Dependent Merges and First-Class Environments

Jinhao Tan
The University of Hong Kong, China

Bruno C. d. S. Oliveira
The University of Hong Kong, China

Abstract

In most programming languages a (runtime) *environment* stores all the definitions that are available to programmers. Typically, environments are a meta-level notion, used only conceptually or internally in the implementation of programming languages. Only a few programming languages allow environments to be *first-class values*, which can be manipulated directly in programs. Although there is some research on calculi with first-class environments for statically typed programming languages, these calculi typically have significant restrictions.

In this paper we propose a statically typed calculus, called E_i, with first-class environments. The main novelty of the E_i calculus is its support for first-class environments, together with an expressive set of operators that manipulate them. Such operators include: *reification* of the current environment, environment *concatenation*, environment *restriction*, and *reflection* mechanisms for running computations under a given environment. In E_i any type can act as a context (i.e. an environment type) and contexts are simply types. Furthermore, because E_i supports subtyping, there is a natural notion of context subtyping. There are two important ideas in E_i that generalize and are inspired by existing notions in the literature. The E_i calculus borrows *disjoint intersection types* and a *merge operator*, used in E_i to model contexts and environments, from the λ_i calculus. However, unlike the merges in λ_i, the merges in E_i can depend on previous components of a merge. From implicit calculi, the E_i calculus borrows the notion of a *query*, which allows *type-based* lookups on environments. In particular, queries are key to the ability of E_i to reify the current environment, or some parts of it. We prove the determinism and type soundness of E_i, and show that E_i can encode all well-typed λ_i programs.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases First-class Environments, Disjointness, Intersection Types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.34

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.9.2.2

Funding Hong Kong Research Grant Council projects number 17209520 and 17209821 sponsored this work.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

In most programming languages, (runtime) *environments* are used to store all the available definitions at a given point in a program. Typically, an environment is a dictionary that maps variable names to values. However, environments are normally a meta-level concept, which does not have any syntactic representation in source programs. Environments may be used internally in the implementation of programming languages. For example, in implementing functional languages, closures are often used to keep the lexical environment of a function around. However, it is impossible for programmers to write directly a closure or manipulate environments explicitly.

First-class environments [18, 23, 24, 29, 36], are environments that can be created, composed, and manipulated at runtime. In programming languages with first-class environments, programs have an explicit syntactic representation for environments that enables them to be first-class values. As argued by Gelernter et al. [18], with first-class environments, the distinction between declarations and expressions can be eliminated. Furthermore many programming language constructs – including *closures*, *modules*, *records* and *object-oriented constructs* – can be modelled with first-class environments. However, only a few programming languages allow environments to be first-class values. These languages are mainly dynamically typed languages such as dialects of Lisp [18] and the R language [17]. Typically, operations on environments include: reification (transforming environments into data objects), reflection (treating data objects as environments), environment restriction (returning part of an environment), and environment composition/concatenation. While dynamically typed languages with first-class environments give users high flexibility in manipulating environments, several runtime type errors are unavoidable due to the absence of static typing.

Compared with work on dynamically typed languages, there is much less research on statically typed languages with first-class environments [44, 45, 47]. In these works, environments as first-class values have a special kind of type which is called an *environment type*. Although static typing prevents some of the runtime errors, subtyping is not included in existing type systems with environment types. At the term level, there are two constructs for environments: one is an evaluation operation $e[\![a]\!]$ that evaluates the expression a under an environment e, and the other is an operator returning the current environment. While these two constructs model importing and exporting of environments respectively, there are no facilities for concatenation or restriction of environments in these calculi.

In this paper we propose a statically typed calculus, called E_i, with first-class environments. The main novelty of the E_i calculus is its support for first-class environments with an expressive set of operators that manipulate first-class environments. In E_i, both reflection mechanisms for running computations under a given environment and reification of the current environment are supported. Moreover, compared with previous work on typed calculi with first-class environments, environment *concatenation* and environment *restriction* are allowed. In E_i, types and contexts (i.e. environment types) are completely unified. That is, any type can act as a context and contexts are simply types. Unlike previous calculi, E_i also supports subtyping and has a natural notion of subtyping of environments. In E_i, users can benefit from static typing for handling type errors at compile-time, while still having various flexible mechanisms to manipulate environments.

In order to model environments, the E_i calculus borrows *disjoint intersection types* and the *merge operator* from the λ_i calculus [32]. The novelty in E_i is to additionally use intersection types to model environment types (or contexts), and disjointness to model disjointness/uniqueness of variables in an environment. Correspondingly, in E_i, the merge operator enables constructing and concatenating environments. Moreover, unlike λ_i, the merges in E_i can depend on previous components of a merge. In other words, merges in E_i are *dependent* (note that the dependency in a merge is term-level dependency, and it should not be confused with dependent types). Unifying contexts and types enables type information flowing from the left branch to the right branch in a merge, such that the type of the left branch becomes part of the context of the right branch. Consequently, with reification, the right branch of a merge can construct an expression based on the left branch of a merge. For example, the following program (with syntactic sugar)

$$\{x = 1\} \fatsemi \{y = x\}$$

is well-typed in E_i. Here y in the right branch can access x and build a value under the environment $\{x = 1\}$. The merge will be evaluated to $\{x = 1\} \fatsemi \{y = 1\}$. Dependent merges are useful for modelling dependent declarations, which are not expressible in λ_i since a field in a single record cannot access the field in a previous record in a merge.

Instead of looking up values by names as in traditional lambda calculi, the E_i calculus borrows the notion of a *query*, which enables *type-based* lookups on environments, from *implicit calculi* [12,31,46]. In implicit calculi, queries are used to query *implicit* environments by type. However, in E_i, queries are applied directly to runtime environments instead, and they are key to the ability of E_i to reify the current environment, or some parts of it. Effectively, a query can synthesize the current context (in typing) and the current environment (during reduction). With type annotations, queries can choose part of the environment based on those annotations, modelling environment restriction.

In our work, we prove the determinism and type soundness of E_i, and show that E_i can encode all λ_i programs. The E_i calculus and all the proofs presented in this paper have been formalized in the Coq theorem prover [9]. In summary, the contributions of this paper are:

- **Dependent merges as first-class environments:** We propose the novel notion of dependent merges, which allow dependencies appearing in merges. With dependent merges, dependent declarations and first-class environments can be modelled easily in a natural way.
- **The E_i calculus:** We present a statically typed calculus called E_i with support for creation, reification, reflection, concatenation and restriction of first-class environments. In addition, we study an extension with fixpoints (shown in the appendix). Both calculi are deterministic and type sound.
- **Encoding of the λ_i calculus:** We show that E_i can encode the type system of the λ_i [32] via a type-directed translation. In other words, standard variables, lambda abstractions, and non-dependent merges can be fully encoded in E_i.
- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover and they are available in the artifact associated to this paper: `https://github.com/tjhao/ecoop2023`

2 Overview

This section gives an overview of our work. We start with some background on the merge operator, first-class environments and program fragments. Then we discuss challenges of modelling first-class environments as merges and finally we discuss the key ideas in our work.

2.1 Background

The merge operator and disjoint intersection types. The original non-dependent merge operator (denoted here by $\,,,\,$) was firstly introduced by Reynolds [38] and later refined by Dunfield [15]. Merges add expressiveness to terms, enabling constructing values that inhabit intersection types. Essentially, with the merge operator, values are allowed to have multiple types. For example, the following program is valid:

```
let x : Bool & Int = true ,, 1 in (not x, succ x)
```

In the program above, the variable `x` has types Bool and Int, encoded by the intersection type Bool & Int. At the term level, `x` is created with the merge operator and can be regarded as either a boolean or an integer when used. For instance, in the program above there are two uses of x, one as a boolean (as the argument to `not`) and one as an integer (as the argument to `succ`). A language with the merge operator is able to extract the value of the right type

Table 1 Summary of common operators on environments. `E` denotes an environment, `I` denotes a set of identifiers, and `T1` and `T2` denote terms in the language.

Operator	**Description**
`export`	Exports/reifies the full *current* environment.
`E\I`	Returns a new *restricted* environment that only contains the identifiers in `I` from the environment `E`.
`import(T1, T2)`	Evaluates `T1` to be an environment `E1`, and uses `E1` to evaluate `T2`.
`import(I, T1, T2)`	Evaluates `T1` to be an environment `E1`, checks that a set of identifiers `I` are defined in `E1`, then uses `E1` to evaluate `T2`.
`E1, E2`	Composes/concatenates two environments.

from merges. In many classical systems with intersection types, but without the presence of the merge operator, the type Bool & Int cannot be inhabited and the program above is not expressible [34].

An important issue that the merge operator introduces is ambiguity. What happens if merges contain multiple values of the same type? For example, we could have `(1,,2):Int`, but if this is allowed, then it could result in either `1` or `2`. To address the ambiguity problem, Oliveira et al. [32] presented the λ_i calculus, which imposed a restriction where only merges of values that have *disjoint types* are accepted (we use $A * B$ to represent that A is disjoint with B). In this way, ambiguous programs such as `1,,2` are rejected since `Int` is not disjoint with itself. However, `Bool` and `Int` are disjoint, and thus `true,,1` is a well-typed expression.

As Dunfield [15] argued, with the merge operator, many language features such as *dynamic typing*, *multi-field records*, and *operator overloading* can be easily encoded. After that, several non-trivial programming language features, including *dynamic mixins* [2], *first-class traits* [5], *nested composition* [6,22] have been enabled with the help of the merge operator and disjoint intersection types. These features provide the foundations for *compositional programming* [51], which is a programming paradigm that enables a simple and natural solution to the Expression Problem [49] and other modularity problems. Compositional programming is realized in the CP language [51], which has been used to demonstrate the expressive power of the paradigm.

First-class environments. Normally, environments are not a syntactic entity of a programming language. Instead, environments exist implicitly at the meta-level for defining formal semantics and implementing languages. However, some dynamically typed languages, including dialects of Lisp [18] or the R language [17], include support for first-class environments. There is a line of research work on first-class environments for dynamically typed languages [18,23,24,29,36]. First-class environments provide a lot of expressive power, and they are used to model many other language constructs. With first-class environments, it is possible to model *closures*, *modules*, *records* or *object-oriented constructs* [18]. Moreover, it is also possible to model declarations directly, eliminating the need to distinguish between declarations and expressions.

To allow environments manipulated by not only compilers or interpreters but also programmers, a form of *reification* and *reflection* of environments is needed. Reification transforms environments into data objects and reflection enables data objects to be treated as environments [23,24]. While formalizations differ, generally speaking, environments are formalized as a mapping from variables to data objects, which can be manipulated at runtime. We summarize typical supported operators to manipulate environments [36] in Table 1 (with notations slightly changed).

Work on first-class environments for typed languages [44, 45, 47] comes with significant restrictions compared to what is supported in dynamically typed languages. In these calculi, types and environment types are defined such that environment types are a special kind of type. The definition of types is $A, B ::= A \to B \mid \ldots \mid E$, and each environment type E has the form of $\{x_1 : A_1, \ldots, x_m : A_m\}$ where A_i $(1 \leq i \leq m)$ is a type and each variable x_i must be distinct (or disjoint) with each other. Environment types encode exactly the normal typing context, which is a set that consists of typing assumptions $x_i : A_i$. Correspondingly, an environment has the form of $\{a_1/x_1, \ldots, a_m/x_m\}$ that binds variables x_i with terms a_i [44, 47]. There are two constructs related to environments:

- The first construct returns the current environment which acts similarly to `export`.
- The second construct is an evaluation operation $e[\![a]\!]$ that evaluates the expression a under an environment e. Note that, this operation is similar to `import(T1, T2)` in Table 1 (where `T1` corresponds to e and `T2` corresponds to a).

With these two constructs, one can create an environment at run-time and use it for evaluation. However, types are not totally unified with environment types in this setting, which results in special treatment of environments. For example, the expression e in $e[\![a]\!]$ can only be an environment. To avoid runtime errors, the typing rule for $e[\![a]\!]$ restricts the type that e has to be an environment type. Existing type systems with environment types do not consider subtyping. At the term level, though environments can be computed by evaluation under other environments and function applications, concatenation or restriction of environments are not supported. Therefore, an environment with a larger/smaller width cannot be constructed on the fly either. In short, there is no subtyping and the operations that are supported in dynamically typed languages in Table 1 are not fully supported in typed calculi with first-class environments.

