Imperative Compositional Programming

Type Sound Distributive Intersection Subtyping with References via Bidirectional Typing

WENJIA YE, The University of Hong Kong, China
YAOZHU SUN, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Compositional programming is a programming paradigm that emphasizes modularity and is implemented in the CP programming language. The foundations for compositional programming are based on a purely functional variant of System F with intersection types, called F_i^+, which includes distributivity rules for subtyping.

This paper shows how to extend compositional programming and CP with mutable references, enabling a modular, imperative compositional programming style. A technical obstacle solved in our work is the interaction between distributive intersection subtyping and mutable references. Davies and Pfenning [2000] studied this problem in standard formulations of intersection type systems and argued that, when combined with references, distributive subtyping rules lead to type unsoundness. To recover type soundness, they proposed dropping distributivity rules in subtyping. CP cannot adopt this solution, since it fundamentally relies on distributivity for modularity. Therefore, we revisit the problem and show that, by adopting *bidirectional typing*, a more lightweight and type sound restriction is possible: we can simply restrict the typing rule for references. This solution retains distributivity and an unrestricted intersection introduction rule. We present a first calculus, based on Davies and Pfenning's work, which illustrates the generality of our solution. Then we present an extension of F_i^+ with references, which adopts our restriction and enables imperative compositional programming. We implement an extension of CP with references and show how to model a modular live-variable analysis in CP. Both calculi and their proofs are formalized in the Coq proof assistant.

CCS Concepts: • **Software and its engineering** → **Imperative languages**; **Data types and structures**.

Additional Key Words and Phrases: Bidirectional Typing, Mutable References, Intersection Types, Distributive Subtyping, Type Soundness, Compositional Programming

ACM Reference Format:
Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2024. Imperative Compositional Programming: Type Sound Distributive Intersection Subtyping with References via Bidirectional Typing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 342 (October 2024), 30 pages. https://doi.org/10.1145/3689782

1 Introduction

Compositional programming [Zhang et al. 2021] is a programming paradigm that emphasizes modularity and is implemented in the CP programming language. The well-known *expression problem* [Wadler 1998] can be naturally solved with compositional programming. Compositional programming borrows ideas from both functional and object-oriented programming (OOP). The OOP idea of *family polymorphism* [Ernst 2001] is closely related to compositional programming. Prior research on family polymorphism [Aracic et al. 2006; Ernst et al. 2006; Nystrom et al. 2004, 2006; Zhang and Myers 2017] explored extensions to OOP that enable inheritance of complete

Authors' Contact Information: Wenjia Ye, The University of Hong Kong, Hong Kong, China, yewenjia@connect.hku.hk; Yaozhu Sun, The University of Hong Kong, Hong Kong, China, yzsun@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/10-ART342
https://doi.org/10.1145/3689782

hierarchies of classes, instead of just a single class. In compositional programming, complete sets of traits [Bi and Oliveira 2018; Ducasse et al. 2006] (which play a similar role to classes) can be composed. This is similar to the composition of whole hierarchies of classes in family polymorphism.

However, the programming style employed in compositional programming departs from traditional OOP programming, and is based on a form of *open* pattern matching. This is an influence from functional programming. Compositional programming fits within a more general and recent theme of work [Fan and Parreaux 2023; Jin et al. 2023; van der Rest and Poulsen 2022, 2023], which aims at addressing modularity problems of functional programming. In particular, in functional programming, both algebraic datatypes and functions, defined by pattern matching, are *closed* to extensions. Several researchers have noted this problem and have proposed language designs with constructs that enable a similar programming style based on pattern matching, but where definitions are *open*, and nonetheless checked for *exhaustiveness* of patterns. In compositional programming, compositional interfaces play a similar role to algebraic datatypes, declaring sets of constructors. Unlike algebraic datatypes, compositional interfaces are composable and extensible. Traits are used to implement sets of reusable patterns, playing a similar role to pattern matching definitions in functional programming. However, traits are also open and extensible.

The foundations for compositional programming are based on a variant of System F [Girard 1972; Reynolds 1974] with *intersection types* [Coppo et al. 1981; Pottinger 1980], called F_i^+ [Fan et al. 2022], which includes *distributive* subtyping rules [Barendregt et al. 1983] and a *merge operator* [Dunfield 2014; Reynolds 1997]. Distributive subtyping rules are needed to support *nested composition* [Bi et al. 2018], which is key to compositional programming, and enables the composition of sets of traits. For instance, the distributivity rule between intersection and function types is:

$$\frac{}{(A \to B_1) \& (A \to B_2) <: A \to (B_1 \& B_2)}\ \textsc{S-d}$$

So far, compositional programming, as well as work within the same theme [Fan and Parreaux 2023; Jin et al. 2023; van der Rest and Poulsen 2022, 2023], has focused on *purely* functional styles. The current implementation of CP and the core F_i^+ calculus is purely functional and lacks support for imperative features. However, the addition of imperative features, such as mutable references, can be helpful. For instance, CP supports objects. Adding mutable references would enable CP to model stateful objects. In addition, mutable references are useful to model certain programs, which are hard to model in a purely functional setting. For instance, in program analysis, we often need algorithms that operate on graphs. More generally, graph algorithms are common in several domains such as graph databases, grammars, finite state machines or transition systems. While in a purely functional setting, it is easy and natural to write programs on tree structures, graph structures are challenging, and not many techniques exist to deal with graph structures (but see [Erwig 2001; Oliveira and Cook 2012b]). The use of references provides a simple way to model graph structures. Cycles can be created using mutable references and detected using reference equality.

Our goal is to extend previous calculi for compositional programming with references, to broaden the scope of compositional programming to imperative programming. This will enable mutable objects, as well as programs that operate on graph structures. Unfortunately, there is a major obstacle to achieving this goal. Davies and Pfenning [2000] have argued that references are incompatible with distributive intersection subtyping, including the rule S-d. They showed that a calculus with standard distributive intersection subtyping and references leads to *type unsoundness*. Thus, since compositional programming fundamentally relies on distributive intersection subtyping, it seems, at first, that there is little hope of adding references to compositional programming.

In this paper, we propose a simple, and lighter, alternative restriction that enables references in the presence of distributive intersection subtyping, while preserving type soundness. Our restriction exploits *bidirectional typing* [Dunfield and Krishnaswami 2021], which distinguishes between two modes of typing: *checking* and *inference*. With bidirectional typing, an expression can be checked against many types, but typically it can only infer the most precise type. We exploit this distinction to restrict the typing rule for references, allowing only a rule in inference mode:

$$\frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Rightarrow \mathsf{Ref}\ A}$$

We show that, with a bidirectional type system with this rule, we can forbid the problems identified by Davies and Pfenning. Our restriction enables an unrestricted intersection introduction rule and distributivity. We present two type sound calculi with distributive intersection subtyping and references. Both calculi adopt our lighter restriction. The first one is a variant of the calculus presented in Davies and Pfenning's work. This calculus is useful to show the key ideas of our approach, and to illustrate that the solution can be applied to a wide range of calculi with intersection types. The second calculus is a variant of F_i^+ with references, and it is the calculus that underlies our new extension of the CP programming language. Using our new implementation of CP extended with references, in Section 2, we illustrate how graph structures are encoded by CP using a *live-variable analysis* example. Overall, the contributions of this paper are as follows:

- **Imperative Compositional Programming:** We show how to integrate references into a language with compositional programming, enabling highly modular imperative programs.
- **A lightweight restriction**, based on bidirectional typing, to enable a calculus featuring distributive intersection subtyping and references.
- **Two type sound calculi** that demonstrate the use of the restriction. The first calculus is a variant of Davies and Pfenning's calculus while the second is a variant of F_i^+ with references.
- **Coq formalization, CP implementation and examples.** Both calculi and their proofs in the paper are formalized in the Coq proof assistant. Furthermore, we have an implementation of CP, based on F_i^+ extended with references, that can type check and run examples for the paper. An extended version of the example is also provided in the artifact [Ye et al. 2024].

2 Compositional Programming with References

In this section, we provide background on the expression problem and compositional programming, and show how compositional programming and the CP language are extended with support for references. We illustrate this support via an example that modularly constructs components of an imperative programming language and its control-flow graph, enabling a live-variable analysis [Aho et al. 2007].

