then $v.l \rightsquigarrow v'$ is called by rule **STEP-SEL** to get the value labelled by l .

4 REASONING WITH ENVIRONMENT-BASED SEMANTICS

In this section we show that λ_E has numerous interesting properties. More importantly, reasoning about these properties is simple, and we argue that is often easier than formulations based on a standard lambda calculus with substitution-based beta-reduction. In particular, because there is no substitution in the calculus, the use of de Bruijn indices is very natural and no reasoning about shifting/unshifting is ever needed. Moreover, the formalization style is very friendly to mechanized theorem provers where binding is a well-known source of complexity [Aydemir et al. 2005; Charguéraud 2012]. In our Coq formalization binding was never a problem.

4.1 Determinism and Syntactic Type Soundness

The operational semantics of λ_E is deterministic and type-sound. The statement of these theorems follows the form by Tan and Oliveira. In contrast to substitution-based reduction, a

value v is responsible for tracking the runtime environment is updated during reduction. As a result, the theorem would not be strong enough if we set v to the empty environment ε . Hence, an arbitrary environment needs to be considered in the theorems, and we need to ensure that the environment involved is well-typed. Note that we do not need to define a typing judgment for environments, since environments and expressions share the same set of typing rules as they are unified. Importantly, note that the substitution lemma, which is used in traditional type soundness proofs [Wright and Felleisen 1994], is not needed, since we do not use substitution in the semantics.

We first show that our type system prevents ambiguous selections. A repeated label is allowed to appear in a multi-field record. For example, let e be $\{l_1 = 1\} \circ \{l_2 = \text{true}\} \circ \{l_2 = 2\}$ and it has type $\{l_1 : \text{Int}\} \& \{l_2 : \text{Bool}\} \& \{l_2 : \text{Int}\}$. Invoking selection of l_1 for e gives 1 safely. However, if the label is l_2 , then $e.l_2$ would evaluate to either true or 2, resulting in ambiguity. The containment judgment $l : A \in B$ in the type system rules out the ambiguous selections statically. Hence, $e.l_2$ is forbidden. Formally, we have the following lemma stating that well-typed selection is deterministic:

LEMMA 4.1 (DETERMINISM OF SELECTION). *If $\Gamma \vdash v.l : A$ and $v.l \rightsquigarrow v_1$ and $v.l \rightsquigarrow v_2$, then $v_1 = v_2$.*

With the help of determinism of selection, we have determinism of reduction.

THEOREM 4.2 (GENERALIZED DETERMINISM). *If $\varepsilon \vdash v : \Gamma$ and $\Gamma \vdash e : A$ and $v \vdash e \hookrightarrow e_1$ and $v \vdash e \hookrightarrow e_2$, then $e_1 = e_2$.*

By instantiating v and Γ with ε , we obtain the standard determinism theorem as a corollary.

COROLLARY 4.3 (DETERMINISM). *If $\varepsilon \vdash e : A$ and $\varepsilon \vdash e \hookrightarrow e_1$ and $\varepsilon \vdash e \hookrightarrow e_2$, then $e_1 = e_2$.*

Next we introduce the progress and preservation theorems for type-soundness. We first prove progress and preservation for the lookup function and label selection.

LEMMA 4.4 (PROGRESS OF LOOKUP). *If $\varepsilon \vdash v : A$ and $\text{lookup}(A, n) = B$, then there exists v' s.t. $\text{lookupv}(v, n) = v'$.*

LEMMA 4.5 (PRESERVATION OF LOOKUP). *If $\varepsilon \vdash v : A$ and $\text{lookup}(A, n) = B$ and $\text{lookupv}(v, n) = v'$, then $\varepsilon \vdash v' : B$.*

LEMMA 4.6 (PROGRESS OF SELECTION). *If $\varepsilon \vdash v : A$ and $l : B \in A$, then there exists v' s.t. $v.l \rightsquigarrow v'$.*

LEMMA 4.7 (PRESERVATION OF SELECTION). *If $\varepsilon \vdash v : A$ and $l : B \in A$ and $v.l \rightsquigarrow v'$, then $\varepsilon \vdash v' : B$.*

With the lemmas above, we know that if the environment v is well-typed, then any value produced during the evaluation of e that refers to v (by lookup or by label selection) should progress and preserve types. Consequently, we have the following generalized theorems.

THEOREM 4.8 (GENERALIZED PROGRESS). *If $\varepsilon \vdash v : \Gamma$ and $\Gamma \vdash e : A$, then either e is a value or there exists e' s.t. $v \vdash e \hookrightarrow e'$.*

THEOREM 4.9 (GENERALIZED PRESERVATION). *If $\varepsilon \vdash v : \Gamma$ and $\Gamma \vdash e : A$ and $v \vdash e \hookrightarrow e'$, then $\Gamma \vdash e' : A$.*

Standard progress and preservation theorems are now corollaries as follows.

COROLLARY 4.10 (PROGRESS). *If $\varepsilon \vdash e : A$, then either e is a value or there exists e' s.t. $\varepsilon \vdash e \hookrightarrow e'$.*

COROLLARY 4.11 (PRESERVATION). *If $\varepsilon \vdash e : A$ and $\varepsilon \vdash e \hookrightarrow e'$, then $\varepsilon \vdash e' : A$.*

4.2 Semantic Type Soundness and Termination

Termination. We also define semantic typing and prove the termination for our calculus following Tait's method [Tait 1967]. Proving the termination of the Simply Typed Lambda Calculus (STLC) by performing induction on the typing judgment directly does not work, since we are stuck at the case of application $e_1 e_2$: the induction hypotheses for e_1 and e_2 are not strong enough to conclude that $e_1 e_2$ terminates. We know that e_1 and e_2 evaluate to a lambda abstraction $\lambda x : A. e$ and a

$$\begin{aligned}
638 \quad & v \in \mathcal{V}[\varepsilon] \triangleq v = \varepsilon \\
639 \quad & v \in \mathcal{V}[\text{Int}] \triangleq \exists i, v = i \\
640 \quad & v \in \mathcal{V}[A \& B] \triangleq \exists v_1, v_2, v = v_1 \circ v_2 \wedge v_1 \in \mathcal{V}[A] \wedge v_2 \in \mathcal{V}[B] \\
641 \quad & v \in \mathcal{V}[\{l : A\}] \triangleq \exists v', v = \{l = v'\} \wedge v' \in \mathcal{V}[A] \\
642 \quad & v \in \mathcal{V}[A \rightarrow B] \triangleq \exists v_1, e, v = \langle v_1, \lambda A. e \rangle \wedge (\forall v_2 \in \mathcal{V}[A], \exists v', \varepsilon \vdash (v_1 \circ v_2) \triangleright e \hookrightarrow^* v' \wedge v' \in \mathcal{V}[B]) \\
643 \quad & \Gamma \models e : A \triangleq \forall v \in \mathcal{V}[\Gamma], \exists v', v \vdash e \hookrightarrow^* v' \wedge v' \in \mathcal{V}[A]
\end{aligned}$$

Fig. 4. Semantic Typing.

value v_2 respectively, and the beta reduction gives $e[v_2/x]$. However, whether the function body e terminates is unknown and the proof cannot proceed.