Program fragments and separate compilation. To motivate our work we will show how first-class environments can be helpful to model a simple form of modules. Our form of modules is inspired by Cardelli's [7] *program fragments*. Here we first introduce the notion of program fragments, and in Section 2.3 we will see how we can model program fragments in E_i.

A *program fragment*, or *module*, is a syntactically well-formed expression where free variables may occur [7]. *Separate compilation* decomposes a program into program fragments that can be typechecked and compiled separately. A program fragment may contain free variables. However, if the required interface that contains adequate type information is specified, then the types of the free variables can be found (without any concrete implementation). Thus, the typechecking of a program fragment can still be carried out *separately*.

In a conventional calculus, such as the simply typed lambda calculus (STLC), we express abstractions over a variable annotated with a type. However, there are no facilities for abstracting over an interface that may consist of multiple (nested) type assumptions. In other words, the STLC is not powerful enough to model separate compilation.

Cardelli [7] proposed a calculus of program fragments for the STLC, and specified high-level abstractions for modules and interfaces. In Cardelli's framework, interfaces are interpreted as typing contexts that are *external* to the language. A module that may require an interface/context is represented as a binding judgment $E \vdash d \mathrel{.\!\cdot.} S$, where E is a context, d a list of definitions, S a list of type declarations. Take the following modules from Cardelli as an example:

```
module
import nothing
export x:Int
begin
  x : Int = 3
end.
```

```
module
import x:Int
export f:Int → Int, z:Int
begin
  f : Int → Int = λ(y:Int).y+x
  z : Int = f(x)
end.
```

These two modules can be modelled as two binding judgments:

$$\emptyset \vdash (x : \mathsf{Int} = 3) \mathrel{.\!\raisebox{1ex}{.}\!.} (x : \mathsf{Int})$$

$$x : \mathsf{Int} \vdash (f : \mathsf{Int} \to \mathsf{Int} = \lambda(y : \mathsf{Int}).y + x, z : \mathsf{Int} = f(x)) \mathrel{.\!\raisebox{1ex}{.}\!.} (f : \mathsf{Int} \to \mathsf{Int}, z : \mathsf{Int})$$

A module is encoded as a list of definitions d, with an import list modelled as a context E and an export list as type declarations S. In the second module, `z` relies on `f`. To model such dependency, the binding judgment $E \vdash d \mathrel{.\!\raisebox{1ex}{.}\!.} S$ is designed to be *dependent*: each component depends on its previous components in d, in the sense that every free variable in this component can refer to its corresponding type. To check whether $z : \mathsf{Int} = f(x)$ is matched by $z : \mathsf{Int}$, the type declaration $f : \mathsf{Int} \to \mathsf{Int}$ is appended to the original context $x : \mathsf{Int}$ to be a type assumption. In this way, the second binding judgment can be checked separately since each variable can access sufficient type information.

Though each binding judgment can be separately compiled to a self-contained entity called a linkset, user-defined abstractions cannot be expressed in Cardelli's work, since a binding judgment itself is a meta-level notion that cannot be created by programmers. In our work, we also regard interfaces as typing contexts. However, we unify typing contexts and types, and there are *first-class* constructs that abstract over a type/interface. We will discuss our ideas in detail in Section 2.3.

2.2 Limitations of Non-Dependent Merges

As Section 2.1 argued, both the (non-dependent) merge operator and first-class environments are useful to model a variety of other language constructs. Some of these language constructs can even be modelled by both merges or first-class environments. Given the overlap between merges and first-class environments it is reasonable to try to unify them, to obtain a more powerful model of statically typed languages with first-class environments. Our goal is to use merges to model first-class environments. However, non-dependent merges in existing calculi such as λ_i are inadequate for this purpose. This section discusses the limitations of non-dependent merges that are addressed by us.

No support for reification and reflection of environments. Intersection types and the merge operator are powerful tools that enable many language features, one of which is multi-field records [40]. In fact, multi-field record types can be turned into an intersection of single-field record types:

$$\{l_1 : A_1,\ \dots,\ l_n : A_n\} \equiv \{l_1 : A_1\} \,\&\, \cdots \,\&\, \{l_n : A_n\}$$

Recall the syntax of conventional typing contexts: $\Gamma ::= \cdot \mid \Gamma, x : A$. A typing context is a list of pairs that bind variables with types. If we view variables as labels, typing contexts can be encoded as multi-field records, which are further desugared to intersections of single-field record types. Similarly, at the term level, a multi-field record is expressed as a merge of single-field ones:

$$\{l_1 = e_1,\ \ldots,\ l_n = e_n\} \equiv \{l_1 = e_1\} \,,,\, \ldots \,,,\, \{l_n = e_n\}$$

For example, $\{x = 2, y = 4\}$ is encoded as $\{x = 2\} \,,,\, \{y = 4\}$. In calculi with a merge operator, merges are always *first-class* expressions and thus they can be passed to functions.

However, in previous calculi with the merge operator [15, 32, 38], merges are not used to model environments. Therefore, there are no reification and reflection facilities for environments in those calculi. Furthermore, intersection types are not used to model contexts, and there is no construct that enables running some computation under a local environment. In short, previous calculi with the merge operator support concatenation, but they do not support other operations in Table 1.

No dependent merges. An important limitation of merges in previous work with respect to environments is that they cannot be dependent. Many programming languages, as well as Cardelli's program fragments, support declarations such as:

```
let x = 2
let y = 4
```

which allows several declarations to be associated with expressions. For the declarations above, we can easily model them as a (non-dependent) merge of two single field records:

```
{x = 2},,{y = 4}
```

where variables `x` and `y` are encoded as field names (or labels), and the values assigned to variables are modelled as record fields.

The previous declarations are *non*-dependent, in the sense that the expression assigned to `y` does not refer to `x`. However, in practice many declarations are *dependent*, where the current declaration relies on previous ones. For instance, fairly often we may have a program:

```
let x = 2
let y = x + x
let main = x + y
```

where `y` depends on `x` and `main` depends on both `y` and `x`. The traditional non-dependent merge operator cannot capture such cases. To be concrete, consider the typing rule for merges from λ_i [32]:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 \Rightarrow A \,\&\, B}\ \textsc{Typ-merge}$$

Here $A * B$ expresses that A and B are disjoint types. For typing a merge $e_1 \,,,\, e_2$, the typing context for the right branch e_2 is Γ and does not contain any type information about the left branch e_1. When typing e_2, the type of e_1 is never used during the typing procedure. As a result, e_2 cannot be built by referring to e_1. Moreover, to cooperate with the static semantics, the two branches of $e_1 \,,,\, e_2$ are evaluated separately without dependency involved in the dynamic semantics of λ_i. That is, the environment for evaluating e_2 does not contain the evaluation result of e_1.

The incapacity of encoding dependent declarations as first-class expressions exposes that λ_i is not able to fully model *module*-related language features. Since dependent definitions/declarations often occur in a module, as shown in our discussion on program fragments in Section 2.1.

2.3 Key Ideas

In this work, we utilize the merge operator together with new constructs to enable dependent merges and first-class environments. We concretize these ideas in a new calculus called E_i. The key ideas of our work are discussed next.

Typing contexts as types. In E_i the typing context in the typing judgment is a *type* instead of an association list. Our grammar for *both* types and contexts is:

$$A, B, \Gamma ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \mid \{l : A\}$$

A typing assumption $x : A$ in conventional calculi is modelled as a record type, and the intersection of two types plays a similar role to the concatenation of two association lists. However, the contexts in our work are not restricted to intersections of record types. In fact, any type (e.g., Int) defined in the syntax of E_i can be a typing context. If a context consists only of Top (the top type), then there is no type information in this context, which corresponds to an empty association list. As we will see, viewing typing contexts as types opens up the possibility of creating interesting language features.

Unifying environments and expressions. Just as typing contexts are types in E_i, environments in the reduction semantics are just *values* instead of association lists which bind variable names to values. Hence, environments are *first-class* in our setting. The top value $\top$ is used to model the empty environment. A merge of two values can be viewed as concatenation of two environments. For example, the merge $\{x = 1\} \,;\, \{y = 2\}$ is a valid environment that binds 1 and 2 to x and y respectively. In E_i, we denote the merge operator by a single comma ($;$) to follow the notation conventionally used in programming languages to denote the concatenation of two environments. With record projection, the value bound to a label can be accessed. Note that unifying environments and values and viewing variables as labels means that extra syntax (or data structures) for environments is not needed. This is different from previous work on typed calculi with first-class environments where an explicit notion of environments is introduced [44, 45, 47].

In E_i, we have two constructs to support reification (or exporting) and reflection (or importing) of environments. For reification, we employ the *query* construct ?. The query construct is inspired by the implicit calculus [12], where queries are used to query *implicit* environments by type. In E_i we apply queries directly to runtime environments instead, whereas in the implicit calculus, access to the regular environments is done conventionally using named variables. The typing rule for ? is simply:

$$\Gamma \vdash ? \Rightarrow \Gamma$$

i.e. the query ? synthesizes the current context. For example, $\{x : \mathsf{Int}\} \vdash ?.x \Rightarrow \mathsf{Int}$ is valid. Here ? obtains the current environment and accesses the field x.

Regarding the reflection of environments, there is a construct $e_1 \rhd e_2$ that is called *box* in E_i. In a box, e_2 is assigned an expression e_1, which is evaluated to be a value that acts as an environment for evaluating e_2. Take $\{x = 1 + 1\} \rhd ?.x + 1$ as an example. The expression $\{x = 1 + 1\}$ is given as the environment to $?.x + 1$. Then $\{x = 1 + 1\}$ is evaluated to $\{x = 2\}$, under which $?.x + 1$ is evaluated to 3. The box construct can be seen as the inverse operator of the query, since $? \rhd e$ is equivalent to e in the sense that ? exports the full environment by default. Allowing e_1 in the box $e_1 \rhd e_2$ to be any well-typed expression instead of a value adds expressiveness to reflection. For example, environment injection can be encoded as $(? \,;\, v) \rhd e$ where v is added to the original environment for e *locally.*

In E_i, type annotations play a role in information hiding. For example, for a merge with an annotation $(\{x = 1\} \fatsemi \{y = 2\}) : \{x : \mathsf{Int}\}$, only $\{x = 1\}$ is visible. Type annotations provide a mechanism to enable restriction, since they are able to prevent visibility of certain values. Since environments are values in our setting, type annotations can seal the environment, such that only components named in the type are accessible. Therefore, reification and reflection are *type-directed* in E_i. With type annotations, users can choose part of the environment that they desire. In summary, E_i can essentially model all the operations on environments in Table 1 with the following expressions:

- $?$ reifies the entire environment;
- $? : A$ obtains part of the environment that has type A;
- $e_1 \triangleright e_2$ evaluates e_1 to an environment and uses that to evaluate e_2 under that environment;
- $(e_1 : A) \triangleright e_2$ evaluates e_1, but restricts the resulting environment to A and uses that to evaluate e_2;
- $e_1 \fatsemi e_2$ concatenates two environments e_1 and e_2.

Dependent merges. To model dependent declarations, the merges in our work are dependent. The right branch can refer to the type of the left branch. The typing rule for dependent merges is:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \,\&\, A \vdash e_2 \Rightarrow B \qquad A * \Gamma \qquad A * B}{\Gamma \vdash e_1 \fatsemi e_2 \Rightarrow A \,\&\, B}\ \textsc{Typ-dmerge}$$

Modelling *typing contexts as types* enables type information flowing from the left branch to the right branch in a merge. Specifically, for $e_1 \fatsemi e_2$, the type of e_1 is added into the current context such that e_2 synthesizes a type under the intersection type $\Gamma \,\&\, A$.

Suppose that the current context Γ is a subtype of some type B, with type annotation, $? : B$ exports B from Γ. With the query construct, a dependent declaration can be encoded as a dependent merge:

```
{x = 2} ⨟ {y = (?:{x:Int}).x + (?:{x:Int}).x}
```

which has type $\{x : \mathsf{Int}\} \,\&\, \{y : \mathsf{Int}\}$. The annotated query $? : \{x : \mathsf{Int}\}$ exports $\{x : \mathsf{Int}\}$ from the context, and projection $(? : \{x : \mathsf{Int}\}).x$ infers the type Int. Note that E_i is meant as a minimal core calculus and it is not built with convenience in mind. So the expression above is more cumbersome than what a programmer would expect to write in a source language. With some basic support for type inference and syntactic sugar in a source language, we could write instead:

```
{x = 2} ⨟ {y = ?.x + ?.x}
```

or even:

```
{x = 2} ⨟ {y = x + x}
```

In Section 5 we show how some of this syntactic sugar and inference can be achieved. For readability purposes, in the following examples, we will take the liberty to use a more lightweight syntax for the examples written in E_i as well.