2.1 Background: The Expression Problem

The imperative programming language that we are going to model starts with a minimal set of language constructs. We first illustrate how one could model such language constructs in Java to motivate and demonstrate the expression problem [Wadler 1998]. The expression problem is a well-known dilemma in programming languages, which shows that some programming languages are good at achieving certain forms of extensibility but bad at some other forms of extensibility. The abstract syntax of the initial language is defined in Java as follows:

```
interface Exp { String print(); }
class Var extends Exp { String print() { ... } }  // variable: x, y, z, ...
class Ass extends Exp { String print() { ... } }  // assignment: x = y
class Seq extends Exp { String print() { ... } }  // sequential composition: p; q
```

`Exp` represents the interface for all language constructs, which only contains a pretty-printing method at the beginning. `Var`, `Ass`, and `Seq` implement the interface and contain different `print` implementations.

First dimension of extensibility: more language constructs. The initial language is so minimal that we can hardly write any useful programs. In particular, it does not even allow primitive values. To make the language more expressive, we may want to add more language constructs like booleans and while-loops. In Java, adding more language constructs is simply done by adding new classes:

```
class NewConstruct extends Exp { String print() { ... } }
```

The addition of language constructs is modular because we do not need to modify existing code. This demonstrates that Java has good support for this form of extensibility.

Second dimension of extensibility: more operations. However, it is hard to modularly add more operations other than pretty-printing in Java. Usually, we have to modify the Java `Exp` interface to:

```
interface Exp { String print(); T newOperation(); }
```

Of course, we need to do similar modifications to the classes that implement the interface to include the new operation. Changing existing code like this is *not* modular. Thus, Java is not good at this second form of extensibility. Such a modularity problem is known as the expression problem. As we shall see, compositional programming provides a natural solution to this dilemma of extensibility.

2.2 A Brief Introduction to Compositional Programming

The compositional programming paradigm [Zhang et al. 2021] is implemented by the CP language and introduces several new notions for modularity. The first notion that needs explanation is that of a *compositional interface*. A compositional interface consists of type signatures for a set of constructors, and plays an analogous role to algebraic datatype definitions in functional languages. For example, the same abstract syntax previously defined in Java can be rewritten in CP as follows:

```
type SeqSig<Exp> = {
  Var: String → Exp;          -- variable: x, y, z, ...
  Ass: String → Exp → Exp;    -- assignment: x = y
  Seq: Exp → Exp → Exp;       -- sequential composition: p; q
};
```

In this compositional interface, we declared three constructors for three language constructs. The type parameter `Exp`, delimited by angle brackets, is called a *sort* and must be the return type of every constructor. A sort abstracts over different kinds of operations that can be applied to the program. A simple example is the aforementioned pretty-printing operation. We can instantiate the sort with type `Print` as follows:

```
type Print = { print: String };
printSeq = trait implements SeqSig<Print> ⇒ {
  (Var     x).print = x;
  (Ass   x e).print = x ++ " = " ++ e.print;
  (Seq e1 e2).print = e1.print ++ "; " ++ e2.print;
};
```

The instantiated interface is implemented by a *compositional trait* [Bi and Oliveira 2018; Ducasse et al. 2006], which is a basic reusable unit in CP. A trait is by default in an anonymous form, such as `trait ... ⇒ { ... }`, but we can always assign it to a variable, for example `printSeq`, to give it a name. The pattern-matching-like syntax `(Var x).print = x` is called a *method pattern* and desugars into a record field with nested traits:

```
  Var = \x → trait ⇒ { print = x };
```

```
-- Extension 1: Compositional interface and trait for booleans and pretty-printing
type BoolSig<Exp> = {
  Lit: Bool → Exp;  -- literal: true, false
  Not: Exp → Exp;   -- negation: not x
};
printBool = trait implements BoolSig<Print> ⇒ {
  (Lit b).print = toString b;
  (Not e).print = "not " ++ e.print;
};
-- Extension 2: Compositional interface and trait for while-loops and pretty-printing
type LoopSig<Exp> = {
  WhileDo: Exp → Exp → Exp;  -- while (...) do { ... }
  DoWhile: Exp → Exp → Exp;  -- do { ... } while (...)
};
printLoop = trait implements LoopSig<Print> ⇒ {
  (WhileDo cond body).print =
    "while (" ++ cond.print ++ ") do { " ++ body.print ++ " }";
  (DoWhile body cond).print =
    "do { " ++ body.print ++ " } while (" ++ cond.print ++ ")";
};
```

Fig. 1. Adding more language constructs for booleans and while-loops in CP.

Such syntactic sugar allows programming in CP in a style resembling pattern matching in functional languages like Haskell and ML.

First dimension of extensibility: more language constructs. Although the initial language is too simple to be useful in isolation, it captures basic imperative constructs in a standalone manner and can be composed with other features later, enabling a modular style advocated in compositional programming. What we expect is to add more language constructors in a modular way like in Java. The extension of language constructs is similarly easy in CP. Instead of new classes, we create a compositional interface and some traits implementing the existing operation for new constructs. For example, we add two more compositional interfaces and traits for booleans and while-loops respectively in Figure 1.

Second dimension of extensibility: more operations. We have shown the second dimension of the expression problem is difficult to solve in mainstream OOP languages like Java. In CP, the solution to the second dimension is trivial. We will show how CP solves the expression problem next.

2.3 Solving the Expression Problem: Adding CFG Construction

Extending CP with imperative features. The original formulation of CP [Zhang et al. 2021] is purely functional and lacks support for imperative programming constructs, such as mutable references. In this work, we extend CP itself with references and four related operations:

(1) **`ref`** `e` creates a reference cell with an initial value `e`.
(2) `!e` reads the value stored in the reference cell `e` (a.k.a. dereference).
(3) `e1 := e2` writes the value `e2` to the reference cell `e1` (a.k.a. assignment).
(4) `e1 >> e2` executes the imperative statement `e1` and then returns `e2` (a.k.a. sequencing).

To illustrate the importance of imperative features in compositional programming, we present an example of a data-flow analysis. The goal is to compute the set of variables that are live at each

```
graphSeq = trait implements SeqSig<Graph> ⇒ {
  (Var     x).graph = let r = ref (newNode ("Var " ++ x) [] [x]) in
                      mkPair r r;
  (Ass   x p).graph = let r = ref (newNode ("Ass " ++ x) [x] []) in
                      addAdj (snk p) [r] >>
                      mkPair (src p) r;
  (Seq p1 p2).graph = addAdj (snk p1) [src p2] >>
                      mkPair (src p1) (snk p2);
};
newNode (s: String) (def: [String]) (use: [String]) = {
  name = s;                   adj = ref ([] : [Ref Node]);
  def = def;                  use = use;
  IN  = ref ([] : [String]);  OUT = ref ([] : [String]);
};
addAdj (x: Ref Node) (ys: [Ref Node]) = !x.adj := !(!x.adj) ++ ys;
```

Fig. 2. Building a CFG for the initial language constructs.

program point. The live-variable analysis is done on a control-flow graph (CFG) constructed from the program. Without mutable references, it is inconvenient to construct a CFG and perform some analysis in CP. The first difficulty is how to represent the directed edges in the CFG. Since a while-loop may introduce a cycle in the graph, the connected nodes should be represented by references instead of values. When traversing a cyclic graph, we need to detect cycles using reference equality to avoid infinite loops. In addition, we also need flag variables to indicate whether an analysis has converged.

Representation of the CFG. In a CFG, each node represents a program segment, whose edges are directed and represent possible execution paths. The directed edges are implemented as an adjacency array stored in a node. The whole graph is represented by a pair of nodes: one for the source, and the other for the sink. The definitions are as follows:

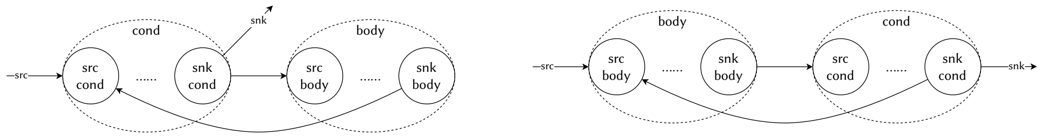
```
interface Node {
  name: String;       adj: Ref [Ref Node];
  def: [String];      use: [String];      -- variables defined and used
  IN:  Ref [String];  OUT: Ref [String];  -- for live-variable analysis
};
type Graph = { graph: PairRefNode };  -- (src, snk)
```

`Node` uses the keyword **`interface`** instead of **`type`** because it is a recursive type – the `adj` field uses `Node` itself. Recursive types are not supported in the work by Zhang et al. [2021]. Our enhanced implementation of CP supports recursive types, via **`interface`** declarations, based on the work by Zhou et al. [2022]. The adjacency array can be updated during the construction of the CFG, so the `adj` field is a reference. In other words, `adj` is similar to a mutable field in other languages. The element of the adjacency array is also a reference – otherwise, the same node can have multiple copies in its predecessors, and cycles in the graph cannot be represented.