The standard method to strengthen the theorem is to define a unary logical relation (a.k.a. logical predicate) as a proxy to decompose the proof into two steps: we first embed the termination property into the logical predicate, and then prove that every well-typed expression satisfies the logical predicate. We follow this approach to prove the termination of the λ_E calculus.

Semantic typing is recursively defined in Figure 4. We define a logical predicate to include values ($v \in \mathcal{V}[A]$) and expressions ($\Gamma \models e : A$) that have the expected semantic behaviour. The semantic type $\mathcal{V}[A]$ captures all the values of type A that terminates. To be more specific, $\mathcal{V}[\varepsilon]$ includes the empty environment ε , $\mathcal{V}[\text{Int}]$ consists of all of the integers; $\mathcal{V}[A \& B]$ and $\mathcal{V}[\{l : A\}]$ are comprised of merge values and record values, respectively. $\mathcal{V}[A \rightarrow B]$ is made up of closures. Moreover, it requires that, for any value v_2 of semantic type A , the result of the beta reduction, i.e., $(v_1 \circ v_2) \triangleright e$, can be evaluated to a value v' of semantic type B . This condition ensures that the function body, which is exposed after the beta reduction, actually terminates (under the updated environment), and thus it provides a strong hypothesis for the termination proof. The definition of $\mathcal{V}[A \rightarrow B]$ is similar to the one in the proof of termination for substitution-based calculi, but a boxed expression, instead of meta-level substitution $e[v_2/x]$, is applied in the definition.

An expression e has semantic type A in typing context Γ , if e can evaluate to a value of type A in every environment v of semantic type Γ . We prove that syntactic typing entails semantic typing:

THEOREM 4.12 (SEMANTIC SOUNDNESS). *If $\Gamma \vdash e : A$ then $\Gamma \models e : A$.*

The proof is done by induction on the typing judgment. In each case, we need to show that the rule of syntactic typing can be derived for semantic typing. For example, we have the lemmas for query, box, and lambda abstraction as follows:

LEMMA 4.13 (SOUNDNESS OF QUERY). $\Gamma \models ? : \Gamma$.

LEMMA 4.14 (SOUNDNESS OF BOX). *If $\Gamma \models e_1 : \Gamma_1$ and $\Gamma_1 \models e_2 : A$, then $\Gamma \models e_1 \triangleright e_2 : A$.*

LEMMA 4.15 (SOUNDNESS OF LAMBDA ABSTRACTION). *If $\Gamma \& A \models e : B$, then $\Gamma \models \lambda A. e : A \rightarrow B$.*

Each lemma has the same “shape” of its corresponding syntactic typing rule. This property also holds for the other constructs. Next we introduce briefly the proof of Lemma 4.15, which is a hard case in the termination proof for substitution-based semantics, but is easy in our formulation. To prove $\Gamma \models \lambda A. e : A \rightarrow B$, suppose that $v_1 \in \mathcal{V}[\Gamma]$, then we have $v_1 \vdash \lambda A. e \hookrightarrow^* \langle v_1, \lambda A. e \rangle$. So we need to show $\langle v_1, \lambda A. e \rangle \in \mathcal{V}[A \rightarrow B]$ by the definition of semantic typing. By the definition of $\mathcal{V}[A \rightarrow B]$, we suppose that $v_2 \in \mathcal{V}[A]$. Now we merge the environment v_1 with v_2 , and we have $v_1 \circ v_2 \in \mathcal{V}[\Gamma \& A]$. By the assumption $\Gamma \& A \models e : B$, we know that there exists a value v' such that $v_1 \circ v_2 \vdash e \hookrightarrow^* v'$ and $v' \in \mathcal{V}[B]$. By the semantics of the box construct, we have $\varepsilon \vdash (v_1 \circ v_2) \triangleright e \hookrightarrow^* (v_1 \circ v_2) \triangleright v' \hookrightarrow^* v'$ and the proof is complete.

Now we can obtain the termination result easily because termination is plugged in the definition of semantic typing. Theorem 4.12 is in a generalized form. By picking Γ to be empty, we get the following termination theorem.

$v \vdash e \Rightarrow e'$

(Big-step Operational Semantics)

$\frac{}{v \vdash ? \Rightarrow v} \text{ BSTEP-CTX}$	$\frac{v \vdash e \Rightarrow v_1}{v \vdash e.n \Rightarrow \text{lookupv}(v_1, n)} \text{ BSTEP-PROJ}$	$\frac{}{v \vdash i \Rightarrow i} \text{ BSTEP-LIT}$	$\frac{}{v \vdash \langle v_1, \lambda A. e \rangle \Rightarrow \langle v_1, \lambda A. e \rangle} \text{ BSTEP-CLOS}$
$\frac{}{v \vdash \varepsilon \Rightarrow \varepsilon} \text{ BSTEP-UNIT}$	$\frac{v \vdash e_1 \Rightarrow v_1 \quad v \circ v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \circ e_2 \Rightarrow v_1 \circ v_2} \text{ BSTEP-MERGE}$	$\frac{v \vdash e_1 \Rightarrow v_1 \quad v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \triangleright e_2 \Rightarrow v_2} \text{ BSTEP-BOX}$	
$\frac{v \vdash e_1 \Rightarrow \langle v_1, \lambda A. e \rangle \quad v \vdash e_2 \Rightarrow v_2 \quad v_1 \circ v_2 \vdash e \Rightarrow v'}{v \vdash e_1 e_2 \Rightarrow v'} \text{ BSTEP-APP}$			
$\frac{}{v \vdash \lambda A. e \Rightarrow \langle v, \lambda A. e \rangle} \text{ BSTEP-LAM}$	$\frac{v \vdash e \Rightarrow v}{v \vdash \{l = e\} \Rightarrow \{l = v\}} \text{ BSTEP-REC}$	$\frac{v \vdash e \Rightarrow v_1 \quad v_1.l \rightsquigarrow v_2}{v \vdash e.l \Rightarrow v_2} \text{ BSTEP-SEL}$	

Fig. 5. Big-step operational semantics.