In general, dependent declarations can be modelled as a merge of expressions $e_1 \fatsemi \cdots \fatsemi e_n$, where the type information accumulates from e_1 to e_n. Modelling declarations as merges means that while we can benefit from the expressiveness of the merge operator, we do not need to introduce an additional syntax for declarations. Besides the condition $A * B$ that avoids conflicts between two branches in a merge, in the typing rule for dependent merges there is an extra disjointness condition $A * \Gamma$ to ensure that the new environment has no conflicts. This extra disjointness condition is needed to ensure that reduction is deterministic in E_i.

TDOS environment-based semantics. In E_i, an environment-based semantics, expressed by a reduction relation of the form $v \vdash e_1 \hookrightarrow e_2$, is employed to capture the dynamic behavior of expressions. In contrast to more conventional small-step reduction relations, which are typically based on substitution and beta reduction, here v plays the role of the runtime environment and no substitution is needed during reduction. Basically, an environment is stored during evaluation and the expression being evaluated can access it. During the reduction procedure, the environment can be changed *locally*. For example, suppose that the current environment is v, to evaluate a dependent merge $e_1 \fatsemi e_2$. The left branch e_1 is evaluated to a value v_1 first. After that, v_1 is merged with v such that e_2 is evaluated under $v \fatsemi v_1$. As a result, e_2 is able to access and fetch v_1. For instance, the dependent merge

```
{x = 2} ⨟ {y = ?.x + ?.x}
```

is evaluated to $\{x = 2\} \fatsemi \{y = 4\}$ under $\top$, since

```
{y = ?.x + ?.x}
```

is evaluated to $\{y = 4\}$ under the environment $\top \fatsemi \{x = 2\}$.

The reduction semantics is based on a *type-directed operational semantics* (TDOS), following the semantics of calculi with the merge operator [21]. As we have seen, type annotations can be used to remove information from values. Thus, unlike many other calculi, the semantics of E_i is type-dependent. That is, types affect the runtime behavior. To deal with such type-dependent semantics based on giving an operational behavior to type annotations we use a TDOS. In the TDOS there is a *casting* relation $v \hookrightarrow_A v'$, where types are used to guide reduction. Since an environment can be selected by a type annotation, casting also acts as a tool for *synthesizing* values in E_i. During the reduction of an annotated query $? : A$ under environment v, casting is triggered, and v' is synthesized as the result. Take the program above as an example, to evaluate $? : \{x : \mathsf{Int}\}$, which is needed in the projection $?.x$, the following cast is triggered:

$$\top \fatsemi \{x = 2\} \hookrightarrow_{\{x:\mathsf{Int}\}} \{x = 2\}$$

In essence, the cast extracts the value $\{x = 2\}$ matching the type being cast. With this value, we can further build an expression for the right part of the merge.

Abstractions in E_i. In E_i, an abstraction has the form $\{e\}^m$ where m denotes a mode. There are two modes for abstractions: $\bullet$ and $\circ$. Here we focus on $\{e\}^\bullet$. Compared with a normal lambda abstraction $\lambda x.e$, there is no variable binding in $\{e\}^\bullet$, since values in the environment are *looked up by types* via the query construct instead of by variable names. For example, after $\{?\}^\bullet : \mathsf{Int} \to \mathsf{Int}$ is applied with integer 1, the input 1 is put in the environment for evaluating $? : \mathsf{Int}$, and then the query construct looks up a value of type Int, which is 1. We require that a well-typed abstraction $\{e\}^\bullet$ has a type annotation. The (slightly simplified) typing rule for abstractions is:

$$\frac{\Gamma * A \qquad \Gamma \,\&\, A \vdash e \Leftarrow B}{\Gamma \vdash \{e\}^\bullet : A \to B \Rightarrow A \to B}\ \textsc{Typ-abs}$$

Similarly to typing normal lambda abstractions, where a typing assumption $x : A$ is added to the typing context, for typing $\{e\}^\bullet$ in E_i, the input type of $\{e\}^\bullet$ is added into the context to type check the body e. For example, $\mathsf{Top} \vdash \{?\}^\bullet : \mathsf{Int} \to \mathsf{Int} \Rightarrow \mathsf{Int} \to \mathsf{Int}$ is valid, since under $\mathsf{Top} \,\&\, \mathsf{Int}$, $?$ can check against Int. Besides, there is also a disjointness condition in this rule, which ensures that there are no conflicts between the context and the input type.

Ambiguity would happen without such a condition since, if the body e contains a ?, there would be different answers to the query ?, as shown in the following example ($\Gamma \vdash e$ is used to denote the situation that the current context for e is Γ):

$$\mathsf{Int} \vdash (\{?\}^{\bullet} : \mathsf{Int} \to \mathsf{Int})\, 2$$

Suppose that the current environment contains only the value 1, which is of type Int. After the function is applied to 2, both 1 and 2 appear in the environment, and they have the same type Int. If ? desires a value of type Int, then there are two candidates, which results in ambiguity. Thus the condition $\Gamma * A$ prevents such programs. On the other hand, the following program is safe in the context Int, since there is only one value, which is 1, having type Int in the environment.

$$\mathsf{Int} \vdash (\{?\}^{\bullet} : \mathsf{Bool} \to \mathsf{Int})\, \mathsf{true}$$

In general, conventional calculi where variables are involved normally ensure that a typing context is unique, i.e., all variables in it are distinct. In our calculus, disjointness plays a similar role as uniqueness. A function cannot accept expressions that have overlapping types with the current context. For record types, $\{x : \mathsf{Int}\}$ is not disjoint with itself, so the following is not allowed:

$$\{x : \mathsf{Int}\} \vdash (\{?.x\}^{\bullet} : \{x : \mathsf{Int}\} \to \mathsf{Int})\, \{x = 1\}$$

In contrast, the following expression is well-typed in the context $\{x : \mathsf{Int}\}$, since two record types are disjoint if they have distinct labels:

$$\{x : \mathsf{Int}\} \vdash (\{?.x\}^{\bullet} : \{y : \mathsf{Int}\} \to \mathsf{Int})\, \{y = 1\}$$

Note that the use of records and distinct label names is how we can model conventional functions that take several arguments of the same type. That is, we can use labels to unambiguously distinguish between arguments of the same type, similarly to the use of distinct variable names in conventional lambda abstractions.

The abstractions in E_i essentially abstract over an interface if we view interfaces as types. The example from Cardelli in Section 2.1 can be encoded in our calculus:

$$\{M = \{x = 3\}\}$$
$$\{N = \{\{f = \{?.y + ?.x\}^{\circ} : \{y : \mathsf{Int}\} \to \mathsf{Int}\} \,,\, \{z = (?.f)\,(?.x)\}\}^{\bullet} : \{x : \mathsf{Int}\} \to \{f : \mathsf{Int} \to \mathsf{Int}\} \,\&\, \{z : \mathsf{Int}\}\}$$

Each module is modelled as a record (if the module does not import anything) or a function that returns a record (if the module imports something). A group of related definitions is expressed as a dependent merge of some other records. An interface, such as the interface of N, that contains typing assumption(s) is encoded as input type(s) of an abstraction, and the export list is the output type. Since merges are dependent in E_i, in the second module z is able to call f. With the $\circ$ mode abstraction, standard lambda abstractions can also be encoded (we will discuss this in Section 5). Both modules are typeable separately (in the empty context). Moreover, we can apply N with M, since $(?.N)\,(?.M)$ is typeable in the context containing N and M. Note that such an application is not expressible in Cardelli's work, since modules are not first-class in his setting.

Closures as a special case of boxes. As in usual environment-based semantics, closures are used in E_i to keep lexical environments around. However, given that we have the box construct in E_i, we do not need to invent a separate construct for closures. In fact, closures have the form of $v \rhd \{e\}^{\bullet} : A \to B$, which is just a special case of a box. In a box closure, the environment is a value and the expression under the environment is an annotated abstraction. Note that closures are values and the abstraction inside is not evaluated. Instead, when a closure is applied with a value, the value is put in the environment of the closure, and the body of the abstraction is going to be evaluated under the extended environment. Take the following evaluation as an example:

$$\begin{array}{ll}
 & (\{\{?\}^{\bullet} : \mathsf{Int} \to \mathsf{Bool}\}^{\bullet} : \mathsf{Bool} \to \mathsf{Int} \to \mathsf{Bool}) \;\; \mathsf{true} \;\; 1 \\
\hookrightarrow & (\top \rhd \{\{?\}^{\bullet} : \mathsf{Int} \to \mathsf{Bool}\}^{\bullet} : \mathsf{Bool} \to \mathsf{Int} \to \mathsf{Bool}) \;\; \mathsf{true} \;\; 1 \\
\hookrightarrow^{*} & (\top \fatsemi \mathsf{true} \rhd \{?\}^{\bullet} : \mathsf{Int} \to \mathsf{Bool}) \;\; 1 \\
\hookrightarrow & \top \fatsemi \mathsf{true} \fatsemi 1 \rhd ? : \mathsf{Bool} \\
\hookrightarrow^{*} & \mathsf{true}
\end{array}$$

The abstraction takes a boolean and an integer as input and returns the boolean. At first, it is packed up with the empty environment to form a closure. Then the two values true and 1 are merged with the environment to evaluate the body $? : \mathsf{Bool}$. With casting, the annotated query is evaluated to true.

Encoding λ_i. To demonstrate the expressiveness of E_i we show that it can encode all well-typed programs in the λ_i calculus [32]: an existing calculus with non-dependent merges and without first-class environments. There are two non-obvious obstacles in the encoding. Firstly, unlike E_i, the λ_i calculus is a conventional lambda calculus with conventional lambda abstractions and variables. Our encoding of λ_i shows that queries and abstractions in E_i can encode conventional variables and lambda abstractions. The second obstacle in the encoding is that dependent merges have more disjointness constraints than non-dependent merges. Therefore, it is not clear how some non-dependent merges may be encoded. However, a combination of dependent merges and other constructs in the E_i calculus enables an encoding of all non-dependent merges. Section 5 details the encoding and proves that all typeable programs in λ_i are encodable and typeable in E_i.

3 The E_i Calculus

In this section we present the E_i calculus, which is a calculus with dependent merges and first-class environments. In E_i, type contexts are types, and run-time environments can be assembled, composed, manipulated explicitly, and used to run computations under such environments.

3.1 Syntax

The syntax of E_i is as follows:

$$\begin{array}{lrl}
\text{Labels} & & l, x, y, z, \ldots \\
\text{Types and Contexts} & A, B, \Gamma & ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \mid \{l : A\} \\
\text{Function modes} & m & ::= \bullet \mid \circ \\
\text{Expressions} & e & ::= ? \mid i \mid \top \mid \{e\}^{m} \mid e_1 \rhd e_2 \mid e_1\, e_2 \mid e_1 \fatsemi e_2 \mid e : A \mid \{l = e\} \mid e.l \\
\text{Values} & v & ::= i \mid \top \mid v \rhd (\{e\}^{\bullet} : A \to B) \mid v \rhd (\{e\}^{\circ} : \{l : A\} \to B) \mid \{l = v\} \mid v_1 \fatsemi v_2
\end{array}$$

Types and contexts. In E_i there is no syntactic distinction between types and contexts: contexts are types and any type can be a context. In standard calculi typing contexts are lists of typing assumptions of the form $x : A$ that associates variable x with type A. This particular case is encoded in E_i with a single-field record type $\{x : A\}$. For clarity, we use different meta-variables to denote different uses of types (A, B, C, etc.) and contexts (Γ). Two basic types are included: the integer type Int and the top type Top. Function types and intersection types are created with $A \to B$ and $A \,\&\, B$ respectively. $\{l : A\}$ denotes a record type in which A is the type of the field. Multi-field record types can be desugared to an intersection of single-field record types [32,40].