CFG construction for the initial language. In CP, adding a new operation is as easy as adding new language constructs – all that we need is to create more traits. In Figure 2, we create one more trait for the initial language. We instantiate `Exp` with type `Graph` in the trait `graphSeq` and implement the CFG construction. Both `Var` and `Def` are mainly creating new CFG nodes, while the task of `Seq` is to add a directed edge from the sink node of `p1`'s graph to the source node of `p2`'s graph.

```
graphBoolLoop = trait implements BoolSig<Print ⇒ Graph> & LoopSig<Graph> ⇒ {
  [self]@(Lit b).graph = let r = ref (newNode self.print [] []) in mkPair r r;
         (Not e).graph = e.graph;
  (WhileDo cond body).graph = addAdj (snk cond) [src body] >>
                              addAdj (snk body) [src cond] >>
                              mkPair (src cond) (snk cond);
  (DoWhile body cond).graph = addAdj (snk body) [src cond] >>
                              addAdj (snk cond) [src body] >>
                              mkPair (src body) (snk cond);
};
```

Fig. 3. Building a CFG for booleans and while-loops.



(a) `while (cond) do { body }` (b) `do { body } while (cond)`

Fig. 4. CFG examples for while-loops.

In `newNode`, we initialize the adjacency array to be empty. Some extra type annotations are needed there because, by default, an empty array `[]` is inferred to have type `[`**`Bot`**`]`. Then **`ref`** `[]` has type **`Ref`** `[`**`Bot`**`]` and cannot be coerced into type **`Ref`** `[`**`String`**`]`, for instance, because the former is not a subtype of the latter. The `addAdj` function adds more nodes to the adjacency array, to add more directed edges that start from a certain node. Since the function body is merely an assignment statement, `addAdj` returns the unit value.

CFG construction for booleans and while-loops. As mentioned in Section 2.2, adding new language constructs is easy. The implementation for both booleans and while-loops is shown in Figure 3. Of course, if desired, we could also separate the implementation into two traits. Compositional programming allows finer- or coarser-grained traits for different scenarios. The boolean part does not affect control flow, but shows CP's ability to modularly inject *dependencies* between different traits [Zhang et al. 2021]. Instead of directly creating a string for the node identifier, as we did in `graphSeq`, we call `print`. Although `print` is not defined in `graphBoolLoop`, we state the dependency on `print` by instantiating `BoolSig` with `<Print ⇒ Graph>`. Thus we can use `self.print` in `(Lit b).graph`. At composition time (see Section 2.4), an implementation of `Print` must be provided to satisfy the dependency. The CFGs built for while-loops are shown in Figure 4. The two variants of CFGs are similar except that the sink node for `while (cond) do { body }` is not in `body`, but in `cond`, since we have to check if the condition holds before quitting the loop.

2.4 Putting Everything Together: Distributive Intersection Types in Action

After we have the five traits that implement pretty-printing and CFG construction for the seven existing language constructs, we may combine them together. Our goal is graphically illustrated in Figure 5. The difficulty here is how to nestedly compose multiple operations (in this case, `print` and `graph`) within the same constructor. In CP, such nested trait composition [Bi et al. 2018] is achieved by the merge operator [Dunfield 2014; Reynolds 1997], which is denoted by a comma:

```
type ProgSig<Exp> = SeqSig<Exp> & BoolSig<Exp> & LoopSig<Exp>;
merged: ProgSig<Print&Graph> = new printSeq,printBool,printLoop,graphSeq,graphBoolLoop;
```

Without nested composition and distributivity (as indicated by the type annotation), `merged` would have type `ProgSig<Print> & ProgSig<Graph>` and contain duplicate fields for each constructor. After desugaring method patterns, `merged` would be:

```
merged = {
  Var = \x → trait ⇒ { print = x };
  Var = \x → trait ⇒ { graph = let r = ... in mkPair r r };
  -- other constructors are similar
};
```

However, what we actually expect is, as shown in Figure 5, to have only one `Var` field that serves as a constructor that can be used for both pretty-printing and CFG construction:

```
merged = {
  Var = \x → trait ⇒ { print = x; graph = let r = ... in mkPair r r };
  -- other constructors are similar
};
```

In other words, we desire `merged` to be coerced into the type `ProgSig<Print&Graph>`. The subtyping relation between `ProgSig<Print> & ProgSig<Graph>` and `ProgSig<Print&Graph>` is valid because intersection types in CP are distributive over function, trait, and record types.

Encoding of traits. It is helpful to understand how traits are encoded in CP, to understand how the distributivity of traits works. As we shall see in Section 4, F^+_{im} does not include any OOP-related constructs. This is because these language constructs are encoded using functions and records in F^+_{im}. For example, a trait in CP is desugared to a function, where the argument of the function is the self-reference of the trait [Bi and Oliveira 2018; Zhang et al. 2021]. This encoding of traits follows the denotational model of inheritance by Cook and Palsberg [1989]. A simple example illustrating the encoding is given next:

```
t1 = trait ⇒ { x = 1 };           -- is desugared to:
t1' = \(self: Top) → { x = 1 };   -- having type: Top → {x: Int}

t2 = trait [self: {x: Int}] ⇒ { y = self.x };  -- is desugared to:
t2' = \(self: {x: Int}) → { y = self.x };      -- having type: {x: Int} → {y: Int}
```

Thus, the nested composition of two traits is essentially merging two functions, powered by the distributive intersection subtyping over functions. In other words, distributivity over trait types boils down to distributivity of intersections over function types, and the rule rule S-D. For example, the merge `t1',t2'` has type

```
(Top → {x: Int}) & ({x: Int} → {y: Int})
```

which is, via distributivity, a subtype of

```
Top & {x: Int} → {x:Int} & {y: Int}  -- or equivalently, {x: Int} → {x: Int; y: Int}
```

Back to CP's trait encoding, the function type above corresponds to `Trait<{x: Int} ⇒ {x: Int; y: Int}>`. That is, the trait itself contains two fields `x` and `y` while assuming the self-reference has type `{x: Int}`. We refer the readers who are interested in the encoding of other constructs like compositional interfaces to previous work [Zhang et al. 2021].

2.5 Live-Variable Analysis

Let us go back to the topic of data-flow analysis. Since we can already build a CFG from a given program, it should be easy to do various analyses based on the CFG. Here we pick live-variable analysis, which computes the set of variables that are live at each program point. As shown in Figure 6, we continue making use of imperative features in CP to make the analysis possible. The

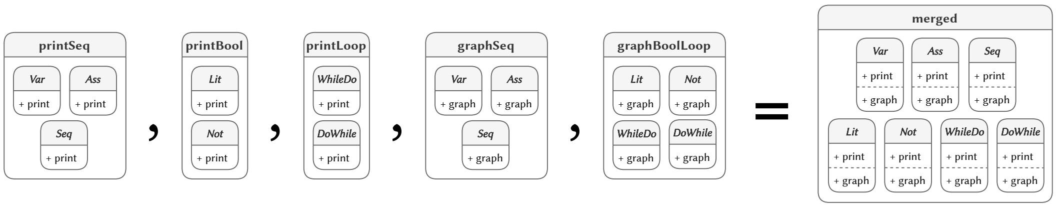


Fig. 5. Nested trait composition in CP.

algorithm is standard and can be found in any textbook on program analysis (see, for instance, Aho et al. [2007]). The analysis is performed in a backwards manner, starting from the sink node of the whole CFG and propagating the live variables back to the source node. The `IN` and `OUT` states represent the set of live variables corresponding to the entry and exit of each node, respectively. The basic idea of the `analyze` function is to iteratively update the `IN`/`OUT` states of each node until no `IN` is changed anymore. The transfer functions for `IN`/`OUT` in each iteration are:

$$\texttt{IN}[P] = \texttt{use}_P \cup (\texttt{OUT}[P] \setminus \texttt{def}_P)$$
$$\texttt{OUT}[P] = \bigcup_{Q \in \texttt{adj}_P} \texttt{IN}[Q]$$