THEOREM 4.16 (TERMINATION). *If $\varepsilon \vdash e : A$ then there exists a value v s.t. $\varepsilon \vdash e \hookrightarrow^* v$.*

Our definition of semantic typing makes use of the environment-based evaluation $v \vdash e \hookrightarrow^* v'$ so that open terms in e can refer to v . In the proof of termination of STLC, a simultaneous substitution γ , which is essentially an environment, is introduced for dealing with open terms. However, since environments are not involved in the substitution-based semantics of STLC, the target expression is set to be $\gamma(e)$, the application of γ with e , for generalizing the theorem. As a result, the lambda case of semantic soundness requires careful reasoning about substitutions. In contrast, reasoning about environments is much easier in our setting and we do not need to handle substitutions at all. In addition, for a substitution-based semantics, γ cannot be any arbitrary substitution. A context relation $\gamma \models \Gamma$ is defined to ensure that γ respects termination: γ should produce expressions that are themselves terminating for all the variables in Γ . On the other hand, the λ_E calculus features *first-class* environments, and therefore $v \in \mathcal{V}[\Gamma]$ can act as such a context relation for the reason that environments are simply values.

4.3 Big-step Semantics

We also present a big-step operational semantics (shown in Figure 5) for λ_E , which is in spirit to the environment-based natural semantics [Kahn 1987]. The big-step semantics specifies the entire transition from an expression to a final value. The rules for values, lambda abstractions and applications are mostly standard. Values reduce to themselves and lambda abstractions reduce to closures. Unlike small-step reduction, rule **BSTEP-APP** does not rely on the box construct, since we do not need to store the intermediate result in the big-step evaluation. Rule **BSTEP-BOX** is the evaluation rule for boxes, where e_1 reduces to a value that acts as an environment for e_2 and then e_2 reduces to the final value. Note that rule **STEP-BOXV** in the small-step semantics is not needed in the big-step evaluation, since the result of reducing e_2 can be obtained directly from the premise of rule **BSTEP-BOX**. Also note that \underline{n} is a syntactic sugar of $?n$. Thus the rule for variables is

$$\frac{}{v \vdash \underline{n} \Rightarrow \text{lookupv}(v, n)} \text{ BSTEP-VAR}$$

In fact, the semantics in Figure 5 can be seen as a modular extension of the environment-based big-step semantics of the lambda calculus, except that the environments in the relation are values.

Next we prove that the small- and big-step semantics are equivalent. The proof is simple, since our small-step semantics is also environment-based. We first show that big-step semantics is sound w.r.t. the small-step semantics:

LEMMA 4.17. *If $v \vdash e \Rightarrow v'$ then $v \vdash e \hookrightarrow^* v'$.*

The proof is done by induction on the big-step relation. For the case of application, we know that $v \vdash e_1 \hookrightarrow^* \langle v_1, \lambda A. e \rangle$ and $v \vdash e_2 \hookrightarrow^* v_2$. Thus we have $v \vdash e_1 e_2 \hookrightarrow^* \langle v_1, \lambda A. e \rangle v_2 \hookrightarrow^* (v_2, v_1) \triangleright e$. However, we only have $v_2, v_1 \vdash e \hookrightarrow^* v'$ by the induction hypothesis for e , while our goal is to prove $v \vdash (v_2, v_1) \triangleright e \hookrightarrow^* v'$. We have the following lemma to fill this gap. That is, the global environment can be assigned as a local environment and the final result is not affected.

LEMMA 4.18. *If $v \vdash e \hookrightarrow^* v'$ and v' is a value, then $v_1 \vdash (v \triangleright e) \hookrightarrow^* v'$ for any v_1 .*

Moreover, we show that the big-step semantics is complete, which relies on the lemma 4.20, stating that one single step can be encompassed by the big-step semantics.

LEMMA 4.19. *If $v \vdash e \hookrightarrow^* v'$ and v' is a value, then $v \vdash e \Rightarrow v'$.*

LEMMA 4.20. *If $v \vdash e_1 \hookrightarrow e_2$ and $v \vdash e_2 \Rightarrow v'$, then $v \vdash e_1 \Rightarrow v'$.*

Finally, by combining lemma 4.17 and lemma 4.19, we have the equivalence result as following.

THEOREM 4.21 (EQUIVALENCE OF SEMANTICS). *$v \vdash e \Rightarrow v'$ if and only if $v \vdash e \hookrightarrow^* v'$.*

4.4 Conservativity over the Simply Typed Lambda Calculus

For our calculus to serve as a replacement to the Simply Typed Lambda Calculus (STLC), it is desirable that it is a conservative extension of the STLC. Thus, λ_E should be able to simulate and express different aspects of the semantics of the STLC. We relate our calculus to the STLC here.

Figure 6 shows the definition of the STLC. Binding is also modelled using de Bruijn indices, and the dynamic semantics is a standard call-by-value small-step reduction. Since the reduction is *weak* (no reductions happen under abstractions), we can employ a simpler form of substitution for beta reduction [Forster and Smolka 2017]. The correspondence between STLC contexts and λ_E contexts is defined by the relation $\Gamma \sim \Gamma_\lambda$. Let $\vdash_\lambda e$ and $\vdash_\lambda A$ represent that e and A are expressions and types respectively in STLC. The connection with the typing of STLC are easy to establish:

THEOREM 4.22 (COMPLETENESS W.R.T. TYPING). *If $\Gamma_\lambda \vdash e : A$ and $\Gamma \sim \Gamma_\lambda$, then $\Gamma \vdash e : A$.*

THEOREM 4.23 (CONSERVATIVITY W.R.T. TYPING). *If $\vdash_\lambda e$, $\vdash_\lambda A$, $\Gamma \sim \Gamma_\lambda$, and $\Gamma \vdash e : A$, then $\Gamma_\lambda \vdash e : A$.*

Now we connect our semantics with the semantics of STLC. Since lambda abstractions are values in STLC, but not in λ_E , we need first to relate values in STLC and values in λ_E . Inspired by the unfolding operation in the work by Forster et al. [2020], we define a substitution function $f(v, e, k)$ that substitutes free variables in a λ_E expression e by their values in the environment v at depth k , and returns an STLC expression after the substitution. In this way, a closure in λ_E is transformed to a corresponding closed lambda abstraction. Note that this substitution function is not the same as the one in the reduction rule of STLC. The substitution in the beta reduction is an operation on STLC expressions and does not have environments involved.