Expressions. Meta-variable e ranges over expressions. Expressions include some constructs in standard calculi with a merge operator: integers (i); a canonical top value $\top$, which can be seen as a merge of zero elements; annotated expressions ($e : A$); application of a term e_1 to term e_2 (denoted by $e_1\ e_2$); and merge of expressions e_1 and e_2 ($e_1 \,{,}\!{,}\, e_2$). The expression $\{l = e\}$ denotes a single-field record where l is the label and e is its field. Similarly to record types, a multi-field record can be viewed as a merge of single-field records. Projection $e.l$ selects the field from e via the label l.

Besides these standard constructs, there are some novel constructs in our system. Unlike standard calculi, where variables are used to lookup values, we borrow the query construct ? from implicit calculi [12] to *synthesize values by types*. However, unlike implicit calculi, in E_i we can completely eliminate the need for variables, since a combination of queries and other constructs can encode traditional uses of variables. Such encoding will be discussed in detail in Section 5. The absence of variables simplifies binding in comparison to other calculi. $\{e\}^m$ stands for *abstractions* in which m is the mode of an abstraction and can be either $\bullet$ or $\circ$. Abstractions play the same role as lambda abstractions, but they abstract over the input type of the function, instead of abstracting over a variable. The $\circ$ mode denotes a special form of abstraction that is useful to encode lambda abstractions. The term $e_1 \rhd e_2$ is called a *box*. A box assigns a local environment e_1 for e_2, and e_2 is not affected by the global context or environment. In other words, boxes allow the computation of e_2 to be performed under the runtime environment resulting from e_1.

Values. The meta-variable v ranges over values. Values include integers, the canonical $\top$ value, *closures*, merges of values and records in which the field is a value. Closures are a special kind of box, in which the local environment e_1 is a value and e_2 is an annotated abstraction. For closures, the type annotation for $\{e\}^\bullet$ can be any arrow type, whereas the input type of the type annotation for $\{e\}^\circ$ can only be a record type.

3.2 Subtyping and Disjointness

Subtyping. The subtyping rules, shown in Figure 1, are standard for a calculus with intersection types [13], but they include an additional rule S-RCD for subtyping record types. Note that the combination of the subtyping rules for intersection types and record types enables us to express both depth and width subtyping for multi-field record types (which are just encoded as intersections of single-field record types). This extended subtyping relation is reflexive and transitive [22].

Disjointness. Compared to λ_i, disjointness is defined in a slightly different way, inspired by an approach suggested by Rehman et al. [37]. To make two functions or two records mergeable, we define disjointness based on ordinary types whose definition is shown in Figure 1. There are two variants of ordinary types in E_i. The one for defining disjointness contains premises marked in gray. In this variant, ordinary types are inductively defined to be types where the top type and intersection types can never appear (except as input types of functions). With the help of ordinary types, we define disjointness as:

$\boxed{A <: B}$ *(Subtyping)*

$$\text{S-Z}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \text{S-TOP}\ \frac{}{A <: \mathsf{Top}} \qquad \text{S-ARR}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \text{S-ANDL}\ \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3}$$

$$\text{S-ANDR}\ \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3} \qquad \text{S-AND}\ \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3} \qquad \text{S-RCD}\ \frac{A <: B}{\{l : A\} <: \{l : B\}}$$

$\boxed{\mathsf{Ordinary}\ A}$ *(Ordinary Types)*

$$\text{O-INT}\ \frac{}{\mathsf{Ordinary}\ \mathsf{Int}} \qquad \text{O-ARROW}\ \frac{\mathsf{Ordinary}\ B}{\mathsf{Ordinary}\ A \to B} \qquad \text{O-RCD}\ \frac{\mathsf{Ordinary}\ B}{\mathsf{Ordinary}\ \{l : B\}}$$

Figure 1 Subtyping and ordinary types.

▶ **Definition 1** (Disjointness). $A * B \equiv \neg(\exists C, \mathsf{Ordinary}\ C \wedge A <: C \wedge B <: C)$

Two types are disjoint if and only if the two types do not share any common ordinary supertype. We have proved that our definition of disjointness is equivalent to the one employed by Huang et. al [22] in their formulation of λ_i. This definition states that atomic values, which can inhabit the two types, cannot have overlapping types. Importantly, our definition allows two arrow types or two record types to be disjoint. For example, $\mathsf{Int} \to \mathsf{Bool}$ is disjoint with $\mathsf{Int} \to \mathsf{Char}$ as the two types do not share a common ordinary supertype. Note that there is also an equivalent algorithmic definition of disjointness, which is shown in the appendix. Some of the fundamental properties of disjointness are shown next:

▶ **Lemma 2** (Disjointness Properties). *Disjointness satisfies:*

1. *If $A * B$, then $B * A$.*
2. *$A * (B \,\&\, C)$ if and only if $A * B$ and $A * C$.*
3. *If $A * (B_1 \to C)$, then $A * (B_2 \to C)$.*
4. *$A * B$ if and only if $\{l : A\} * \{l : B\}$.*
5. *$C * D$ if and only if $(A \to C) * (B \to D)$.*
6. *If $A <: B$ and $A * C$, then $B * C$.*

3.3 Bidirectional Typing

The type system of E_i shown in Figure 2 is bidirectional. There are two modes of typing, where $\Rightarrow$ and $\Leftarrow$ denote the synthesis and checking modes respectively. The notation $\Leftrightarrow$ is a metavariable for typing modes. The meaning of typing judgment $\Gamma \vdash e \Leftrightarrow A$ is standard: under the context Γ (which is a type), expression e can synthesize (with $\Rightarrow$) or check against (with $\Leftarrow$) A.

Typing the query construct. Rule TYP-CTX states that ? can synthesize the context. With rule TYP-SUB, ? checks against any type that is a supertype of the context. In addition, with rule TYP-ANNO, under a context Γ, for any supertype A of Γ, $? : A$ can synthesize A. Since

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Bidirectional Typing)*

$$\textsc{Typ-lit}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-ctx}\ \frac{}{\Gamma \vdash ? \Rightarrow \Gamma} \qquad \textsc{Typ-top}\ \frac{}{\Gamma \vdash \top \Rightarrow \mathsf{Top}} \qquad \textsc{Typ-anno}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}$$

$$\textsc{Typ-abs}\ \frac{\Gamma * A \qquad C <: A_m \qquad \Gamma \,\&\, A \vdash e \Leftarrow B}{\Gamma \vdash \{e\}^m : A \to B \Rightarrow C \to B} \qquad \textsc{Typ-app}\ \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}$$

$$\textsc{Typ-box}\ \frac{\Gamma \vdash e_1 \Rightarrow \Gamma_1 \qquad \Gamma_1 \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1 \triangleright e_2 \Rightarrow A} \qquad \textsc{Typ-rcd}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}} \qquad \textsc{Typ-proj}\ \frac{\Gamma \vdash e \Rightarrow \{l : A\}}{\Gamma \vdash e.l \Rightarrow A}$$

$$\textsc{Typ-sub}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B} \qquad \textsc{Typ-mergev}\ \frac{\Gamma \vdash v_1 \Rightarrow A \qquad \Gamma \vdash v_2 \Rightarrow B \qquad v_1 \approx v_2}{\Gamma \vdash v_1 ,, v_2 \Rightarrow A \,\&\, B}$$

$$\textsc{Typ-dmerge}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \,\&\, A \vdash e_2 \Rightarrow B \qquad A * \Gamma \qquad A * B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}$$

Type Extraction A_m

$A_\bullet$	$=$	A
$\{l : A\}_\circ$	$=$	A

Figure 2 Bidirectional type system of E_i. The syntax for the bidirectional modes is defined as $\Leftrightarrow ::= \Rightarrow \mid \Leftarrow$.

contexts are types in our system, a supertype of a type means a portion of a typing context. By annotating ? with a supertype of the context, we can proactively pick the desired type information (or equivalently, hide part of type information) from the context. For example, $\mathsf{Int} \,\&\, \mathsf{Bool} \vdash ? : \mathsf{Int} \Rightarrow \mathsf{Int}$ is valid, and allows us to pick Int from a typing context with $\mathsf{Int} \,\&\, \mathsf{Bool}$.

$$\textsc{Typ-anno}\ \dfrac{\textsc{Typ-sub}\ \dfrac{\mathsf{Int} \,\&\, \mathsf{Bool} \vdash ? \Rightarrow \mathsf{Int} \,\&\, \mathsf{Bool} \qquad \mathsf{Int} \,\&\, \mathsf{Bool} <: \mathsf{Int}}{\mathsf{Int} \,\&\, \mathsf{Bool} \vdash ? \Leftarrow \mathsf{Int}}}{\mathsf{Int} \,\&\, \mathsf{Bool} \vdash ? : \mathsf{Int} \Rightarrow \mathsf{Int}}$$

Typing abstractions. Rule Typ-abs is the typing rule for abstractions. An abstraction can synthesize an arrow type, in which the shape of the input type is determined by the mode. For $\{e\}^\bullet : A \to B$ we simply synthesize the type $A \to B$. For $\{e\}^\circ$ with an annotation, $\{e\}^\circ$ is well-typed only if the input type is a record type. Furthermore $\{e\}^\circ : \{l : A\} \to B$ synthesizes $A \to B$ where A is extracted from $\{l : A\}$. This peculiar treatment of $\{e\}^\circ : \{l : A\} \to B$ is because we wish to be able to model conventional lambda abstractions of the form $\lambda l.\ e : A \to B$ faithfully. In conventional lambda abstractions, the labels or variable names are only used *internally*, but they are not reflected on the type. The $\{e\}^\circ$ abstractions model this behavior and also hide the label information on the type. While an abstraction with the $\bullet$ mode accepts an expression of a type which is the exact input type of its annotation, a well-typed abstraction with the $\circ$ mode can only have an annotation of form $\{l : A\} \to B$. The label information for the input type is forgotten for the overall type of the abstraction.

Note also that, for obtaining type preservation, there is a subtyping condition in rule Typ-abs, similarly to the approach employed by Huang and Oliveira [21]. In an implementation of E_i, this subtyping condition can be omitted and we can let $\{e\}^m : A \to B$ infer $A_m \to B$ directly, since the condition is only used in E_i to ensure that closures, which are used during reduction at runtime, are type-preserving. In addition to avoiding ambiguity of the type-based lookups, when we introduce assumptions into the context, we need to ensure that the new assumptions are disjoint to the existing assumptions in the environment. Thus rule Typ-abs also has a disjointness premise to ensure this.

Typing dependent merges. Rule Typ-dmerge is the typing rule for merges. Unlike previous work for intersection types and the merge operator [22], the merges are *dependent* in our work. For a specific merge $e_1 \mathbin{,,} e_2$, the right branch e_2 may depend on the left branch e_1. The typing context for e_2 in the premises is the intersection type $\Gamma \,\&\, A$, which means that e_2 is affected by not only the global context Γ but also the synthesized type of e_1. In this way, e_2 can be constructed with the information of e_1, as illustrated by the following example:

$$\{z : \mathsf{Int}\} \vdash \{x = 1\} \mathbin{,,} \{y = (? : \{x : \mathsf{Int}\}).x + 1\} \;\Rightarrow\; \{x : \mathsf{Int}\} \,\&\, \{y : \mathsf{Int}\}$$

The right branch $\{y = (? : \{x : \mathsf{Int}\}).x + 1\}$ makes use of the type information of the left branch, by using ? to pick $\{x : \mathsf{Int}\}$ from $\{z : \mathsf{Int}\} \,\&\, \{x : \mathsf{Int}\}$. Then it will be able to utilize the value information from $\{x = 1\}$ to evaluate the expression in the right branch of the merge.

There are two disjointness conditions in rule Typ-dmerge. One is $A * B$, which makes two branches e_1 and e_2 be merged safely without ambiguities as in previous work [22]. However, this condition is not sufficient to prevent all the conflicts between values when the merges are dependent. An additional disjointness condition $A * \Gamma$ is needed to ensure that the synthesized type of the left branch e_1 is disjoint with the context. Without this extra condition, there can be conflicts between e_1 and the current environment. Take the following as an example:

$$(\mathsf{Int} \to \mathsf{String}) \,\&\, \mathsf{Int} \vdash 2 \mathbin{,,} ((? : \mathsf{Int} \to \mathsf{String})\; (? : \mathsf{Int}))$$

The context contains type $\mathsf{Int} \to \mathsf{String}$ and Int, and the left branch, 2, has type Int which clashes with the Int that is already in the context. The right branch is an application, which picks a closure and another integer value, say 1, from the current environment. Suppose that the closure returns the string representation of the input integer. Then $? : \mathsf{Int}$ in the right branch can choose either 1 from the environment or 2 from the left branch, and the merge above can be non-deterministically evaluated to either $2 \mathbin{,,}$ "1" or $2 \mathbin{,,}$ "2". Since we wish to have deterministic evaluation, we prevent such cases with the additional disjointness condition $A * \Gamma$.