The live variables at the entry of a node P are those used in P plus the live variables at the exit of P that are not defined in P. The live variables at the exit of P are the union of those at the entry of all nodes that are successors of P. Since `IN`/`OUT` need to be updated iteratively, they are modeled as references in CP, or mutable fields in other languages. Some implementations of set operations on arrays like `union` and `difference` are omitted for brevity. Note that the notation `adj!!i` is used to access the i-th element of the array `adj`. The `visited` set is used to avoid repeated visits to the same node as well as infinite loops if the CFG is cyclic. The set is cleared before a new iteration. The flag variable `changed`, in the first line of `analyzeLiveVar`'s body in Figure 6, is also a reference. It is initialized to store the boolean value **`false`** at the beginning of each iteration and will be updated with **`true`** if there is any change on the `IN` states. The `analyzeLiveVar` function is essentially a loop that checks if the dereferenced value `!changed` is **`true`**. If so, it will start a new iteration; otherwise, the loop ends. A sample program is constructed in the last part of Figure 6. The **`open`** keyword is used to open a record, which brings all constructors in `merged` into the current scope. Thus we can directly use, for example, `Seq` instead of `merged.Seq`. Then we call `analyzeLiveVar` on the program to compute the live variables, which are stored in `IN`/`OUT` fields of each node.

In conclusion, CP extended with mutable references enables *imperative* compositional programming, as showcased by our modular live-variable analysis example. Some more complicated algorithms that usually depend on mutability, such as program analysis, are made possible by the imperative features. What is more, distributive intersection types can interact well with mutable references in CP, resulting in a type sound language.

2.6 CP Implementation and Extended Example

All the examples presented in this section can run in our CP implementation. Note that we extended the original implementation by Zhang et al. [2021]. In particular, we added references and some other features such as recursive types [Zhou et al. 2022]. In the original design of CP, the implementation works by elaborating CP into a core calculus, called F_i^+ [Fan et al. 2022], which is based on *disjoint intersection types* [Oliveira et al. 2016] and *disjoint polymorphism* [Alpuim et al. 2017] and distributive subtyping. Our implementation works similarly, except that we extend F_i^+ with additional features. In Section 4, we formalize an extension of F_i^+ with references, called F_{im}^+, which serves as a foundation for our implementation. The elaboration from source language constructs (such as traits and

```
analyze (x: Ref Node) (changed: Ref Bool): () = if isVisited x then () else
  visited := !visited ++ [x] >>
  letrec go (i: Int): [String] =
    let adj = !(!x.adj) in
    if i == #adj then [] else analyze (adj!!i) changed >>
                              union !(!(adj!!i).IN) (go (i+1)) in
  !x.OUT := go 0 >>
  let oldIN = !(!x.IN) in
  !x.IN := union !x.use (difference !(!x.OUT) !x.def) >>
  if equal oldIN !(!x.IN) then () else changed := true;
analyzeLiveVar (prog: Graph): () =
  let changed = ref true in
  letrec go (_: ()): () =
    if !changed then changed := false >>
                     clearVisited () >>
                     analyze (src prog) changed >>
                     go ()
    else () in
  go ();
prog = open merged in
  Seq (Ass "x" (Lit true))                      -- x = true;
 (Seq (Ass "y" (Lit false))                     -- y = false;
 (Seq (WhileDo (Var "x") (Ass "x" (Var "y")))  -- while (x) do { x = y };
      (Ass "z" (Not (Var "x")))));              -- z = not x
analyzeLiveVar prog  -- main body
```

Fig. 6. Live-variable analysis based on the CFG.

compositional interfaces) is treated in previous work [Bi and Oliveira 2018; Zhang et al. 2021]. The addition of references is mostly orthogonal to source language constructs and has little impact on the elaboration. In other words, source language constructs like assignments or reference creation map directly into similar constructs in F^{+}_{im}. Thus, they are trivial to add to the elaboration step. One significant change in our implementation of CP is that CP now supports compilation into JavaScript. The previous implementation was based on an interpreter. Our extended implementation of CP is available in the artifact.

Extended example. An extended version of the example presented in this section is also available in our artifact. All code that is omitted for space reasons in this section can be found there. In the extended example, we add more language constructs like numeric literals, more operations on booleans and integers, and if-then-else statements. We also implement a depth-first traversal of the CFG, which can print the live variables at each program point.

3 A Type Sound Calculus with Distributive Subtyping and References

This section proposes a simple restriction for calculi with distributive subtyping, intersection types and references based on bidirectional typing. This restriction is lighter than the one proposed by Davies and Pfenning [2000], since it allows distributive subtyping in the presence of references. In addition, it does not require a value restriction. Since in CP distributive subtyping is fundamental, our lighter restriction enables the design of calculi that support compositional programming in the presence of references, as shown in Section 4. We start by revisiting the type unsoundness problem identified by Davies and Pfenning, and then present our solution.

$$\frac{}{A_1 \,\&\, A_2 <: A_1}\ \textsc{S-AndL} \qquad \frac{}{A_1 \,\&\, A_2 <: A_2}\ \textsc{S-AndR} \qquad \frac{A <: B_1 \qquad A <: B_2}{A <: B_1 \,\&\, B_2}\ \textsc{S-And}$$

$$\frac{B <: A \qquad A <: B}{\mathsf{Ref}\ A <: \mathsf{Ref}\ B}\ \textsc{S-Ref} \qquad \frac{}{(A \to B_1) \,\&\, (A \to B_2) <: A \to (B_1 \,\&\, B_2)}\ \textsc{S-D}$$

Fig. 7. Declarative subtyping of Davies and Pfenning [2000] (excerpt).

```
let x = ref 1 : Ref Nat & Ref Pos in
let y = (x := 0) in
let z = !x in z : Pos
 ↪ o = 1; let y = (o := 0) in
          let z = !o in z : Pos
 ↪ o = 0; let z = !o in z : Pos
 ↪ o = 0; 0 : Pos
```
(a)

```
let x = (λx. ref 1) unit : Ref Nat & Ref Pos in
let y = (x := 0) in
let z = !x in z : Pos
 ↪ let x = ref 1 : Ref Nat & Ref Pos in
    let y = (x := 0) in let z = !x in z : Pos
 ↪* o = 0; let z = !o in z : Pos
 ↪* o = 0; 0 : Pos
```
(b)

Fig. 8. Counter-examples identified by Davies and Pfenning [2000].

3.1 Problematic Interaction between References and Distributive Subtyping

Davies and Pfenning [2000] proposed a calculus and type assignment system for a language with intersection types and references. As usual for most type assignment systems, expressions are allowed to have multiple types. For instance, 1 can have as type Nat or Pos, among others. In their subtyping relation, they use standard rules for intersection and reference types in Figure 7. Note that subtyping for reference types (rule S-ref) is invariant ($A <: B$ and $B <: A$). The distributivity rule S-d states that an intersection of two functions with the same input type is a subtype of a function that returns the intersection of the output types. Davies and Pfenning [2000] concluded that general intersection types are unsound with mutable references. Next we will illustrate why type unsoundness happens using their examples.