Definition 4.24 (Substitution). The substitution function $f(v, e, k)$ is recursively defined as follows:

$$f(v, i, k) = i$$

$$f(v, \underline{n}, k) = \underline{n} \quad \text{if } n < k$$

$$f(v, \underline{n}, k) = i \quad \text{if } n \geq k \text{ and } \text{lookup}_v(v, n - k) = i$$

$$f(v, \underline{n}, k) = \lambda A. f(v', e, 1) \quad \text{if } n \geq k \text{ and } \text{lookup}_v(v, n - k) = \langle v', \lambda A. e \rangle$$

$$f(v, \lambda A. e, k) = \lambda A. f(v, e, k + 1)$$

$$f(v, e_1 e_2, k) = f(v, e_1, k) f(v, e_2, k)$$

Types	$A, B ::= \text{Int} \mid A \rightarrow B$		
Terms	$e ::= \underline{n} \mid i \mid e_1 e_2 \mid \lambda A. e$		
Values	$v_\lambda ::= i \mid \lambda A. e$		
Contexts	$\Gamma_\lambda ::= \cdot \mid \Gamma_\lambda, A$		
$\boxed{\Gamma_\lambda \vdash e : A}$	(Typing of STLC)		
STYP-LIT	STYP-VAR	STYP-APP	STYP-LAM
$\frac{}{\Gamma_\lambda \vdash i : \text{Int}}$	$\frac{\Gamma_\lambda[n] = A}{\Gamma_\lambda \vdash \underline{n} : A}$	$\frac{\Gamma_\lambda \vdash e_1 : A \rightarrow B \quad \Gamma_\lambda \vdash e_2 : A}{\Gamma_\lambda \vdash e_1 e_2 : B}$	$\frac{\Gamma_\lambda, A \vdash e : B}{\Gamma_\lambda \vdash \lambda A. e : A \rightarrow B}$
$\boxed{e \hookrightarrow e'}$	(Reduction of STLC)		
SRED-BETA	SRED-APPL	SRED-APPR	
$\frac{}{(\lambda A. e) v \hookrightarrow e[0 \mapsto v]}$	$\frac{e_1 \hookrightarrow e'_1}{e_1 e_2 \hookrightarrow e'_1 e_2}$	$\frac{e_2 \hookrightarrow e'_2}{v_1 e_2 \hookrightarrow v_1 e'_2}$	
$\boxed{\Gamma \sim \Gamma_\lambda}$	(Context Correspondence)		
CREL-TOP	CREL-AND		
$\frac{}{\varepsilon \sim \cdot}$	$\frac{A \sim \Gamma_\lambda}{A \& B \sim \Gamma_\lambda, B}$		
$\boxed{v_\lambda \sim v}$	(Value Correspondence)		
VREL-LIT	VREL-LAM		
$\frac{}{i \sim i}$	$\frac{f(v, e', 1) = e}{\lambda A. e \sim \langle v, \lambda A. e' \rangle}$		

Fig. 6. Simply Typed Lambda Calculus and correspondence to λ_E .

With the help of the substitution function, now we can define value correspondence, as shown in Figure 6. Any integer corresponds to itself. An STLC lambda abstraction $\lambda A. e$ corresponds to a λ_E closure $\langle v, \lambda A. e' \rangle$ such that substituting e' with environment v at depth 1 gives the function body e . We prove that the dynamic semantics of λ_E is complete and conservative w.r.t. the one of STLC.

THEOREM 4.25 (COMPLETENESS W.R.T. DYNAMIC). *If $\vdash e : A$ and $e \hookrightarrow^* v_\lambda$, then there exists v s.t. $\varepsilon \vdash e \hookrightarrow^* v$ and $v_\lambda \sim v$.*

THEOREM 4.26 (CONSERVATIVITY W.R.T. DYNAMIC). *If $\vdash_\lambda e$, $\varepsilon \vdash e : A$ and $\varepsilon \vdash e \hookrightarrow^* v$, then there exists v_λ s.t. $e \hookrightarrow^* v_\lambda$ and $v_\lambda \sim v$.*

An important point to note is that the conservativity proof to the STLC is the only place in our proofs where we need to reason about substitution. This is, of course unavoidable, since the standard formulation of the STLC employs substitution in beta-reduction.

4.5 Closure Calculus $\lambda\hat{\rho}$

We also connect the λ_E calculus with the CBV version of the closure calculus $\lambda\hat{\rho}$ [Biernacka and Danvy 2007] shown in Figure 7. Since $\lambda\hat{\rho}$ is an untyped calculus, we only show a connection with respect to the dynamic semantics. In $\lambda\hat{\rho}$, terms consist of de Bruijn indices, applications and lambda abstractions. A closure is either a lambda equipped with a substitution, or an application of two closures. Values include function closures, and substitutions are lists of closures.

Unlike conventional lambda calculi, the reduction in $\lambda\hat{\rho}$ is defined for closures instead of lambda terms. Applications of closures are evaluated from left to right. Moreover, closures are distributive over application of lambda terms, and environment duplicates for each term in the application

834	Terms	$e ::= \underline{n} \mid e_1 e_2 \mid \lambda.e$
835	Substitutions	$s ::= \emptyset \mid v \cdot s$
836	Closures	$c ::= e[s] \mid c_1 c_2$
837	Values	$v ::= (\lambda.e)[s]$
838		
839		
840	$\boxed{c \hookrightarrow c'}$	(Reduction of $\lambda\hat{\rho}$)
841	RED-VAR	RED-BETA
842	$\frac{}{\underline{n}[v_1 \cdots v_n \cdots v_m] \hookrightarrow v_n}$	$\frac{}{((\lambda.e)[s]) v \hookrightarrow e[v \cdot s]}$
843		RED-APPL
844		$\frac{c_1 \hookrightarrow c'_1}{c_1 c_2 \hookrightarrow c'_1 c_2}$
845		RED-APPR
846		$\frac{c_2 \hookrightarrow c'_2}{v_1 c_2 \hookrightarrow v_1 c'_2}$
847		RED-APPDIST
848		$\frac{}{(e_1 e_2)[s] \hookrightarrow (e_1[s]) (e_2[s])}$

Fig. 7. The $\lambda\hat{\rho}$ Calculus

(rule **RED-APPDIST**). Note that this is different from how we evaluate boxes in λ_E . If there is an application $e_1 e_2$ boxed by an environment v , then v becomes the global environment for evaluating $e_1 e_2$, and no duplication of environments is performed for evaluating boxed expressions.

We show that the semantics of the λ_E calculus can encode the one of $\lambda\hat{\rho}$ by first translating the substitutions in $\lambda\hat{\rho}$ to the environments in λ_E :

$$\begin{aligned} \llbracket \emptyset \rrbracket_s &= \varepsilon \\ \llbracket v \cdot s \rrbracket_s &= \llbracket s \rrbracket_s \circ \llbracket v \rrbracket_c \end{aligned}$$

Here a list of values is translated to a merge of corresponding values in λ_E , and the translation of closures $\llbracket c \rrbracket_c$ is defined as follows:

$$\begin{aligned} \llbracket c_1 c_2 \rrbracket_c &= \llbracket c_1 \rrbracket_c \llbracket c_2 \rrbracket_c \\ \llbracket (\lambda.e)[s] \rrbracket_c &= \langle \llbracket s \rrbracket_s, \lambda\varepsilon.e \rangle \\ \llbracket e[s] \rrbracket_c &= \llbracket s \rrbracket_s \triangleright e, \text{ otherwise} \end{aligned}$$

Applications of closures are translated to applications of two expressions. A function closure is translated to a closure in λ_E , in which an untyped lambda abstraction in $\lambda\hat{\rho}$ is encoded as a lambda abstraction with a dummy unit type in λ_E . For any other closures in $\lambda\hat{\rho}$, they are encoded as boxes. With the translation defined, we can prove that the λ_E is semantically complete to $\lambda\hat{\rho}$.