Consistency, boxes and closures. Rule Typ-mergev is the typing rule for consistent merges. This rule is identical to the rule in previous work using non-dependent merges [21]. Like in previous work, rule Typ-mergev is a special run-time typing rule for merges of values and can be omitted in a programming language implementation. If two consistent values are well-typed then it is safe to merge them together. One may wonder why in this rule the context is not extended with A to type-check v_2. The reason is that values are *closed*, so they cannot depend on the information that is present in the context. During the reduction process, such information has been already filled in into the values. Consistency is defined in terms of casting (whose definition is shown in Figure 3):

▶ **Definition 3** (Consistency). *Two values v_1 and v_2 are said to be consistent (written as $v_1 \approx v_2$) if for any type A, the result of casting for the two values is identical.*

$$v_1 \approx v_2 \equiv \forall\, A,\ \textit{if}\ v_1 \hookrightarrow_A v_1' \ \textit{and}\ v_2 \hookrightarrow_A v_2' \ \textit{then}\ v_1' = v_2'$$

$\boxed{v \hookrightarrow_A v'}$ *(Casting)*

$$\textsc{Casting-int}\quad \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{Casting-top}\quad \frac{}{v \hookrightarrow_{\mathsf{Top}} \top}$$

$$\textsc{Casting-arrow}\quad \frac{\neg\rceil D \lceil \qquad C <: A_m \qquad B <: D}{v \rhd (\{e\}^m : A \to B) \hookrightarrow_{C \to D} v \rhd (\{e\}^m : A \to D)}$$

$$\textsc{Casting-arrowtl}\quad \frac{\rceil D \lceil \qquad C <: A_m \qquad B <: D}{v \rhd (\{e\}^m : A \to B) \hookrightarrow_{C \to D} (C \to D)^{\uparrow}}$$

$$\textsc{Casting-mergevl}\quad \frac{v_1 \hookrightarrow_A v_1' \qquad \mathsf{Ordinary}\ A}{v_1 \fatsemi v_2 \hookrightarrow_A v_1'}$$

$$\textsc{Casting-mergevr}\quad \frac{v_2 \hookrightarrow_A v_2' \qquad \mathsf{Ordinary}\ A}{v_1 \fatsemi v_2 \hookrightarrow_A v_2'}$$

$$\textsc{Casting-and}\quad \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 \fatsemi v_2} \qquad \textsc{Casting-rcd}\quad \frac{v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}}$$

Figure 3 Casting of E_i.

Given two values, if they have disjoint types, then they are consistent:

▶ **Lemma 4** (Disjointness implies consistency). *If $A * B$, $\Gamma_1 \vdash v_1 \Rightarrow A$, and $\Gamma_2 \vdash v_2 \Rightarrow B$, then $v_1 \approx v_2$.*

Rule Typ-box is the rule for boxes. To make a box $e_1 \rhd e_2$ well-typed, the global context Γ is replaced for e_2 with type Γ_1, which is the synthesized type of the local environment e_1. In other words, the expression e_2 in the box is only affected by the local context. As a special kind of box, closures are closed since the local environment for them is a value and this information is stored for the abstraction. Thus, it is always safe to change the context for closures to any other context. However, we cannot do that for abstractions. For example, if the context for $\{?\}^{\bullet} : \mathsf{Int} \to \mathsf{Int}$ is changed from Top to Int, then the disjointness condition in rule Typ-abs is broken.

Generally speaking, changing the typing context may introduce more type information such that disjointness does not hold anymore. For example, suppose that the current context is Int, which is disjoint with $\mathsf{Char} \,\&\, \mathsf{Bool}$. If we replace Int with $\mathsf{Int} \,\&\, \mathsf{Bool}$, then the new type information Bool is introduced in the context and it conflicts with $\mathsf{Char} \,\&\, \mathsf{Bool}$. For disjoint values, Lemma 4 ensures that the values are also consistent, so they can be merged together. Therefore, typing two disjoint values does not rely on rule Typ-dmerge, which restricts the type of the left branch to be disjoint with the context. In fact, another way to describe the closedness of values is to show that the typing context for values can be replaced arbitrarily:

▶ **Lemma 5** (Value closedness). *If $\Gamma_1 \vdash v \Leftrightarrow A$, then $\Gamma_2 \vdash v \Leftrightarrow A$.*

3.4 Semantics

We now introduce the call-by-value semantics of E_i using an environment-based operational semantics. The semantics employs a type-directed operational semantics (TDOS) [21]. In TDOS, in addition to a reduction relation, there is also a *casting* relation, which is introduced to reduce values based on the type of a given value.

$$\text{Frames}\quad F ::= [\,] : A \mid [\,] \mathbin{;} e \mid \{l = [\,]\} \mid [\,].l \mid [\,]\, e \mid v\, [\,] \mid [\,] \triangleright e$$

$$\boxed{v \vdash e \hookrightarrow e'} \qquad \textit{(Reduction)}$$

$$\textsc{Step-ctx}\ \frac{}{v \vdash \textsf{?} \hookrightarrow v} \qquad \textsc{Step-annov}\ \frac{v_1 \hookrightarrow_A v_1'}{v \vdash v_1 : A \hookrightarrow v_1'} \qquad \textsc{Step-merger}\ \frac{v \mathbin{;} v_1 \vdash e \hookrightarrow e'}{v \vdash v_1 \mathbin{;} e \hookrightarrow v_1 \mathbin{;} e'}$$

$$\textsc{Step-closure}\ \frac{}{v \vdash \{e\}^m : A \to B \hookrightarrow v \triangleright (\{e\}^m : A \to B)} \qquad \textsc{Step-box}\ \frac{v_1 \vdash e \hookrightarrow e' \qquad \neg\ \textsf{Closure}\ (v_1 \triangleright e)}{v \vdash v_1 \triangleright e \hookrightarrow v_1 \triangleright e'}$$

$$\textsc{Step-boxv}\ \frac{}{v \vdash v_1 \triangleright v_2 \hookrightarrow v_2} \qquad \textsc{Step-projv}\ \frac{}{v \vdash \{l = v_1\}.l \hookrightarrow v_1} \qquad \textsc{Step-eval}\ \frac{v \vdash e \hookrightarrow e'}{v \vdash F[e] \hookrightarrow F[e']}$$

$$\textsc{Step-beta}\ \frac{v_1 \hookrightarrow_{A_m} v_1'}{v \vdash (v_2 \triangleright (\{e\}^m : A \to B))\, v_1 \hookrightarrow (v_2 \mathbin{;} A_m^{v_1'}) \triangleright (e : B)}$$

Value Construction A_m^v

$$\boxed{\begin{array}{lcl} A_\bullet^v & = & v \\ \{l : A\}_\circ^v & = & \{l = v\} \end{array}}$$

$$\boxed{v \vdash e \hookrightarrow^* e'} \qquad \textit{(Multistep Reduction)}$$

$$\textsc{Multi-refl}\ \frac{}{v \vdash e \hookrightarrow^* e} \qquad \textsc{Multi-step}\ \frac{v \vdash e \hookrightarrow e' \qquad v \vdash e' \hookrightarrow^* e''}{v \vdash e \hookrightarrow^* e''}$$

Figure 4 Call-by-value reduction and multistep reduction of E_i.

Casting. The casting relation, shown in Figure 3, is defined on values. The casting relation is essentially the same as the relation in Huang et al.'s work [22]. The only difference is that, instead of having lambda abstractions as values, we now have closures as values. So the rules Casting-arrow and Casting-arrowTL change correspondingly to adapt to the new form of values. Rule Casting-int casts any integer value to itself under type Int. Rule Casting-top casts any value to a $\top$ under the top type. For merges, rule Casting-mergevL and rule Casting-mergevR cast one of the two branches under an ordinary type. These two rules can be viewed as value selectors for merges. The definition of ordinary types is the variant without the conditions marked in gray shown in Figure 1. In other words, ordinary types used in casting are those types that are not the top type or intersection types. With rule Casting-and, a value is cast under two parts of an intersection type respectively, and a merge is returned by combining the two results via the merge operator. Rule Casting-rcd casts a record value under a record type with the same label, and the result is a new record that is constructed from the result of casting the inner value under the inner type of the record type.

A closure $v \triangleright \{e\}^m : A \to B$ can be cast under an arrow type $C \to D$ to be a new value. If D is not top-like, then rule Casting-arrow casts the closure such that the return type is changed to D. Rule Casting-arrowTL ensures the determinism of casting by casting a closure to be a value generated by the value generator function ($A^\uparrow$) for top-like types. Without this rule, casting a merge of two closures via a top-like type can lead to different results. The definition of top-like types and the value generator are shown in the appendix.

Reduction. Reduction is shown in Figure 4. In the reduction relation $v \vdash e_1 \hookrightarrow e_2$, the environment v is a value. Since environments are involved in reduction, the definition of multi-step reduction is changed accordingly as shown in Figure 4. Briefly speaking, $v \vdash e_1 \hookrightarrow^* e_2$ means that e_1 can be reduced to e_2 by multiple steps under the same environment v, though the environment is possibly changed locally, during single-step reductions.

Synthesizing values by types. Rule Step-ctx reduces a query ? to the current environment. Rule Step-annov is the rule for annotated values, which triggers casting. In TDOS, casting uses type information from type annotations to guide the reduction to ensure determinism. Moreover, in E_i, casting also allows values to be fetched by types from the environment.

$$\text{\textsc{Multi-step}}\;\dfrac{\text{\textsc{Step-eval}}\;\dfrac{\text{\textsc{Step-ctx}}\;\dfrac{}{v \vdash\ ? \hookrightarrow v}}{v \vdash\ ? : A \hookrightarrow v : A} \qquad \dfrac{v \hookrightarrow_A v'}{v \vdash v : A \hookrightarrow v'}\;\text{\textsc{Step-annov}}}{v \vdash\ ? : A \hookrightarrow^* v'}$$

As shown in the derivation tree above, with $v \hookrightarrow_A v'$, we can conclude that $? : A$ will be evaluated to v' eventually. That is, the answer to a query that is equipped with a specific type, is the result of casting the current environment under that type. For example, suppose that the environment is $1 \mathbin{,\!,} \mathsf{true}$. Then the answer to the query $? : \mathsf{Int}$ is 1 while the answer to the query $? : \mathsf{Bool}$ is true.

Evaluating dependent merges. Similarly to the reduction strategy in calculi with intersection types and a merge operator, merges are evaluated from left to right in E_i. That is, for a merge $e_1 \mathbin{,\!,} e_2$, the right branch e_2 is evaluated only if the left branch e_1 is a value. However, since merges are dependent in E_i, the evaluation of e_2 relies on e_1. Specifically, for a merge $v_1 \mathbin{,\!,} e$ in which v_1 is already a value, rule Step-merger evaluates the right branch e under a new environment $v \mathbin{,\!,} v_1$ such that e can access not only the original environment v but also v_1. The following is an example of evaluating dependent merges, assuming an initial environment $\top$:

$$\begin{array}{ll}
 & \{x = 1\} \mathbin{,\!,} \{y = (? : \{x : \mathsf{Int}\}).x + 1\} \\
\hookrightarrow & \{x = 1\} \mathbin{,\!,} \{y = ((\top \mathbin{,\!,} \{x = 1\}) : \{x : \mathsf{Int}\}).x + 1\} \\
\hookrightarrow & \{x = 1\} \mathbin{,\!,} \{y = (\{x = 1\}).x + 1\} \\
\hookrightarrow & \{x = 1\} \mathbin{,\!,} \{y = 1 + 1\} \\
\hookrightarrow & \{x = 1\} \mathbin{,\!,} \{y = 2\}
\end{array}$$

The initial merge is evaluated to $\{x = 1\} \mathbin{,\!,} \{y = 2\}$. In every single step of the evaluation above, rule Step-merger is triggered and the right branch $\{y = \ldots\}$ is evaluated under $\top \mathbin{,\!,} \{x = 1\}$.