Type unsoundness from intersection introduction. We show some typing rules of Davies and Pfenning [2000] that are used to illustrate the problem next:

$$\frac{\Sigma;\Gamma \vdash e : A \qquad A <: B}{\Sigma;\Gamma \vdash e : B}\ \textsc{Pty-sub} \qquad \frac{\Sigma;\Gamma \vdash e : A}{\Sigma;\Gamma \vdash \mathsf{ref}\ e : \mathsf{Ref}\ A}\ \textsc{Pty-ref} \qquad \frac{\Sigma;\Gamma \vdash e : \mathsf{Ref}\ A}{\Sigma;\Gamma \vdash !e : A}\ \textsc{Pty-def}$$

$$\frac{\Sigma;\Gamma \vdash e : A \qquad \Sigma;\Gamma \vdash e : B}{\Sigma;\Gamma \vdash e : A \,\&\, B}\ \textsc{Pty-intro} \qquad \frac{\Sigma;\Gamma \vdash e_1 : \mathsf{Ref}\ A \qquad \Sigma;\Gamma \vdash e_2 : A}{\Sigma;\Gamma \vdash e_1 := e_2 : \mathsf{Unit}}\ \textsc{Pty-set}$$

Note that, in our presentation of the two examples, we first present the original program by Davies and Pfenning in the first 4 lines, and then present reduction steps to illustrate what happens during execution. The first problematic program that Davies and Pfenning illustrate is in Figure 8a. In this program, 1 has type **Pos**. Moreover, since **Pos** is a subtype of **Nat**, by rule Pty-sub, 1 can have type **Nat** as well. Then, by rule Pty-ref, the expression **ref** 1 can have type **Ref Pos** and also type **Ref Nat**. Finally, because of rule Pty-intro, the expression **ref** 1 can be typed as the intersection type **Ref Nat** & **Ref Pos**. From the subtyping rule S-AndL for intersection types, we know that **Ref Nat** & **Ref Pos** is a subtype of **Ref Nat**. Thus, x can be assigned a **Nat** number 0. Because x has type **Ref Nat** & **Ref Pos**, and it can be upcast to **Ref Pos** via the subsumption rule, then z can have type **Pos**. Therefore, this program is typeable. However, when we run the program, z is going to be 0. In the first step, a new cell is created with value 1 referenced by the location o. Note that a semicolon (;) is used to separate the cell (o = 1) and the program. Then a value 0 is assigned to that location. After that, we read the content and annotate the value to have type **Pos**. Obviously, 0 can not be typed as **Pos**. So typing is unsound. In essence, the assignment in the program is problematic since `x` also has type **`Ref Pos`**, but **`ref`** `0` should not have that type, since `0` is not a positive number. In this program the intersection introduction rule plays an important role because it allows **`ref`** `1` to be typed with two different reference types.

Type unsoundness from distributivity. A second problematic program that Davies and Pfenning illustrate is in Figure 8b. This program is nearly the same as the first one, except for the first line, where **`ref`** `1` is replaced by `(λx.`**`ref`** `1)` **`unit`**. The expression `(λx.`**`ref`** `1)` can be typed with `(`**`Unit`** `→` **`Ref Nat`**`) & (`**`Unit`** `→` **`Ref Pos`**`)`. From subtyping distributivity (rule S-D), `(`**`Unit`** `→` **`Ref Nat`**`) & (`**`Unit`** `→` **`Ref Pos`**`)` <: **`Unit`** `→ (`**`Ref Nat`** `&` **`Ref Pos`**`)` holds. Therefore, type **`Unit`** `→ (`**`Ref Nat`** `&` **`Ref Pos`**`)` can be used to type check `(λx.`**`ref`** `1)`. Then the lambda body **`ref`** `1` can be checked as **`Ref Nat`** `&` **`Ref Pos`**. This expression exploits the subtyping distributivity rule, to also create a variable `x` with the same type as the `x` in the first program. After one step of reduction, the program in Figure 8b becomes essentially the program in Figure 8a. Then, using the same line of reasoning for the first program, we are able to conclude that this program also leads to type unsoundness.

It is worthwhile noting, that while we show counter-examples using **`Nat`** and **`Pos`**, the counter-examples can be adapted to many other settings, as Davies and Pfenning observe. For instance, suppose that the calculus contains a canonical top value ⊤, which can only be typed with ⊤, then we can easily produce similar counter-examples by replacing **`Nat`** by ⊤, and `0` by ⊤. Another example would be two record types A and B where $A <: B$. In this case, A would play the role of **`Pos`** and B would play the role of **`Nat`**. Then we could replace `1` and `0` by two records of type A and B. Thus, it is easy to adapt the counter-examples to various other settings, and obtain similar counter-examples to type soundness.

3.2 Davies and Pfenning's Solution

Davies and Pfenning [2000] suggested that the intersection introduction rule, and the distributivity rule used in subtyping are the causes of type unsoundness. To address this problem they proposed two restrictions: 1) a *value restriction* on intersection introduction; and 2) *removing distributivity* from subtyping. Their value restriction is inspired by the restriction employed in ML [Wright 1995] to solve the unsoundness problem of parametric polymorphism. Concretely, only values are allowed in intersection introduction:

$$\frac{\Sigma; \Gamma \vdash v : A \qquad \Sigma; \Gamma \vdash v : B}{\Sigma; \Gamma \vdash v : A\&B}$$

Now, we can consider the first program again. With the restricted intersection introduction rule, the first program fails to pass the type checker. Even though **`ref`** `1` can be typed with **`ref Nat`** and **`Ref Pos`**, **`ref`** `1` is not a value. In standard calculi with references, the values denoting references can only be locations. Therefore, the first program is rejected. More generally, type unsoundness cannot be triggered by this restricted intersection introduction rule.

However, we still need to deal with the second program. For lambdas, `(λx.`**`ref`** `1)` is a value, so the value-restricted intersection introduction can still be used in this case, and the program is still well-typed. In other words, the distributivity rule on function types can escape the value restriction. Therefore, Davies and Pfenning remove distributivity (rule S-D) from subtyping. Without distributivity, then the program is rejected.

While, with their restrictions, Davies and Pfenning obtain a type sound system, this comes at the cost of expressive power. In particular, we cannot adopt their restriction of removing distributivity in subtyping in compositional programming. As we have seen in Section 2, distributivity plays a fundamental role in CP to model nested composition, and enable a high degree of modularity. Thus we take a different approach in our work, and propose a lighter restriction, which is able to retain both distributivity and an unrestricted intersection introduction rule.

3.3 Our Solution: Bidirectional Typing to the Rescue

While Davies and Pfenning identified intersection introduction and distributivity for the type unsoundness problems, we believe that these features are not necessarily a problem. Indeed, we can design a type system and calculus with references and both features, which is type sound.

One reason for the two counter-examples to exist is because expressions can infer multiple types. This is expected in a *type assignment system* (TAS), which is often used to *specify* type systems. For instance, in the TAS proposed by Davies and Pfenning, the expression 1 can have multiple types, including Nat or Pos. However, implementations often cannot use a TAS, since a TAS may not be syntax-directed and can guess types. Moreover, while a TAS is often specified *without* type annotations, for many type systems (for instance those with undecidable type inference) we must specify some annotations. Thus, implementations have to adopt other approaches, such as *bidirectional typing* [Dunfield and Krishnaswami 2021]. For instance, in Davies and Pfenning's work, they adopt bidirectional typing for algorithmic typing.

For algorithmic typing and concrete implementations, we would expect that type inference is more (if not completely) *deterministic*. We would usually expect that an inferred type is unique, and that type annotations are needed in some parts of the program. In other words, we do not expect to run the type checker twice on the same program and get different types; or to be able to type all programs without type annotations (for many type systems, at least).

Our first observation is that with bidirectional typing, we can naturally require annotations, and avoid inference of multiple types for the same expression if necessary. For instance, for the expression 1 we can *infer* the type **Pos** *only*. Nevertheless, with bidirectional typing, 1 can still be checked against many types, including both **Pos** and **Nat**. If we want 1 to be of type **Nat**, then we can, for instance, require a type annotation 1 : **Nat**, and then 1 has type **Nat** via the check mode.

Our second, and most important observation, is that with bidirectional typing we can prevent the counter-examples by restricting the rule for typing references. In their algorithmic typing formulation, Davies and Pfenning [2000] use the following checking rule for references:

$$\frac{\Sigma; \Gamma \vdash e \Leftarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Leftarrow \mathsf{Ref}\ A}$$

This rule allows `ref 1` to be checked against both `Ref Pos` and `Ref Nat`, since 1 can be checked with both **Pos** and **Nat**. As such, their algorithmic typing, if unrestricted, can still type-check the problematic programs in Figure 8. We propose instead to use the following rule:

$$\frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Rightarrow \mathsf{Ref}\ A}$$

With this rule, both counter-examples (or other similar ones) are not typeable in the bidirectional type systems proposed in this paper:

- In the program in Figure 8a, intersection introduction cannot be used to type `ref 1` with `Ref Pos & Ref Nat`, since it would require checking `ref 1` with `Ref Nat`. Because of our restricted reference rule, `ref 1` can only infer the type `Ref Pos`, and it is not possible to check `ref 1` against `Ref Nat`. Using subsumption does not allow `ref 1` to have type `Ref Nat`, since `Ref Pos` is not a subtype of `Ref Nat`, due to the invariant subtyping rule for references (rule Sub-ref). Thus, the program is not typeable.
- In the program in Figure 8b, because `ref 1` infers `Ref Pos`, then the lambda λx.`ref 1` in the second program cannot be checked by (`Unit` → `Ref Nat`) & (`Unit` → `Ref Pos`) or (`Unit` → (`Ref Nat` & `Ref Pos`)). Therefore, the second program is not typeable either, even in the presence of distributivity.