THEOREM 4.27 (COMPLETENESS). *If $c \hookrightarrow^* v$ in $\lambda\hat{\rho}$, then $\varepsilon \vdash \llbracket c \rrbracket_c \hookrightarrow^* \llbracket v \rrbracket_c$.*

5 COMPILATION TO AN ABSTRACT MACHINE

We compile our calculus to a modern SECD abstract machine [Leroy and Grall 2009]. The syntax for the abstract machine is:

873	Instructions	$I ::= \text{Lit}(i) \mid \text{Unit} \mid \text{Clos}(C) \mid \text{Ret} \mid \text{App} \mid \text{Query} \mid \text{Proj}(n) \mid \text{Box}(C) \mid \text{Merge} \mid \text{Trans}$
874		$\mid \text{Del} \mid \text{Rec}(l) \mid \text{Sel}(l)$
875	Values	$V ::= i \mid \varepsilon \mid C[V] \mid V_1 \circ V_2 \mid \{l = V\}$
876	Code	$C ::= \bullet \mid I, C$
877	Stacks	$S ::= \bullet \mid S, V \mid S, (C, V)$

Code is represented as a list of instructions, and instructions are tailored to accommodate the operations required for implementing the constructs of the λ_E calculus. The form of values corresponds

883		$\llbracket i \rrbracket_e = \text{Lit}(i)$
884		$\llbracket \epsilon \rrbracket_e = \text{Unit}$
885	$\langle \text{Lit}(i), C; S; V \rangle \mapsto \langle C; S, i; V \rangle$	$\llbracket ? \rrbracket_e = \text{Query}$
886	$\langle \text{Unit}, C; S; V \rangle \mapsto \langle C; S, \epsilon; V \rangle$	$\llbracket \lambda A. e \rrbracket_e = \text{Clos}(\llbracket e \rrbracket_e, \text{Ret})$
887	$\langle \text{Clos}(C'), C; S; V \rangle \mapsto \langle C; S, C'[V]; V \rangle$	$\llbracket e_1 e_2 \rrbracket_e = \llbracket e_1 \rrbracket_e, \llbracket e_2 \rrbracket_e, \text{App}$
888	$\langle \text{App}, C; S, C'[V_2], V_1; V \rangle \mapsto \langle C'; S, (C, V); V_2 \circ V_1 \rangle$	$\llbracket e_1 \circ e_2 \rrbracket_e = \llbracket e_1 \rrbracket_e, \text{Trans}, \llbracket e_2 \rrbracket_e, \text{Del}, \text{Merge}$
889	$\langle \text{Ret}, C; S, (C', V_2), V_1; V \rangle \mapsto \langle C'; S, V_1; V_2 \rangle$	$\llbracket e_1 \triangleright e_2 \rrbracket_e = \llbracket e_1 \rrbracket_e, \text{Box}(\llbracket e_2 \rrbracket_e, \text{Ret})$
890	$\langle \text{Query}, C; S; V \rangle \mapsto \langle C; S, V; V \rangle$	$\llbracket e.n \rrbracket_e = \llbracket e \rrbracket_e, \text{Proj}(n)$
891	$\langle \text{Proj}(n), C; S, V'; V \rangle \mapsto \langle C; S, V'.n; V \rangle$	$\llbracket \{l = e\} \rrbracket_e = \llbracket e \rrbracket_e, \text{Rec}(l)$
892	$\langle \text{Box}(C'), C; S, V'; V \rangle \mapsto \langle C'; S, (C, V); V' \rangle$	$\llbracket e.l \rrbracket_e = \llbracket e \rrbracket_e, \text{Sel}(l)$
893	$\langle \text{Merge}, C; S, V_1, V_2; V \rangle \mapsto \langle C; S, (V_1 \circ V_2); V \rangle$	
894	$\langle \text{Trans}, C; S, V'; V \rangle \mapsto \langle C; S, V'; V \circ V' \rangle$	$\llbracket i \rrbracket_v = i$
895	$\langle \text{Del}, C; S; V \circ V' \rangle \mapsto \langle C; S; V \rangle$	$\llbracket \epsilon \rrbracket_v = \epsilon$
896	$\langle \text{Rec}(l), C; S, V'; V \rangle \mapsto \langle C; S, \{l = V'\}; V \rangle$	$\llbracket \langle v, \lambda A. e \rangle \rrbracket_v = (\llbracket e \rrbracket_e, \text{Ret})[\llbracket v \rrbracket_v]$
897	$\langle \text{Sel}(l), C; S, V'; V \rangle \mapsto \langle C; S, V'.l; V \rangle$	$\llbracket v_1 \circ v_2 \rrbracket_v = \llbracket v_1 \rrbracket_v \circ \llbracket v_2 \rrbracket_v$
898		$\llbracket \{l = v\} \rrbracket_v = \{l = \llbracket v \rrbracket_v\}$
899		
900		
901		
902		
903		

Fig. 8. Transition of the Abstract Machine and Compilation from λ_E .

to the one in the λ_E calculus where $C[V]$ denotes a function closure in the machine. Like in the calculus, an environment of the machine is simply a value and we do not need to introduce a separate definition for environments. A stack consists of values and return frames.

5.1 Transition Relation

The machine state is defined as a triple consisting of code, a stack and an environment. The transition relation $\langle C; S; V \rangle \mapsto \langle C'; S'; V' \rangle$, shown in Figure 8, defines the behaviour of the machine. For example, $\text{Clos}(C)$ creates a closure for code C and Ret returns to the calling function, while App facilitates the execution of a function application. Query reifies the environment by pushing the current environment V to the stack. Proj returns the n -th values $V'.n$ of the top value V' of the stack. $\text{Rec}(l)$ and $\text{Sel}(l)$ return a single-field record and the selection for a label respectively. Moreover, Merge returns a merge of the top two values of the stack. With Trans and Del , we are able to implement dependent merges. For a dependent merge $e_1 \circ e_2$, we first compute the code translated from e_1 , and this results that a machine value V_1 will appear at the top of the stack. Then Trans appends V_1 to the environment for computing the code corresponding to e_2 . After that, Del deletes the V_1 from the environment for avoiding that V_1 affects future computation. Finally, Merge gives the result by merging V_1 and V_2 . The instruction that corresponds to the box construct in λ_E is Box . Basically, $\text{Box}(C')$ imports an environment for code C' . The top value V' from the stack is assigned as the environment to compute C' . After the computation of C' , we need to continue to compute code C with environment V . Thus, a return frame (C, V) is pushed into the stack in the transition rule for Box .