Closures and the beta rule. In our call-by-value semantics, when a function, which is not a value, is applied with a value, rule Step-closure transforms the function to a closure by assigning the current environment to it. Then rule Step-beta reduces the application, where the argument is cast first with the input type of the annotation of the closure. After that, the casting result is merged with the environment in the closure, and this merge becomes the local environment of a box. The body of the box is the body of the abstraction inside the applied closure. Thus, the body of the abstraction will be evaluated further under the new environment, which is a merge carrying the information from both the argument and the environment of the closure.

In rule Step-beta, the value $A_m^{v_1'}$ that is added to the environment is different according to the mode of the abstraction. For $v_2 \triangleright (\{e\}^{\bullet} : A \to B)$, $A_{\bullet} = A$ and $A_{\bullet}^{v_1'} = v_1'$, which is the result of casting v_1 with type A. If the mode is $\circ$, then the input type for the abstraction can only be a record type, say $\{l : A\}$. Thus for $v_2 \triangleright (\{e\}^{\circ} : \{l : A\} \to B)$, $\{l : A\}_{\circ} = A$ and $\{l : A\}_{\circ}^{v_1'} = \{l = v_1'\}$. That is, $v_2 \triangleright (\{e\}^{\circ} : \{l : A\} \to B)$ can accept a value of type A as input, and the value is given the name l such that it becomes a record during *runtime*. In this way, the body of the abstraction e can use the label to access the information in the record. When the evaluation context is the body of a box, rule Step-eval evaluates the local environment under the global environment until it is a value. After that, rule Step-box evaluates the body of the box under the local environment. A condition is set in rule Step-box to prevent closures from being reduced further. When the body is evaluated to a value, rule Step-boxv returns that value.

4 Determinism and Type Soundness

In this section, we show that the operational semantics of E_i is deterministic and type-sound. Unlike previous work on calculi with the merge operator, the typing contexts and the environments appearing in the theorems are generalized to arbitrary ones, since environments are first-class and can be manipulated explicitly in our system.

4.1 Determinism

To obtain the determinism of reduction, the determinism of casting is needed. With the help of consistency, any well-typed value that is cast under the same type results in a unique value.

▶ **Lemma 6** (Determinism of casting). *If* $\Gamma \vdash v \Rightarrow B$, $v \hookrightarrow_A v_1$, *and* $v \hookrightarrow_A v_2$, *then* $v_1 = v_2$.

With determinism of casting, we can prove the following generalized version of determinism, which states that if an expression e is well-typed under the type of the environment v, then the reduction result is the same.

▶ **Theorem 7** (Generalized determinism). *If* $\Gamma \vdash e \Leftrightarrow A$, $\mathsf{Top} \vdash v \Rightarrow \Gamma$, $v \vdash e \hookrightarrow e_1$, *and* $v \vdash e \hookrightarrow e_2$, *then* $e_1 = e_2$.

We cannot prove the standard theorem (where the typing context for e is Top and the environment v is $\top$) directly. The reason is that the environment is changed in rule Step-merger (from v to $v \mathbin{,,} v_1$) and rule Step-box (from v to v_1). If we prove the standard theorem directly, then the premises in the inductive hypothesis restrict the environment to be $\top$, which is not strong enough. Therefore, we generalize the theorem. Also note that the typing context for v can be any type in the theorem, since from Lemma 5 we know that the context for a well-typed value can be arbitrary. This fact is important for the proofs of metatheory. When a value is well-typed, we want it also to be well-typed under the context (say Top) appearing in the formalization of the theorem. Consider rule Step-box for example. The environment v_1 in the box is well-typed under the type of v, and it is also well-typed under Top, which meets the condition in the inductive hypothesis.

The standard determinism theorem can then be obtained as a corollary:

▶ **Corollary 8** (Determinism). *If* $\mathsf{Top} \vdash e \Leftrightarrow A$, $\top \vdash e \hookrightarrow e_1$, *and* $\top \vdash e \hookrightarrow e_2$, *then* $e_1 = e_2$.

4.2 Progress and Preservation

For progress and preservation, we need the following properties of casting:

▶ **Lemma 9** (Progress of casting). *If* $\Gamma \vdash v \Leftrightarrow A$ *then there exists* v' *such that* $v \hookrightarrow_A v'$.

▶ **Lemma 10** (Transitivity of casting). *If* $v \hookrightarrow_A v_1$ *and* $v_1 \hookrightarrow_B v_2$ *then* $v \hookrightarrow_B v_2$.

▶ **Lemma 11** (Consistency after casting). *If* $\Gamma \vdash v \Rightarrow C$, $v \hookrightarrow_A v_1$ *and* $v \hookrightarrow_B v_2$, *then* $v_1 \approx v_2$.

▶ **Lemma 12** (Preservation of casting). *If* $v \hookrightarrow_A v'$ *and* $\Gamma \vdash v \Rightarrow B$ *then* $\Gamma \vdash v' \Rightarrow A$.

These lemmas follow the logic of proving type soundness by Huang and Oliveira [21]. Lemma 9 states that a well-typed value can always be cast with its type. Lemma 10 ensures that casting results in the same value whether a value is cast directly or not. With this property and the determinism of casting, we can prove that the casting results of a value are consistent (Lemma 11), which ensures that casting preserves types (Lemma 12).

Progress and preservation. Similarly to generalized determinism, we have generalized progress and preservation lemmas. Both theorems are proved by induction on the typing judgment.

▶ **Theorem 13** (Generalized progress). *If* $\Gamma \vdash e \Leftrightarrow A$, *then*

- e *is a value, or*
- *for any value* v, *if* $\mathsf{Top} \vdash v \Rightarrow \Gamma$, *then there exists* e' *s.t.* $v \vdash e \hookrightarrow e'$.

▶ **Theorem 14** (Generalized preservation). *If* $\Gamma \vdash e \Leftrightarrow A$, $\mathsf{Top} \vdash v \Rightarrow \Gamma$, *and* $v \vdash e \hookrightarrow e'$, *then* $\Gamma \vdash e' \Leftrightarrow A$.

With the generalized theorems above, the standard progress and preservation theorem can then be obtained as corollaries:

▶ **Corollary 15** (Progress). *If* $\mathsf{Top} \vdash e \Leftrightarrow A$, *then* e *is a value, or there exists* e' *s.t.* $\top \vdash e \hookrightarrow e'$.

▶ **Corollary 16** (Preservation). *If* $\mathsf{Top} \vdash e \Leftrightarrow A$ *and* $\top \vdash e \hookrightarrow e'$, *then* $\mathsf{Top} \vdash e' \Leftrightarrow A$.

Type-safety. Combining generalized progress and preservation, we have generalized type safety where the multistep relation is involved. Basically, this generalized result indicates that under a well-typed environment, a well-typed expression will never get stuck.

▶ **Corollary 17** (Generalized type safety). *If* $\Gamma \vdash e \Leftrightarrow A$, $\mathsf{Top} \vdash v \Rightarrow \Gamma$, v *is a value, and* $v \vdash e \hookrightarrow^* e'$, *then either* e' *is a value or there exists* e'' *s.t.* $v \vdash e' \hookrightarrow e''$.

Thus, the standard type safety is an immediate corollary where the environment is instantiated to be the top value.

▶ **Corollary 18** (Type safety). *If* $\mathsf{Top} \vdash e \Leftrightarrow A$ *and* $\top \vdash e \hookrightarrow^* e'$, *then either* e' *is a value or there exists* e'' *s.t.* $\top \vdash e' \hookrightarrow e''$.

5 Encoding of λ_i

In this section, we show that E_i can encode the type system of the λ_i [32] via a type-directed translation. In other words, every well-typed expression in λ_i can be translated into a well-typed expression in E_i. We do not prove the operational correspondence because of the significant differences between the formulations of the semantics of λ_i and the environment-based semantics of E_i. However, as we discussed in Section 2, the E_i calculus enables first-class environments and dependent merges, which cannot be modelled by λ_i. The translation of λ_i to E_i demonstrates a few different things:

1. **Variables and lambda abstractions are encodable.** The first purpose of this translation is to show that standard variables and lambda abstractions can be fully encoded in E_i. Since λ_i has conventional lambda abstractions, the translation from λ_i to E_i demonstrates that lambdas are encoded in a general way.
2. **Non-dependent merges are encodable.** The second purpose of the translation is to show that non-dependent merges are also encodable. This encoding is not obvious since dependent merges introduce new disjointness restrictions that are not present in calculi such as λ_i. We show that a combination of E_i constructs can express all non-dependent merges without loss of generality.
3. **The E_i calculus subsumes λ_i.** Finally, with the two previous points, we can generally conclude that all typeable programs in λ_i can be encoded in E_i. So E_i is more powerful than λ_i. This is a desirable property since E_i is designed as a potential replacement for λ_i. Therefore, we should be able to express all the programs that are expressible in λ_i.

5.1 Syntax

The definitions of types, expressions, and typing contexts of λ_i are shown as follows:

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \\
\text{Expressions} & E & ::= & x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,,\, E_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Note that λ_i is a conventional lambda calculus with standard lambda abstractions and a standard context definition. Moreover, in λ_i contexts are *not* types, and environments are not first class.

5.2 Type-Directed Translation of λ_i to E_i

To utilize the information from λ_i contexts to construct expressions of E_i, we need to transform λ_i contexts to E_i contexts which are types. The translation function for contexts is defined as follows.

▶ **Definition 19** (Context translation). $|\Gamma|$ *transforms contexts of* λ_i *to types of* E_i.

$$
\begin{aligned}
|\cdot| &= \mathsf{Top} \\
|\Gamma, x : A| &= |\Gamma| \,\&\, \{x : A\}
\end{aligned}
$$

Figure 5 shows the typing rules of λ_i with an elaboration into E_i. Four of the rules are straightforward. Rule Styp-lit simply translates an integer to itself. Similarly, Rule Styp-top translates the top value to itself. Rule Styp-sub produces an expression by adding a type annotation, which is a super type of the type of the expression in the premise. Rule Styp-app simply combines the two elaborated expressions into an application in E_i.

$\boxed{\Gamma \vdash E : A \rightsquigarrow e}$ *(Typing with elaboration)*

$$\textsc{Styp-lit}\quad \frac{}{\Gamma \vdash i : \mathsf{Int} \rightsquigarrow i} \qquad \textsc{Styp-top}\quad \frac{}{\Gamma \vdash \top : \mathsf{Top} \rightsquigarrow \top} \qquad \textsc{Styp-var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow (? : \{x : A\}).x}$$

$$\textsc{Styp-sub}\quad \frac{\Gamma \vdash E : A \rightsquigarrow e \qquad A <: B}{\Gamma \vdash E : B \rightsquigarrow e : B} \qquad \textsc{Styp-abs}\quad \frac{\Gamma, x : A \vdash E : B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E : A \to B \rightsquigarrow \{e\}^{\circ} : \{x : A\} \to B}$$

$$\textsc{Styp-app}\quad \frac{\Gamma \vdash E_1 : A \to B \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 : A \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 : B \rightsquigarrow e_1\, e_2}$$

$$\textsc{Styp-merge}\quad \frac{\Gamma \vdash E_1 : A \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 : B \rightsquigarrow e_2 \qquad A * B \qquad \mathsf{fresh}\ x}{\Gamma \vdash E_1 \,,,\, E_2 : A \,\&\, B \rightsquigarrow \{x = ?\} \rhd (?.x \rhd e_1) \fatsemi ((? : \{x : |\Gamma|\}).x \rhd e_2)}$$

Figure 5 Type system of λ_i and its type-directed translation into E_i.

Encoding variables. Rule Styp-var uses labels to model variables. If a variable x has type A, then $x : A$ must appear in the context. This information from contexts is encoded as a record type $\{x : A\}$ in E_i. Thus, it becomes safe to annotate the query ? with $\{x : A\}$. To get the type of x, a record projection is performed to extract the value of type A from $\{x : A\}$.