With this simple change on a bidirectional type system, we can have a type sound calculus with distributive intersection subtyping, references, and an unrestricted intersection introduction rule.

Types	$A, B, C ::= \mathsf{Pos} \mid \mathsf{Nat} \mid \mathsf{Unit} \mid \top \mid A \to B \mid A\&B \mid \mathsf{Ref}\ A$
Expressions	$e ::= x \mid i \mid \mathsf{unit} \mid \top \mid \lambda x : A.e \mid e : A \mid e_1\ e_2 \mid o \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2$
Pre-Values	$p ::= \mathsf{unit} \mid \top \mid o \mid i \mid \lambda x : A.e$
Values	$v ::= p : A \mid \lambda x : A.e$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Value Stores	$\mu ::= \cdot \mid \mu, o = v$
Typing Stores	$\Sigma ::= \cdot \mid \Sigma, o : A$
Frames	$F ::= v\ \square \mid \square\ e \mid \mathsf{ref}\ \square \mid !\square \mid v := \square \mid \square := e$

Fig. 9. Syntax.

A complication is that we can no longer use a TAS and unannotated expressions to prove type soundness directly. Our solution relies on bidirectional typing and the presence of type annotations. Thus, we need to prove type soundness directly using a type system based on bidirectional typing. Furthermore, the operational semantics needs to account for type annotations. Fortunately, there are several calculi in the literature where type annotations are relevant and influence the result of reduction [Fan et al. 2022; Garcia et al. 2016; Herman et al. 2010; Huang et al. 2021; Siek and Wadler 2009; Wadler and Findler 2009]. These works provide techniques that are helpful to conduct type soundness proofs in similar settings to the one that we have here. Thus, we can reuse some of those ideas, to create a calculus similar to Davies and Pfenning's and show type soundness. As we shall see in Section 3.5, for this calculus (but not the one in Section 4), type annotations are still *computationally irrelevant* and can be erased. Thus no overhead needs to be introduced at runtime.

3.4 Syntax and Type System

The calculus that we present next is similar to the one by Davies and Pfenning, but we directly employ bidirectional typing. Since our goal is to illustrate how bidirectional typing can help to solve the problematic interaction between distributive intersection types and references, we present a simple type system, without a focus on maximizing type inference. We also make one simplification in lambda abstractions, and allow only abstractions of the form $\lambda x : A.e$, with a single annotation for the variable. This form is limiting for calculi with intersection types, since it forbids certain forms of overloading such as $\lambda x.x : (\mathsf{Nat} \to \mathsf{Nat})\&(\mathsf{Unit} \to \mathsf{Unit})$. To keep the full expressive power of an intersection type system it is well-known that standard forms of type annotations cannot work, and we must employ other forms of annotations [Pierce 1993; Reynolds 1997]. We avoid the extra complexity of those solutions here. Nevertheless, our form of lambda abstractions is still sufficient to encode the abstraction needed by the program in Figure 8b, which illustrates the issues with distributivity.

Syntax. The syntax of our calculus is shown in Figure 9. Meta-variables A, B and C range over types. A type is either a positive number (Pos), a natural number (Nat), a unit type (Unit), a top type ($\top$) or a compound type. Compound types include arrow types ($A \to B$), intersection types ($A\&B$) and reference types ($\mathsf{Ref}\ A$).

Meta-variable e ranges over expressions. Most expressions are standard. For references, there are locations (o), reference expressions $\mathsf{ref}\ e$, dereferences $!e$ and reference assignments $e_1 := e_2$. Values v are pre-values with a type ($p : A$) and lambdas. Since there is no constructor for intersection types, the annotation for values is used for preservation. For instance, to preserve types, $1 : \mathsf{Nat}\&\mathsf{Pos}$ should keep the annotations $\mathsf{Nat}\&\mathsf{Pos}$, otherwise the preservation lemma would not hold. Our contexts Γ have bindings of term variables with their types ($x : A$). There are stores for values μ and types Σ. Value stores are bindings of reference locations with their values ($o = v$), while a type store contains bindings of reference locations with the type of values ($o : A$). We use context evaluation, and the frames are standard.

$$
\begin{array}{ll}
\text{Ordinary Types} & A^\circ, B^\circ ::= \mathsf{Pos} \mid \mathsf{Nat} \mid \mathsf{Unit} \mid \top \mid \mathsf{Ref}\ A \mid A \to B^\circ \\
\text{Typing Modes} & \Leftrightarrow ::= \Rightarrow \mid \Leftarrow
\end{array}
$$

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

$$
\frac{}{A <: A}\ \textsc{S-refl}
\qquad
\frac{}{\mathsf{Pos} <: \mathsf{Nat}}\ \textsc{S-posi}
\qquad
\frac{A_1 <: A_3^\circ}{A_1 \,\&\, A_2 <: A_3^\circ}\ \textsc{S-andl}
\qquad
\frac{B_1 <: A_1 \qquad A_2 <: B_2^\circ}{A_1 \to A_2 <: B_1 \to B_2^\circ}\ \textsc{S-arr}
$$

$$
\frac{}{A <: \top}\ \textsc{S-top}
\qquad
\frac{\begin{array}{c} B <: A \\ A <: B \end{array}}{\mathsf{Ref}\ A <: \mathsf{Ref}\ B}\ \textsc{S-ref}
\qquad
\frac{\begin{array}{c} A <: B_1 \qquad A <: B_2 \\ B_1 \lhd B \rhd B_2 \end{array}}{A <: B}\ \textsc{S-and}
\qquad
\frac{A_2 <: A_3^\circ}{A_1 \,\&\, A_2 <: A_3^\circ}\ \textsc{S-andr}
$$

$\boxed{B \lhd A \rhd C}$ *(Type Splitting)*

$$
\frac{}{A \lhd A \,\&\, B \rhd B}\ \textsc{Sp-and}
\qquad
\frac{C \lhd B \rhd D}{A \to C \lhd A \to B \rhd A \to D}\ \textsc{Sp-arrow}
$$

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

$$
\frac{}{\Sigma; \Gamma \vdash \top \Rightarrow \top}\ \textsc{ITyp-top}
\qquad
\frac{}{\Sigma; \Gamma \vdash \mathsf{unit} \Rightarrow \mathsf{Unit}}\ \textsc{ITyp-unit}
\qquad
\frac{}{\Sigma; \Gamma \vdash 0 \Rightarrow \mathsf{Nat}}\ \textsc{ITyp-lit}
$$

$$
\frac{i > 0}{\Sigma; \Gamma \vdash i \Rightarrow \mathsf{Pos}}\ \textsc{ITyp-pos}
\qquad
\frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x \Rightarrow A}\ \textsc{ITyp-var}
\qquad
\frac{\begin{array}{c} A_2 <: A_1 \\ \Sigma; \Gamma, x : A_1 \vdash e \Leftarrow B \end{array}}{\Sigma; \Gamma \vdash \lambda x : A_1 .\, e \Leftarrow A_2 \to B}\ \textsc{ITyp-abs}
$$

$$
\frac{\Sigma; \Gamma \vdash e_1 \Rightarrow A_1 \to A_2 \qquad \Sigma; \Gamma \vdash e_2 \Leftarrow A_1}{\Sigma; \Gamma \vdash e_1\, e_2 \Rightarrow A_2}\ \textsc{ITyp-app}
\qquad
\frac{\Sigma; \Gamma \vdash e \Leftarrow A}{\Sigma; \Gamma \vdash e : A \Rightarrow A}\ \textsc{ITyp-anno}
$$

$$
\frac{o : A \in \Sigma}{\Sigma; \Gamma \vdash o \Rightarrow \mathsf{Ref}\ A}\ \textsc{ITyp-loc}
\qquad
\frac{\Sigma; \Gamma \vdash e_1 \Rightarrow \mathsf{Ref}\ A \qquad \Sigma; \Gamma \vdash e_2 \Leftarrow A}{\Sigma; \Gamma \vdash e_1 := e_2 \Rightarrow \mathsf{Unit}}\ \textsc{ITyp-ass}
$$