We define \mapsto^+ as the transitive closure of \mapsto . That is, there is one or more transitions along the computation. To establish the connection with the machine, we need to define the compilation from λ_E to the abstract machine (Figure 8). The compilation function $\llbracket e \rrbracket_e$ compiles each λ_E expression to machine code that corresponds to the construct. We also need to connect the values from the source language to the machine values. $\llbracket v \rrbracket_v$ compiles a λ_E value to a corresponding machine value. Note that for a closure $\langle v, \lambda A. e \rangle$ in λ_E , the environment v is also needed to be compiled.

We can prove the correctness of the compilation using our big-step semantics. That is, the compilation preserves program behaviour. If a λ_E expression e is evaluated to a value v' , then the machine reaches a final state, in which the corresponding machine code $\llbracket e \rrbracket_e$ is consumed and the corresponding machine value $\llbracket v' \rrbracket_v$ is left on top of the stack.

THEOREM 5.1 (SEMANTIC PRESERVATION). *If $v \vdash e \Rightarrow v'$ then for any C and S , $\langle (\llbracket e \rrbracket_e, C); S; \llbracket v \rrbracket_v \rangle \mapsto^+ \langle C; (S, \llbracket v' \rrbracket_v); \llbracket v \rrbracket_v \rangle$.*

6 DISCUSSION, VARIANTS AND EXTENSIONS

We have only studied simply typed calculi. With more advanced calculi and type systems there can be additional complications. Nonetheless, many programming language features are still quite easy to add to calculi employing our proposed environment-based semantics. To illustrate this point we extended λ_E with references. We also discuss some challenges that we foresee with more advanced type systems, and some alternative designs may help with some challenges.

Extension with references. To investigate the addition of features to λ_E , we created an extension with references. We extend the type system and small-step semantics with reference-related constructs and rules. The extended calculus is deterministic and type sound. We also define a big-step semantics and prove that it is equivalent to the small-step semantics. Moreover, we extend the abstract machine in Section 5 and compile the extended calculus to the extended abstract machine. The definition of the extended calculus, as well as the mechanized formalization and proofs in Coq, can be found in the supplementary materials.

An alternative design: No first-class environments. In the design of λ_E we opted to include constructs for first-class environments, since they do not introduce significant complexity. However, we can easily remove first-class environments, while preserving our environment-based semantics and making the calculus closer to the lambda calculus. Firstly, we can remove the constructs (and associated rules) highlighted with red for lookups by label. Those constructs are not needed for modelling a conventional lambda calculus. Secondly, we can simplify the box construct highlighted with yellow, such that the environment in the box is always a value:

$$\frac{\varepsilon \vdash v : \Gamma_1 \quad \Gamma_1 \vdash e : A}{\Gamma \vdash v \triangleright e : A} \text{ TYP-BOX}$$

In this way, we can remove $[] \triangleright e$ from frames since the environment does not need to be evaluated in the box. Finally, we can remove the blue constructs for queries and index lookups. Instead, we define directly de Bruijn indexes \underline{n} , whose static and dynamic behaviour are the same as $? . n$:

$$\frac{}{\Gamma \vdash \underline{n} : \text{lookup}(\Gamma, n)} \text{ TYP-VAR} \quad \frac{}{v \vdash \underline{n} \hookrightarrow \text{lookup}_v(v, n)} \text{ STEP-VAR}$$

After these steps, and removing corresponding rules, we have a simple calculus with closures. The simplified calculus may be interesting for language designers, which are not interested in having first-class environments. Furthermore, in some settings, first-class environments themselves may introduce complexity, which would not appear without first-class environments. In this case a closure calculus may be simpler to design and formalize.

More powerful calculi and type systems. We foresee some challenges when extending our ideas to more powerful calculi. For example, in System F, substitution is performed on types. In the type system, *substitution on types* is needed for the output type of type applications:

$$\frac{\Gamma \vdash e : \forall X. B}{\Gamma \vdash e A : B[A/X]} \text{ TYP-TAPP}$$

and *substitution of types in terms* $e[A/X]$ is applied on beta-reduction of type applications:

$$\frac{(\lambda X. e) A \hookrightarrow e[A/X]}{} \text{ STEP-TAPP}$$

in the dynamic semantics. We expect to be able to extend the idea of environment-based reduction and delaying substitutions to the dynamic semantics: pushing A into the environment rather than substituting A for the type variables in e . That is, firstly $\Lambda X. e$ would reduce to a closure $\langle v, \Lambda X. e \rangle$, and then $\langle v, \Lambda X. e \rangle A$ is beta-reduced to $(v \text{ , } X = A) \triangleright e$ (for readability, here we use a named presentation for variables in the environment).

It is less clear how to deal with the type-level substitution in the typing rule for type applications. If we are to follow the spirit of an environment-based semantics, then we ought to replace the type level substitution with a closure. This hints for a reformulation of the type syntax to include closures. However, how to redesign the type syntax of System F and its type system with this closure formulation is non-obvious. Another alternative is to keep the substitution at the type level, and still benefit from the environment-semantics for terms. However, some complications would be introduced in this setting. For example, the reasoning would require a form of type correspondence that connects types with respect to their environments, since substitution is performed in the typing rule but type references are not resolved in the reduction rule. We expect that the work by [Amin and Rompf \[2017\]](#) will be helpful to pursue this direction, since they present a definitional (big-step style) interpreter for System $F_{<}$, which does not employ substitution. Nevertheless, it is clear that extending the environment semantics to System F requires further study.

While many programming languages and calculi rely only on weak forms of reduction, such as call-by-value reduction, some calculi also employ full reduction. For example, dependent types are a powerful tool adopted by proof assistants such as Coq. Typically, in a dependently typed language, determining the equality of types is necessary and is captured as the conversion rule. The definitional equality of two dependent types is implemented by evaluating the terms on which the types rely to normal form and subsequently comparing these normalized terms. For example, to check $\text{Vec } (1 + 1)$ is equal to $\text{Vec } 2$ we need to reduce $1 + 1$ to 2. Dealing with equality requires reducing sub-terms at any position including those inside lambdas. However we have not studied full reduction in our work. For future work, we want to explore the design of environment-based semantics with full reduction for first-class environments, while expecting to simplify the complicated reasoning of (explicit) substitutions in previous work [[Nishizaki 1994](#); [Sato et al. 2001](#)].