Encoding lambda abstractions. Similarly, the type information of the bound variable x in a λ_i lambda abstraction is also translated to $\{x : A\}$. For any $\lambda x.\, E$ of type $A \to B$, rule Styp-abs encodes it as $\{e\}^{\circ} : \{x : A\} \to B$, which has type $A \to B$ instead of $\{x : A\} \to B$. In this way, it can accept values of type A instead of $\{x : A\}$. For example, $\lambda x.\, x$ with type $\mathsf{Int} \to \mathsf{Int}$ is translated to $\{(? : \{x : \mathsf{Int}\}).x\}^{\circ} : \{x : \mathsf{Int}\} \to \mathsf{Int}$, which can accept the integer 1 as input in an application.

Encoding non-dependent merges. Merges in λ_i are non-dependent and are encoded in an interesting way in E_i. For dependent merges, the global context should be disjoint with the type of the left branch. To prevent overlapping between $|\Gamma|$ and A, a fresh label x that does not appear in the existing types is picked to create a record $\{x : |\Gamma|\}$ that holds the current environment. This record becomes the context for the merge and is disjoint with A, since x is fresh and consequently A cannot contain a record with a field x. With the box construct, the merge is assigned the local environment $\{x = ?\}$. For the left branch of the merge, projection $?.x$ unwraps the context to take back the original context $|\Gamma|$. Similarly, unwrapping is needed for the right branch. However, since A appears in the typing scope for the right branch in a dependent merge, the annotation $\{x : |\Gamma|\}$ is needed for hiding A. In this way, only $|\Gamma|$ appears in the typing context of e_2.

Example. In λ_i, the merge $x\,,,\,\lambda y.\,y$ can have type $\mathsf{Int}\,\&\,(\mathsf{Int} \rightarrow \mathsf{Int})$ in the context $x : \mathsf{Int}$. This expression is translated to the following expression in E_i:

$$\{z = ?\} \triangleright (?.z \triangleright (? : \{x : \mathsf{Int}\}).x) \,_{9}\, ((? : \{z : \{x : \mathsf{Int}\}\}).z \triangleright \{(? : \{y : \mathsf{Int}\}).y\}^{\circ} : \{y : \mathsf{Int}\} \rightarrow \mathsf{Int})$$

where z is the fresh label that wraps the environment. This E_i expression infers $\mathsf{Int}\,\&\,(\mathsf{Int} \rightarrow \mathsf{Int})$ in the context $\{x : \mathsf{Int}\}$.

Type safety of the translation. The following result shows the type-safety of the translation, and that the type system of λ_i can be translated into E_i without loss of expressivity. Importantly, normal lambda abstractions and non-dependent merges are expressible in E_i.

▶ **Theorem 20** (Well-typed encoding of λ_i). *If* $\Gamma \vdash E : A \rightsquigarrow e$, *then* $|\Gamma| \vdash e \Rightarrow A$.

6 Related Work

First-class environments. First-class environments enable environments to be manipulated by programmers. Gelernter et al. [18] invented a programming language called Symmetric Lisp that enriches Lisp with a kind of first-class environment, which can be used to evaluate expressions. They argued using several examples that the first-class environments they defined generalize a variety of constructs including modules, records, closures, and classes. However, the formal semantics of the language is not included in their work. Miller and Rozas [29] also proposed an extension to the Scheme programming language. In their work, environments are created with `make-environment`, and a binary `eval` function is used to perform computations under a first-class environment. Jagannathan [23, 24] defined a dialect of Scheme called Rascal, in which two key operators related to first-class environments are introduced: `reify` that returns the current environment as a data object, and `reflect` which transforms data objects to an environment.

Queinnec and de Roure [36] present a form of first-class environments as an approach to share data objects for the Scheme programming language. Operators on environments, such as composition, importing, and exporting, are supported in their setting. Moreover, the first-class environments they proposed obey the quasi-static discipline [26] such that variables are either static or quasi-static during importing and exporting. Note that our treatment of variable names is similar to the quasi-static scoping approach [26] in some sense. To solve the issue of name capturing, in quasi-static scoping, a free variable has an *internal* name and an *external* name. The external name is for sharing variable bindings and is not α-convertible. The programmer has to resolve it before dereferencing. In our setting, the label x in the abstraction $\{?.x\}^{\bullet} : \{x : \mathsf{Int}\} \rightarrow \mathsf{Int}$ acts as an external name. In order to avoid ambiguities, the external names in quasi-static scoping must be different in their setting, which is similar to our approach where names are ensured to be different via disjointness.

All the work above is done in a dynamically typed setting. Regarding typed languages, there is little work on first-class environments, which are basically based on explicit substitutions [1]. Sato et al. [44, 45] introduced a simply typed calculus called $\lambda\varepsilon$ with environments as first-class values. In their work, full reduction is supported, and lambda abstractions allow local renaming of bounded variables to fresh names. Sato et al. proved some desirable properties, such as subject reduction, confluence and strong normalizability, for this calculus. First-class environments are called explicit environments in $\lambda\varepsilon$, which are sets of variable-value pairs. Moreover, there is an evaluation operation $e[\![a]\!]$ that evaluates the expression a under an environment e. This construct is similar to the box construct in E_i. However, reification and environment concatenation are not supported in his work. Nishizaki [47] proposed a similar calculus with first-class environments, in which a construct called *id* is introduced to return the current environment. This construct acts as reification and is similar to our queries, but Nishizaki's calculus does not support restriction. In E_i queries together with type annotations can retrieve parts of an environment, and model environment restriction. While there is an operator called extension, which can be viewed as a special case of concatenation in Nishizaki's work, the types do not accumulate. In contrast, environment concatenation in E_i is modelled via dependent merges with type information flowing from left to right. Subtyping is not included in existing type systems with environment types. In contrast, E_i supports subtyping and has a natural notion of subtyping of environments. As a result, it enables more applications. For instance, objects and inheritance can be modelled in E_i [5].

Module systems. Module systems [27] are a key structuring mechanism to build reusable components in modular programming. In ML-style languages, module systems serve as a powerful tool for data abstraction. Generally speaking, a module is a named collection of (dependent) declarations that aim to define an environment. Since dependent merges are supported in E_i, a simple form of modules is allowed by using records and merges in our work. For example, the record $\{M = \{x = 1\} \mathbin{,\!\!,} \{y = ?.x\}\}$ in E_i defines a module named M that contains dependent declarations. Conventionally, ML-style languages are stratified into two parts: a core language, which is associated with ordinary values and types; and a module language consisting of modules and module types (or signatures). In this way, modules are second-class since a module cannot be passed as an argument to a function. In E_i, a simple form of *first-class* modules is enabled via first-class environments. Therefore in our setting, modules can be created and manipulated on the fly. For instance, the above module M encoded as a record can be passed to a function, such that the values bound with x and y could be updated.

There is much work on getting around this stratification to enable first-class modules. One approach is to utilize dependent types. Harper and Mitchell proposed XML calculus [20] which is a dependent type system to formalize modules as Σ and Π types. After that, translucent sums [19] and singleton types [48] were present as extensions and refinement of the XML calculus. On the other hand, Rossberg et al. proposed the F-ing method [42] to encode the ML module system using System F_ω [3] rather than dependent types. Following the F-ing method, 1ML was proposed by Rossberg [41] in which core ML and modules are collapsed into one language. Compared with E_i, the calculi in this kind of work are more expressive due to the use of powerful type systems, where type declarations and abstract types are typically supported. However, expressions, declarations, and modules are separate in the syntax. In contrast, we demonstrate a new approach to enable a simple form of first-class modules via a unified syntax in our work. A variety of entities, including environments, records, declarations, and modules, are simply expressions in E_i.

Implicit calculi. Implicits are a mechanism for implicitly passing arguments based on their types, which are supported in Scala as a generic programming mechanism to reduce boilerplate code. Oliveira et al. [11] investigated the connection between Haskell type classes and Scala implicits. They showed that many extensions of the Haskell type class system can be encoded using implicits. After that, Oliveira et al. [12] synthesized the key ideas of implicits formally in a general core calculus that is called the implicit calculus. The implicit calculus supports a number of source language features that are not supported by type classes. In implicit calculi there are two kinds of contexts and/or environments: there are regular contexts (and environments) tracking variable bindings; and there is also an implicit environment, which tracks values that can be used to provide implicit arguments automatically. In E_i, we borrow the notion of a query, which enables type-based lookups on implicit environments, from the implicit calculus. While queries are used to query *implicit* environments by type in the implicit calculus, queries in E_i are applied directly to *runtime* environments and there is no distinction between implicit and regular environments.

Rouvoet [43] extended the work of Oliveira et al. and showed that the ambiguous resolution from the implicit calculus is undecidable. Following up on the earlier work on the implicit calculus [12], Schrijvers et al. [46] reformalized the ideas of implicits and presented a coherent and type-safe formal model, which supports first-class overlapping implicits and higher-order rules. Moreover, a more expressive unification-based algorithmic resolution, which is closely related to the idea of propositions as types [50], is described. While a highly complex mechanism is imposed to ensure coherence and the semantics is given by elaboration in their work, in E_i we adopt a TDOS to utilize the type information for guiding reduction and to enable determinism in a natural way. Odersky et al. [31] proposed the SI calculus. The SI calculus generalizes implicit parameters in Scala to implicit function types that have the form of $\mathsf{T}? \!\rightarrow \mathsf{T}$, which provides a way to abstract over the contexts consisting of running code. The idea of this generalization was inspired by an early draft of Schrijvers et al.'s work. Unlike the work of Schrijvers et al. and our work, SI lacks unambiguity. Thus a disambiguation scheme is needed in the implementation. While forms of implicit contextual abstraction are offered in the implicit calculi above, a form of contextual abstraction is also supported in E_i. Indeed, since environments are first-class values in E_i, one can easily abstract over the contexts by using abstractions. More recently, Marntirosian et al. [28] added modus ponens to subtyping to make resolution a special case of subtyping and to enable *implicit* first-class environments. Unlike E_i, the runtime environments in their work are still second class.

The merge operator. The merge operator was firstly proposed by Reynolds in the Forsythe language [38] to add the expressiveness for calculi with intersection types. Reynolds' merge operator is quite restrictive and does not allow, for instance, overloaded functions. Since then, several other researchers [8, 15, 32, 33] have removed restrictions and shown more applications of the merge operator. Dunfield [15] presents a powerful calculus with an unrestricted merge operator and an elaboration semantics that can encode various language features. While the elaboration semantics is type-safe, determinism or coherence [39] cannot be ensured. To enable determinism, a disjointness restriction on merges has been proposed in the work of Oliveira et al. [32]. In this work we borrow the idea of merges, intersection types and disjointness from previous work on the merge operator. Unlike previous work, our merges are dependent and E_i has operators to manipulate first-class environments that are not available in earlier calculi with the merge operator. In previous calculi, environments are not first class and the only operators supported on merges are concatenation and restriction.

Staged calculi and modal logic. Staging is a technique to separate the computations of a program, such that abstraction can be realized without loss of efficiency. Davies and Pfenning [14] proposed a type system that captures staged computation based on the intuitionistic variant of the modal logic S4 [35]. The modal necessity operator $\Box$ is introduced, and $\Box A$ represents the type of code that will be evaluated in an upcoming stage. At the term level, expressions of type $\Box A$ have the form $\mathsf{box}(e)$. Corresponding to the modal rule of necessitation, $\mathsf{box}(e)$ has type $\Box A$ if e has type A in the *empty* context. Later, after this work, the box construct is generalized by Nanevski et al. in the work of contextual modal type theory [30]. In this work, the box construct has the form $\mathsf{box}(\Psi.e)$ where Ψ is a context and e can utilize the information in Ψ. The construct $\mathsf{box}(\Psi.e)$ is similar to $e_1 \rhd e_2$ in

E_i in the sense that the context Ψ shadows the current context. Both constructs capture the dependence of expressions on contexts, in effect modelling data injection. However, since Ψ is a context, e in $\mathsf{box}(\Psi.e)$ can only utilize type information, whereas in $e_1 \triangleright e_2$, e_2 relies on the concrete environment information from the expression e_1 directly. Furthermore, in modal type theory contexts Ψ are defined in the usual way and are not types, nor are first class in the language. In contrast, contexts are types in E_i, and environments are first class values.