$$
\frac{\Sigma; \Gamma, x : A \vdash e \Leftarrow \top}{\Sigma; \Gamma \vdash \lambda x : A .\, e \Leftarrow \top}\ \textsc{ITyp-abst}
\qquad
\frac{\Sigma; \Gamma \vdash e \Rightarrow \mathsf{Ref}\ A}{\Sigma; \Gamma \vdash !e \Rightarrow A}\ \textsc{ITyp-deref}
\qquad
\frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Rightarrow \mathsf{Ref}\ A}\ \textsc{ITyp-ref}
$$

$$
\frac{\Sigma; \Gamma \vdash e \Rightarrow A \qquad A <: B}{\Sigma; \Gamma \vdash e \Leftarrow B}\ \textsc{ITyp-sub}
\qquad
\frac{B_1 \lhd B \rhd B_2 \qquad \Sigma; \Gamma \vdash e \Leftarrow B_1 \qquad \Sigma; \Gamma \vdash e \Leftarrow B_2}{\Sigma; \Gamma \vdash e \Leftarrow B}\ \textsc{ITyp-intro}
$$

Fig. 10. Type system.

Subtyping. The subtyping rules are shown at the top of Figure 10. The subtyping relation is a variant of the standard BCD-style distributive subtyping relation [Barendregt et al. 1983], and is similar to the one employed by Davies and Pfenning. However, we choose a presentation for the rules based on the work by Huang et al. [2021], which employs *splittable types* and *ordinary types*. Ordinary types [Davies and Pfenning 2000; Huang et al. 2021], represented as A°, are types that are not intersections or function types returning intersections. The opposite to ordinary types are splittable types, which are types that are intersections or function types returning intersections. Splittable types can be decomposed, unlike ordinary types, and they are helpful to eliminate the need for an explicit transitivity rule. Splittable types, along with ordinary types, are also helpful for obtaining an algorithmic formulation, as well as developing the metatheory of subtyping.

Ordinary and splittable types remove some overlap between the rules for intersection types. When an intersection appears on the right ($A <: B\&C$), we know that *we never lose expressive power* by employing rule S-And so, we can always pick this rule instead of the other two intersection

$\boxed{\mu; e \longmapsto \mu'; e'}$ *(Small-Step Semantics)*

$$\frac{\mu; e \longmapsto \mu'; e'}{\mu; F[e] \longmapsto \mu'; F[e']}\ \textsc{IStep-eval} \qquad \frac{o \notin \mu}{\mu; \mathsf{ref}\ v \longmapsto \mu, o = v; o}\ \textsc{IStep-ref}$$

$$\frac{o = v \in \mu}{\mu;\ !(o : \mathsf{Ref}\ A) \longmapsto \mu; v : A}\ \textsc{IStep-deref} \qquad \frac{e \neq p \qquad \mu; e \longmapsto \mu'; e'}{\mu; e : A \longmapsto \mu'; e' : A}\ \textsc{IStep-anno}$$

$$\frac{o = p : C \in \mu}{\mu; o : \mathsf{Ref}\ A := v \longmapsto \mu[o \mapsto |v| : C]; \mathsf{unit}}\ \textsc{IStep-ass} \qquad \frac{}{\mu; p : A : B \longmapsto \mu; p : B}\ \textsc{IStep-annov}$$

$$\frac{}{\mu; p \longmapsto \mu; p : \mathsf{ty}(p)_\mu}\ \textsc{IStep-p} \qquad \frac{}{\mu; (\lambda x : A_1 . e : B_1 \to B_2)\ v \longmapsto \mu; e[x \mapsto |v| : A_1] : B_2}\ \textsc{IStep-beta}$$

Fig. 11. Reduction.

rules. Thus, the ordinary condition expresses that in the two other rules (rules S-andl and S-andr) the type on the right cannot be an intersection. So, the other rules will not apply in cases where intersections appear on the right. In a setting with distributive subtyping, we need to generalize the definition of ordinary types a bit more because $A \to B\&C$ is equivalent to $(A \to B)\&(A \to C)$. So function types such as those should be considered as non-ordinary (or splittable). Ordinary and splittable types also help in proving inversion lemmas about the subtyping relation, which are needed for the type soundness proof. Usually, in subtyping relations with less overlapping rules, it becomes easier to prove inversion lemmas.

Types are split in rule S-and by the type splitting relation $B \lhd A \rhd C$, which is shown in the middle of Figure 10. An intersection type $A\&B$ is split into A and B. An arrow type is split by the output type. The rule S-d rule is admissible from rule S-and [Huang et al. 2021]. Rules S-arr, S-andl, and S-andr are for ordinary types. The subtyping rule S-ref for references is standard: it compares two reference types and requires that both types are subtypes of each other. Note that Ref $(A\&B)$ is not equivalent to (Ref A)&(Ref B), and that reference types are unsplittable. In other words, there is no distributivity rule for references. The subtyping rules are reflexive and transitive.

Type system. The bidirectional type system is shown at the bottom of Figure 10. The typing judgement is represented as $\Sigma; \Gamma \vdash e \Leftrightarrow A$. There are two typing modes $\Leftrightarrow$: infer mode ($\Rightarrow$) and check mode ($\Leftarrow$), shown at the top of Figure 10. An expression e is inferred or checked by type A under the reference typing store Σ and term variable context Γ. Most rules are standard. As we discussed, for references, expressions must infer a type (rule ITyp-ref). Because of distributivity, the intersection introduction rule requires type splitting. Lambdas are checked by $\top$ or a function type $A_2 \to B$ if A_2 is the subtype of the lambda input type A_1 and the body is checked by B.

3.5 Dynamic Semantics

Figure 11 shows the reduction rules of our calculus. We have two variants of reduction. One variant has type annotations, which are needed to prove type soundness. The other variant has no annotations and is used to illustrate that annotations are computationally irrelevant in this calculus.

Reduction with annotations. Rule IStep-eval reduces the expressions under the frames. Rule IStep-p reduces inferrable raw values to be values by annotating their types ($\mathsf{ty}(p)_\mu$). The definition for inferrable raw values is:

$$\mathsf{ty}(\top)_\mu = \top \quad \mathsf{ty}(\mathsf{unit})_\mu = \mathsf{Unit} \quad \mathsf{ty}(o)_\mu = \mathsf{Ref}\ \mathsf{ty}(\mu(o))_\mu \quad \mathsf{ty}(0)_\mu = \mathsf{Nat} \quad \mathsf{ty}(\mathrm{i})_\mu = \mathsf{Pos}\ (\mathrm{i} > 0)$$

Rule IStep-anno is applied for annotations and e cannot be a pre-value p. Annotations are not included in the evaluation context because there is a side condition stating that e is not a p. The side condition is needed to avoid rule IStep-anno and rule IStep-p creating a cycle. Rule IStep-annov replaces the annotation for a value. Rule IStep-beta substitutes argument values by replacing the annotated type with the input type of lambdas after erasing the outer annotation. The result of beta reduction is annotated by the output type of the function. Since values include annotated values and lambdas, $|v|$ erases the annotation of annotated values but returns lambdas themselves. So, after annotating the input type of functions, the argument is still a value. Similarly, rule IStep-ass replaces new values with the type of old values in the store before updating. Rule IStep-ref generates a fresh location and stores the reference values. Rule IStep-deref gets the values in the store. Importantly, our calculus is proved to be type sound by the usual progress (Theorem 3.1) and preservation (Theorem 3.2) theorems.

Theorem 3.1 (Progress). *If* $\Sigma; \cdot \vdash e \Leftrightarrow A$ *and* $\Sigma \vdash \mu$ *then either* e *is a value or* $\exists\, e'\ \mu', \mu; e \longmapsto \mu'; e'$.

Theorem 3.2 (Type preservation). *If* $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, *and* $\mu; e \longmapsto \mu'; e'$ *then* $\exists\, \Sigma' \supseteq \Sigma$, $\Sigma'; \cdot \vdash e' \Leftrightarrow A$ *and* $\Sigma' \vdash \mu'$.