7 RELATED WORK

First-class environments. First-class environments were first introduced by [Gelernter et al. \[1987\]](#) in the research literature. They proposed a programming language called Symmetric Lisp that enriches Lisp with first-class environments that can be manipulated at runtime by users. Following the work by [Gelernter et al.](#), there have been several attempts to incorporate first-class environments into dialects of Lisp. For example, [Queinnec and Roue \[1996\]](#) presented a form of first-class environments as an approach to share data objects for the Scheme programming language.

There is also some work on statically typed calculi with first-class environments. [Nishizaki \[1994\]](#) proposed $\lambda_{\text{env}}^{\rightarrow}$ with reification and reflection on environments. In our work, the two corresponding operators are the query and box constructs in λ_E . $\lambda_{\text{env}}^{\rightarrow}$ is a non-deterministic calculus defined from the weak reduction of the $\lambda\sigma$ calculus [[Curien et al. 1996](#)]. $\lambda_{\text{env}}^{\rightarrow}$ is proved to have confluence and normalization. Normalization is proved indirectly by translating the calculus to another calculus with records. The issue of defining the set of free variables is not covered in $\lambda_{\text{env}}^{\rightarrow}$.

[Sato et al. \[2001\]](#) identified the importance of the correct definition of free variables in the presence of first-class environments. They proposed the $\lambda\epsilon$ calculus and addressed the issue of variables and bindings. The $\lambda\epsilon$ calculus is conservative over the lambda calculus with full reduction. Both $\lambda_{\text{env}}^{\rightarrow}$ and $\lambda\epsilon$ use a named representation for variables, while de Bruijn indices are applied in λ_E . However, record-like environments and label-based lookups are still possible in λ_E . The proof

	$\lambda\hat{p}$ 2007	$\lambda_{\text{env}}^{\rightarrow}$ 1994	$\lambda\varepsilon$ 2001	λ_c 2008	E_i 2023	λ_E present work
First-class Environments	×	✓	✓	✓	✓	✓
Reification of Environments	×	✓	×	×	✓	✓
Conservativity over lambda calculus	✓	×	✓			✓
Termination		✓	✓			✓
Reduction Strategy	Weak	Weak	Full	CBV	CBV	CBV
Big-step Semantics						✓
Compilation to an Abstract Machine	✓					✓
Extended with References				✓		✓
Mechanized Proofs	×	×	×	×	✓	✓

Table 1. A comparison between closure calculi and calculi with first-class environments. A \times symbol denotes a negative result (the property or feature does not hold). A \checkmark denotes a positive result. White-space denotes that the property/feature has not been studied or it is unknown.

of normalization in $\lambda_{\text{env}}^{\rightarrow}$ and $\lambda\varepsilon$ requires careful reasoning about names and (explicit) substitutions. For example, the proof in $\lambda\varepsilon$ requires some sophisticated techniques, such as decoration trees, for tracking the movements of the environments. As discussed by [Sato et al.](#), subtle counter-examples occur if the reduction rule for nested evaluation is not defined properly. In contrast, normalization is easy to prove with the environment-based semantics in λ_E . [Dezani-Ciancaglini et al. \[2008\]](#) also proposed a typed calculus with first-class environments, which we call λ_c . The calculus focuses on modelling context-dependent behaviour of objects and employs call-by-value reduction. However, termination and the correspondence to the lambda calculus are unknown.

None of the work above is mechanically formalized in theorem provers. The E_i calculus proposed by [Tan and Oliveira](#) is a calculus with environment-based semantics. Similarly to $\lambda\varepsilon$ and $\lambda_{\text{env}}^{\rightarrow}$, first-class environments are represented as records. Conventional variables and lambda abstractions, however, cannot be expressed in E_i . Thus conservativity over the lambda calculus is unknown, and termination is not investigated. Fetching values from (global or local) contexts is done by the query construct and label projections. However, the access of a value from the environment is indirect: an annotated multi-field record needs to be cast to a single-field record before the lookup by label. In contrast, the selection in λ_E is direct since no casting is triggered.

Closure calculi. The calculus of closures $\lambda\rho$ [[Curien 1991](#)] is a calculus that aims to maintain faithfulness to the substitution-based lambda calculus, while better reflecting the computational aspects of abstract machines. Explicit substitutions [[Abadi et al. 1991](#)] extend the idea of the closure calculus where reduction under lambdas is allowed in general. As a result, if de Bruijn notation is used for encoding variables, shifting needs to be defined and an amount of reasoning of shifting is involved in the proof of metatheory. In contrast, $\lambda\rho$ targets weak lambda calculus and there is no reduction under lambdas. Hence there is no need to be concerned about name clashes, and α -conversion does not need to be performed at any point. Thus de Bruijn indices fit very well with $\lambda\rho$. Our calculus λ_E generalizes this idea with first-class environments. For example, while all substitutions are at the top level in $\lambda\rho$, nested environments, such as $(e_1 \triangleright e_2) \triangleright e_3$, are allowed in λ_E . Moreover, a global environment is also applied for evaluating environments with open terms.

The $\lambda\hat{p}$ calculus [[Biernacka and Danvy 2007](#)] is a minimal extension of $\lambda\rho$, where one-step reductions can be expressed. In this way, call-by-name and call-by-value strategies are investigated in $\lambda\hat{p}$. Following the refocusing method [[Danvy and Nielsen 2004](#)], several environment machines can be derived for corresponding reduction strategies. In our work, the calculus is compiled to an abstract machine using a big-step semantics. [Jay \[2019\]](#) also proposes a closure calculus in a formulation different from [Curien](#). The calculus requires no metatheory for substitution and can

be translated to a calculus of combinators. However, the connection with the lambda calculus or abstract machines is unknown. Table 1 provides an overview of the key results in various closure calculi and calculi with first-class environments.

Type Soundness for Definitional Interpreters. Siek [2013] proposed a simple type soundness proof for the STLC, based on a (big-step style) definitional interpreter, inspired by an approach to prove type soundness by Ernst et al. [2006]. Amin and Rompf [2017] extended Siek’s approach to System $F_{<}$ and variants of the Dependent Object Calculus (DOT) [Rompf and Amin 2016]. In their approach, definitional interpreters, which are big-step and substitution free, are parametrized by a step-counter and errors are dealt with monads. The use of the later two techniques addresses well-known limitations of a big-step style for type soundness proofs. In particular in terms of distinguishing errors from non-termination. Furthermore, the use of step-counters avoids the need for coinduction, and the proofs can remain inductive. They showed that their approach extends to common features, such as references or exceptions. The environment semantics that we employ is small-step (although we also present a big-step version), so the complications due to the big-step style do not arise in our setting. Thus we do not need to employ step-counters or monads, which add some additional complexity to proofs and reasoning. Furthermore, we present a calculus with first-class environments, and we study a broader set of results besides type soundness proofs. Unlike us, their work covers more advanced type systems. As mentioned in Section 6 we believe that their work will be helpful when extending our results to polymorphism.