Abstract machines. Abstract machines, such as the SECD machine [25], Krivine's machine, the categorical abstract machine [10], and the CEK machine [16], are state transition systems that serve as a basis for the implementation of functional languages. Typically, a state in abstract machines is a tuple that contains an expression, an environment, and some other entities (such as stack and continuation) for reduction. Similarly, in E_i the semantics is an environment-based semantics, and closures are used to keep environments around during the reduction. However, abstract machines are models for lambda calculus, and thus they are not aimed at providing languages with first-class environments. In contrast, the E_i calculus supports first-class environments and operators that manipulate environments.

7 Conclusion

In this paper, we have presented a statically typed calculus called E_i, that supports the creation, reification, reflection, concatenation and restriction of first-class environments. The E_i calculus borrows disjoint intersection types and a merge operator from the λ_i [32] calculus, but employs them to model environments. In E_i, intersection types are used to model contexts, and disjointness is imposed to model (and generalize) the uniqueness of variables in an environment. However, unlike previous work, merges in E_i are dependent, which enables modelling dependent declarations. From *implicit calculi* [12, 31, 46], E_i borrows queries to synthesize the full current context (at the type level) and the entire current environment (at the term level), and to enable type-based lookups. We prove the determinism and type-soundness of E_i. Furthermore, we show that the type system of λ_i can be encoded by E_i via a type-directed translation. In other words, standard variables, lambda abstractions, and non-dependent merges are all encodable in E_i, enabling the E_i calculus to subsume λ_i. We also study an extension of the calculus with fixpoints. The E_i calculus, as well as the extension, and all the proofs presented in this paper have been formalized using Coq theorem prover.

As for future work, we are interested in extensions with more features. For example, we plan to investigate how to incorporate BCD subtyping [4]. With the merge operator and BCD subtyping, a powerful form of composition called nested composition [6] can be enabled. We would also like to extend the current calculus with polymorphism and show that abstract types can be encoded with the extended calculus. In this setting, since type variables could occur in contexts, we plan to use labels to model type variables, just like what we have done for term variables.

References

1 Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, 1989.

2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

3 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991. doi:10.1017/s0956796800020025.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

5 Xuan Bi and Bruno C. d. S. Oliveira. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

6 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

7 Luca Cardelli. Program fragments, linking, and modularization. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *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*, pages 266–277. ACM Press, 1997. doi:10.1145/263699.263735.

8 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.

9 Coq development team. The coq proof assistant. http://coq.inria.fr/.

10 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987. doi:10.1016/0167-6423(87)90020-7.

11 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.

12 Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 35–44. ACM, 2012. doi:10.1145/2254064.2254070.

13 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*, 2000.

14 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001. doi:10.1145/382780.382785.

15 Jana Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.

16 Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 314–325. ACM Press, 1987. doi:10.1145/41625.41654.

17 Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: An ir for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, pages 55–66. Association for Computing Machinery, 2019.

18 David Gelernter, Suresh Jagannathan, and Thomas London. 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*, pages 98–110. ACM Press, 1987. doi:10.1145/41625.41634.

19 Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, 1994.

20 Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993. doi:10.1145/169701.169696.

21 Xuejing Huang and Bruno C. d. S. Oliveira. A type-directed operational semantics for a calculus with a merge operator. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:32, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.26.

22 Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. Taming the merge operator. *Journal of Functional Programming*, 31:e28, 2021. doi:10.1017/S0956796821000186.

23 Suresh Jagannathan. Dynamic modules in higher-order languages. In Henri E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 74–87. IEEE Computer Society, 1994. doi:10.1109/ICCL.1994.288391.

24 Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Trans. Program. Lang. Syst.*, 16(3):456–492, 1994. doi:10.1145/177492.177578.

25 P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi:10.1093/comjnl/6.4.308.

26 Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 479–492. ACM Press, 1993. doi:10.1145/158511.158706.

27 David B. MacQueen. Modules for standard ML. In Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr., editors, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, pages 198–207. ACM, 1984. doi:10.1145/800055.802036.

28 Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. Resolution as intersection subtyping via modus ponens. *Proc. ACM Program. Lang.*, 4(OOPSLA):206:1–206:30, 2020. doi:10.1145/3428274.

29 James S. Miller and Guillermo Juan Rozas. Free variables and first-class environments. *LISP Symb. Comput.*, 4(2):107–141, 1991.

30 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–49, 2008.

31 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), 2017.

32 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.

33 Benjamin C Pierce. *Programming with intersection types and bounded polymorphism.* PhD thesis, University of Pennsylvania, 1991.

34 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

35 Dag Prawitz. *Natural deduction: A proof-theoretical study.* Courier Dover Publications, 2006.

36 Christian Queinnec and David De Roure. Sharing code through first-class environments. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 251–261. ACM, 1996. doi:10.1145/232627.232653.

37 Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 25:1–25:31, 2022.

38 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.

39 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.

40 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. Birkhauser Boston Inc., 1997.

41 Andreas Rossberg. 1ml - core and modules united. *J. Funct. Program.*, 28:e22, 2018. `doi:10.1017/S0956796818000205`.

42 Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of functional programming*, 24(5):529–607, 2014.

43 Arjen Rouvoet. Programs for free: Towards the formalization of implicit resolution in scala. Master's thesis, TU Delft, 2016.

44 Masahiko Sato, Takafumi Sakurai, and Rod M. Burstall. Explicit environments. *Fundam. Informaticae*, 45(1-2):79–115, 2001. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-05`.

45 Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *J. Funct. Log. Program.*, 2002, 2002. URL: `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-04.pdf`.

46 Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: stable and coherent implicits. *J. Funct. Program.*, 29:e3, 2019. `doi:10.1017/S0956796818000242`.

47 Shin-ya Nishizaki. Simply typed lambda calculus with first-class environments. *Publications of the Research Institute for Mathematical Sciences*, 30(6):1055–1121, 1994.

48 Christopher A Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)*, 7(4):676–722, 2006.

49 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

50 Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. `doi:10.1145/2699407`.

51 Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(3):1–61, 2021.

A Some Relations

A.1 Algorithmic Disjointness

$\boxed{A \sqcap B}$ *(COSTs)*

$$\textsc{Cost-Int}\ \frac{}{\mathsf{Int} \sqcap \mathsf{Int}} \qquad \textsc{Cost-AndL}\ \frac{A \sqcap C}{A \,\&\, B \sqcap C} \qquad \textsc{Cost-AndR}\ \frac{B \sqcap C}{A \,\&\, B \sqcap C} \qquad \textsc{Cost-RAndL}\ \frac{A \sqcap B}{A \sqcap B \,\&\, C} \qquad \textsc{Cost-RAndR}\ \frac{A \sqcap C}{A \sqcap B \,\&\, C}$$

$$\textsc{Cost-Arr}\ \frac{B \sqcap D}{A \to B \sqcap C \to D} \qquad \textsc{Cost-Rcd}\ \frac{A \sqcap B}{\{l : A\} \sqcap \{l : B\}}$$

Here we define a relation called COSTs (Common Ordinary Super Types), which is used to define algorithmic disjointness as following:

▶ **Definition 21** (Algorithmic Disjointness). $A *_a B \equiv \neg(A \sqcap B)$

The algorithmic disjointness is equivalent to the specification of disjointness (Definition 1).

▶ **Theorem 22** (Disjointness Equivalence). *$A *_a B$ if and only if $A * B$.*

A.2 Top-like Types

$\boxed{\rceil A \lceil}$ *(Top-like Types)*

$$\textsc{TL-top}\ \frac{}{\rceil \mathsf{Top} \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{TL-arr}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-rcd}\ \frac{\rceil B \lceil}{\rceil \{l : B\} \lceil}$$

A.3 Value Generator

▶ **Definition 23** (Value Generator). *$A^{\uparrow}$ generates a value for top-like type A.*

$$\begin{aligned} \mathsf{Top}^{\uparrow} &= \top \\ (A \to B)^{\uparrow} &= \top \triangleright (\{B^{\uparrow}\}^{\bullet} : A \to B) \\ (A \,\&\, B)^{\uparrow} &= A^{\uparrow} \,, B^{\uparrow} \\ \{l : A\}^{\uparrow} &= \{l = A^{\uparrow}\} \end{aligned}$$

B Fixpoints

In this section, we discuss an extension of E_i with fixpoints.

Expressions	$e ::= \dots \mid \mathsf{fix}\ A.e$
Values	$v ::= \dots \mid v \triangleright (\mathsf{fix}\ A.e : B)$

Syntax and typing. Expressions are extended with fixpoint $\mathsf{fix}\ A.e$ in which A is the type annotation. For values, closures are extended with boxes containing a fixpoint. Note that for $\mathsf{fix}\ A.e$ in a closure, an additional type annotation B is required. Rule Typ-fix is the typing rule for fixpoints, which is shown at the top of Figure 6. To make $\mathsf{fix}\ A.e$ well-typed, the body e needs to be checked under the context extended with A. Similarly to the typing rule for abstractions, there is also a disjointness condition $\Gamma * A$ to prevent ambiguities.

Casting and reduction. The extended casting and reduction rules for fixpoints are shown in Figure 6. Basically, $v \triangleright (\mathsf{fix}\ A.e : B)$ is cast with a supertype C and the result depends on whether C is top-like or not. If C is not a top-like type, then the casting result is $v \triangleright (\mathsf{fix}\ A.e : C)$. Otherwise, $v \triangleright (\mathsf{fix}\ A.e : B)$ is cast to a value generated by the value generator for C. This is similar to the treatment of casting abstractions for ensuring determinism. Note that C is required to be ordinary in rule Casting-fix and rule Casting-fixtl. This is to avoid overlapping with rule Casting-and when C is an intersection type.

For reduction, there are three rules for fixpoints. Rule Step-fix transforms $\mathsf{fix}\ A.e$ to a closure by assigning the current environment and giving an additional annotation to it. When $v_2 \triangleright \mathsf{fix}\ C.e : A \to B$ is applied to value v_1, rule Step-fixbeta "unwinds" the closure in the sense that the closure is put into the environment. In this way, when the application $(e : A \to B)\, v_1$ is evaluated, it can access and utilize the closure containing the fixpoint again. Note that the closure put in the environment is $v_2 \triangleright \mathsf{fix}\ C.e : C$ instead of $v_2 \triangleright \mathsf{fix}\ C.e : A \to B$. This is to ensure that the body e of the fixpoint is well-typed under the same context $\Gamma \,\&\, C$ for type preservation. Similarly, when a record projection is required, rule Step-fixproj "unwinds" the closure, and evaluates $(e : \{l : B\}).l$ under the environment that contains the fixpoint information.

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Extended Bidirectional Typing)*

$$\frac{\Gamma * A \qquad \Gamma \,\&\, A \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{fix}\ A.e \Rightarrow A}\ \textsc{Typ-fix}$$

$\boxed{v \hookrightarrow_A v'}$ *(Extended Casting)*

$$\frac{B <: C \qquad \neg\rceil C \lceil \qquad \mathsf{Ordinary}\ C}{v \rhd (\mathsf{fix}\ A.e : B) \hookrightarrow_C v \rhd (\mathsf{fix}\ A.e : C)}\ \textsc{Casting-fix}$$

$$\frac{B <: C \qquad \rceil C \lceil \qquad \mathsf{Ordinary}\ C}{v \rhd (\mathsf{fix}\ A.e : B) \hookrightarrow_C C^{\uparrow}}\ \textsc{Casting-fixtl}$$

$\boxed{v \vdash e \hookrightarrow e'}$ *(Extended Reduction)*

$$\frac{}{v \vdash \mathsf{fix}\ A.e \hookrightarrow v \rhd (\mathsf{fix}\ A.e : A)}\ \textsc{Step-fix}$$

$$\frac{}{v \vdash (v_2 \rhd \mathsf{fix}\ C.e : A \to B)\, v_1 \hookrightarrow (v_2 \fatsemi (v_2 \rhd \mathsf{fix}\ C.e : C)) \rhd (e : A \to B)\, v_1}\ \textsc{Step-fixbeta}$$

$$\frac{}{v \vdash (v_2 \rhd \mathsf{fix}\ A.e : \{l : B\}).l \hookrightarrow (v_2 \fatsemi (v_2 \rhd \mathsf{fix}\ A.e : A)) \rhd (e : \{l : B\}).l}\ \textsc{Step-fixproj}$$

Figure 6 Extended typing, casting, and reduction rules for E_i with fixpoints.

Determinism and type-soundness. The extension with fixpoints retains the properties of determinism and type-soundness. All the metatheory does not require significant changes for this extension and is formalized in the Coq theorem prover.