Binding in mechanized proofs. While paper proofs often overlook the complexities of binders, mechanical proofs that involve binders tend to be more intricate. To measure progress of mechanizing proofs, the POPLmark challenge [Aydemir et al. 2005] and POPLMark Reloaded [Abel et al. 2019] have introduced a collection of benchmarks about the metatheory of programming languages. Many tools have been developed to simplify the process of defining infrastructure and enhance proof automation. For example, Autosubst [Schäfer et al. 2015; Stark et al. 2019] is a Coq library that provides automation for de Bruijn syntax and substitution based on the $\lambda\sigma$ calculus [Abadi et al. 1991]. Besides de Bruijn indices, there are other binding representations, including higher order abstract syntax (HOAS) [Pfenning and Elliott 1988] that relies binders in the meta-language to model binders in the object language, and locally nameless representation [Charguéraud 2012] which uses de Bruijn indices to represent bound variables but applied names to represent free variables. In our work we suggest that, by adopting an environment-based approach in formalizations, many common issues with binding and substitution can essentially be avoided. However, more study is needed to cover more advanced type systems with an environment-based semantics.

8 CONCLUSION

This paper presents a case for the formalization of programming languages by adopting an environment-based semantics. An environment-based semantics provides substantial advantages over the traditional substitution model. The proposed call-by-value statically typed calculus, λ_E , aligns closely with practical execution models and simplifies reasoning, thereby facilitating more efficient and manageable formalizations with proof assistants. By elevating environments to first-class citizens within the language, λ_E enables powerful and flexible abstractions, promoting the development of sophisticated and modular programs. Furthermore, λ_E has several desirable properties, such as determinism, type soundness, and normalization, while also establishing operational correspondence with other models. The extension of λ_E with references and the Coq formalization of all calculi and proofs further underscores the potential of environment-based semantics as a viable alternative to the substitution model in programming languages.

REFERENCES

- Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416. <https://doi.org/10.1017/S0956796800000186>
- Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19.
- Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, 666–679.
- Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3603)*, Joe Hurd and Thomas F. Melham (Eds.). Springer, 50–65. https://doi.org/10.1007/11541868_4
- Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.
- Malgorzata Biernacka and Olivier Danvy. 2007. A concrete framework for environment machines. *ACM Trans. Comput. Log.* 9, 1 (2007), 6. <https://doi.org/10.1145/1297658.1297664>
- Luca Cardelli. 1997. Program Fragments, Linking, and Modularization. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 266–277. <https://doi.org/10.1145/263699.263735>
- Arthur Charguéraud. 2012. The locally nameless representation. *Journal of automated reasoning* 49 (2012), 363–408.
- Alonzo Church. 1941. *The calculi of lambda-conversion*. Princeton University Press.
- Pierre Crégut. 1990. An Abstract Machine for Lambda-Terms Normalization. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 333–340. <https://doi.org/10.1145/91556.91681>
- Douglas Crockford. 2008. ADsafe. <http://www.adsafe.org>
- Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. 1996. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *J. ACM* 43, 2 (1996), 362–397. <https://doi.org/10.1145/226643.226675>
- P-L Curien. 1991. An abstract framework for environment machines. *Theoretical Computer Science* 82, 2 (1991), 389–402.
- Olivier Danvy and Lasse R Nielsen. 2004. Refocusing in reduction semantics. *BRICS Report Series* 11, 26 (2004).
- Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.
- Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. 2008. A Calculus of Evolving Objects. *Sci. Ann. Comput. Sci.* 18 (2008), 63–98. <http://www.info.uai.ro/bin/Annals/Article?v=XVIII&a=3>
- Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, 270–282.
- Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ -calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Eberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.
- Yannick Forster, Fabian Kunze, and Marc Roth. 2020. The weak call-by-value λ -calculus is reasonable for both time and space. *Proc. ACM Program. Lang.* 4, POPL (2020), 27:1–27:23. <https://doi.org/10.1145/3371095>
- Yannick Forster and Gert Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10499)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer, 189–206. https://doi.org/10.1007/978-3-319-66107-0_13
- David Gelernter, Suresh Jagannathan, and Thomas London. 1987. Environments as First Class Objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 98–110. <https://doi.org/10.1145/41625.41634>
- Suresh Jagannathan. 1994. Dynamic Modules in Higher-Order Languages. In *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, Henri E. Bal (Ed.). IEEE Computer Society, 74–87. <https://doi.org/10.1109/ICCL.1994.288391>
- Barry Jay. 2019. A simpler lambda calculus. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 1–9. <https://doi.org/10.1145/3294032.3294085>
- Gilles Kahn. 1987. Natural semantics. In *Annual symposium on theoretical aspects of computer science*. Springer, 22–39.

- P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. <https://doi.org/10.1093/COMJNL/6.4.308>
- Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Inf. Comput.* 207, 2 (2009), 284–304. <https://doi.org/10.1016/J.IC.2007.12.004>
- Adrian Mettler, David A. Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society. <https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java>
- Mark Miller. 2006. *Robust composition: Towards a unified approach to access control and concurrency control*. Johns Hopkins University.
- Mark S. Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-Based Financial Instruments. In *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1962)*, Yair Frankel (Ed.). Springer, 349–378. https://doi.org/10.1007/3-540-45472-1_24
- Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. Google, Inc., Tech. Rep (2008).
- Shin-ya Nishizaki. 1994. Simply Typed Lambda Calculus with First-Class Environments. *Publications of the Research Institute for Mathematical Sciences* 30, 6 (1994), 1055–1121.
- Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA, June 22-24, 1988, Richard L. Wexelblat (Ed.). ACM, 199–208. <https://doi.org/10.1145/53990.54010>
- Christian Queinnec and David De Roure. 1996. Sharing Code through First-class Environments. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 251–261. <https://doi.org/10.1145/232627.232653>
- John C Reynolds. 1997. Design of the programming language Forsythe. In *ALGOL-like languages*. Birkhauser Boston Inc., 173–233.
- Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (Amsterdam, Netherlands) (OOPSLA 2016)*. 624–641.
- Masahiko Sato, Takafumi Sakurai, and Rod M. Burstall. 2001. Explicit Environments. *Fundam. Informaticae* 45, 1-2 (2001), 79–115. <http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-05>
- Masahiko Sato, Takafumi Sakurai, and Yuki Yoshi Kameyama. 2002. A Simply Typed Context Calculus with First-class Environments. *J. Funct. Log. Program.* 2002 (2002). <http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-04.pdf>
- Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24
- Jeremy Siek. 2013. Type safety in three easy lemmas. <http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html>
- Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 166–180. <https://doi.org/10.1145/3293880.3294101>
- William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. <https://doi.org/10.2307/2271658>
- Jinhao Tan and Bruno C d S Oliveira. 2023. Dependent Merges and First-Class Environments. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94.