Type annotations are computationally irrelevant. We show an alternative dynamic semantics, without annotations in the extended version of the paper. The rules are standard. Importantly, we show that dynamic semantics with annotations reduces to the same values as the dynamic semantics without annotations (Theorem 3.3). The erase functions $|e|$ and $|\Sigma|$ remove annotations from expressions and values of typing stores. To be distinguishable, we use the subscript $|\mathsf{i}|$ to represent the dynamic semantics without annotations.

Theorem 3.3 (Type annotations are computationally irrelevant). *If* $\mu; e \longmapsto^* \mu'; v$ *then* $|\mu|; |e| \longmapsto^*_{|\mathsf{i}|} |\mu'|; |v|$.

4 The F^+_{im} Calculus: Extending F^+_{i} with References

In this section, we introduce the F^+_{im} calculus: a variant and extension of the F^+_{i} calculus [Fan et al. 2022] with references. This calculus is the core language employed by our implementation of CP with references. The F^+_{im} calculus employs bidirectional typing and adopts the restriction proposed in Section 3. In addition, F^+_{im} includes several other features, including the *merge operator* [Dunfield 2014; Reynolds 1997], and *disjoint polymorphism* [Alpuim et al. 2017], which are inherited from F^+_{i}. Note that disjoint polymorphism is a form of parametric polymorphism where type variables can have disjointness constraints.

4.1 Syntax

The syntax of F^+_{im} is shown in Figure 12. Many of the syntactic forms are the same as the calculus in Section 3, but we have a few new constructs.

Types and expressions. Types are extended with a type variable (X), single field record types ($\{l : A\}$) and disjoint polymorphic quantification [Alpuim et al. 2017] ($\forall X * A.B$). With disjoint polymorphism, disjointness constraints like $\forall X * A$ express that the type that instantiates X must be disjoint to the type A. This form of constraint is similar to absence constraints seen in row polymorphic calculi [Cardelli and Mitchell 1989; Harper and Pierce 1991; Leijen 2005; Shields and Meijer 2001], except that disjoint polymorphism accounts for subtyping. Ordinary types also include records with ordinary fields and polymorphic types with ordinary type bodies. The type of numbers is simplified to be only an integer type Int. Expressions are extended with merge expressions ($e_1 ,, e_2$), record projections ($e.l$), and expressions for polymorphism. Merges can be used to encode

Types	$A, B, C ::= \mathsf{Int} \mid \mathsf{Unit} \mid \top \mid A \to B \mid A \& B \mid \mathsf{Ref}\ A \mid X \mid \{l : A\} \mid \forall X * A.B$
Ordinary Types	$A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \mathsf{Unit} \mid \top \mid X \mid \mathsf{Ref}\ A \mid A \to B^\circ \mid \{l : A^\circ\} \mid \forall X * A.B^\circ$
Expressions	$e ::= x \mid i \mid \mathsf{unit} \mid \top \mid \lambda x.e \mid e : A \mid e_1\ e_2 \mid o \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2$ $\mid e_1 \mathbin{,\!,} e_2 \mid \{l = e\} \mid e.l \mid \Lambda X.e \mid e\ A$
Functionals	$f ::= (\lambda x.e) : A \to B \mid f : A \to B$
Pre-Values	$p ::= o \mid \Lambda X.e$
Values	$v ::= p : A \mid f \mid \mathsf{unit} \mid \top \mid i \mid v_1 \mathbin{,\!,} v_2 \mid \{l = v\}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, X * A$
Value Stores	$\mu ::= \cdot \mid \mu, o = v$
Typing Stores	$\Sigma ::= \cdot \mid \Sigma, o : A$
Application Arguments	$\mathrm{arg} ::= v \mid A \mid l$
Frames	$F ::= v\ \square \mid \square\ e \mid \square : A \mid \mathsf{ref}\ \square \mid !\square \mid v := \square \mid \square := e$ $\mid v \mathbin{,\!,} \square \mid \square \mathbin{,\!,} e \mid \square.l \mid \{l = \square\} \mid \square\ A$

Fig. 12. The syntax of the F^{+}_{im} calculus.

multiple field records with single field records ($\{l = e\}$) [Reynolds 1997]. For polymorphism, we have type abstractions ($\Lambda X.e$) and type applications ($e\ A$).

Values, application arguments and contexts. Because we prove type preservation using a bidirectional type system, and F^{+}_{im} has to account for type annotations, our form of values requires some non-standard forms to ensure that enough type information is preserved on values, and throughout the reduction process. For instance, if $v : A$ reduces to v, simply forgetting about the annotation, this would be ok in terms of the runtime behavior (for the calculus in Section 3.5), but would not be ok for preservation. An example would be $(\lambda x.x) : \mathsf{Int} \to \mathsf{Int}$ reducing to $\lambda x.x$. After doing this reduction, we would not be able to type check the resulting term using the bidirectional typing rules. Thus, we would not be able to prove type preservation. Therefore, values need to have sufficient annotations and reductions like the above should not be allowed.

We employ a form of values similar to those adopted in approaches based on type-directed operational semantics [Fan et al. 2022; Huang et al. 2021]. A functional value (f) is a lambda expression with one or more arrow types. Pre-values p are locations and type abstractions. A value v can be an annotated pre-value ($p : A$), a functional value, a top value $\top$, an integer value i, a merge of values or a record value ($v_1 \mathbin{,\!,} v_2$). The value form ($p : A$) is just a simple way to ensure sufficient annotations, but we could inline and/or simplify the definition of values, at the cost of making some rules more complicated. Furthermore, consider a function with two annotations $(\lambda x.e) : \mathsf{Int} \to \mathsf{Int} : (\mathsf{Int}\&\mathsf{Bool}) \to \top$. If we want to preserve types, we must retain $(\mathsf{Int}\&\mathsf{Bool}) \to \top$. However, for the runtime behavior, we must make sure that the argument is cast to Int before reduction. Thus we must also retain the inner input type Int. A very similar issue occurs in gradually typed languages, for essentially similar reasons: we wish to have preservation but, at the same time, types can have an effect on reduction and must be preserved for that effect to be enforced. In the work of Wadler and Findler [2009], they solve this problem by accumulating annotations of lambdas. We adopt a similar solution too. The only difference is that we syntactically capture that functional values can only be a lambda followed by annotations using the syntactic category f. Overall, these value categories help to simplify the presentation.

Values, types and projection labels can also be the argument of parallel elimination (discussed in Section 4.3). Thus, we have a syntactic category for application arguments. Our contexts Γ are extended with bindings of type variables with their disjoint types ($X * A$). Frames are augmented with merges, records, projections and type applications as well as annotations.

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

$$\frac{}{X <: X}\ \textsc{S-var} \qquad \frac{A_2 <: A_1 \qquad B_1 <: B_2^{\circ}}{\forall X * A_1.\, B_1 <: \forall X * A_2.\, B_2^{\circ}}\ \textsc{S-forall} \qquad \frac{A <: B^{\circ}}{\{l : A\} <: \{l : B^{\circ}\}}\ \textsc{S-rcd}$$

$\boxed{B \lhd A \rhd C}$ *(Extended Type Splitting)*

$$\frac{C \lhd B \rhd D}{\forall X * A.\, C \lhd \forall X * A.\, B \rhd \forall X * A.\, D}\ \textsc{Sp-forall} \qquad \frac{B \lhd A \rhd C}{\{l : B\} \lhd \{l : A\} \rhd \{l : C\}}\ \textsc{Sp-rcd}$$

$\boxed{A *_{\mathrm{ax}} B}$ *(Disjointness Axioms for References)*

$$\frac{}{\mathsf{Ref}\ A *_{\mathrm{ax}} \mathsf{Int}}\ \textsc{Dax-refInt} \qquad \frac{}{\mathsf{Int} *_{\mathrm{ax}} \mathsf{Ref}\ A}\ \textsc{Dax-IntRef} \qquad \frac{}{A_1 \to A_2 *_{\mathrm{ax}} \mathsf{Ref}\ A}\ \textsc{Dax-ArrRef}$$

$$\frac{}{\forall X * A.\, B *_{\mathrm{ax}} \mathsf{Ref}\ A}\ \textsc{Dax-allref} \qquad \frac{}{\mathsf{Ref}\ A *_{\mathrm{ax}} \{l : B\}}\ \textsc{Dax-refrcd} \qquad \frac{}{\{l : B\} *_{\mathrm{ax}} \mathsf{Ref}\ A}\ \textsc{Dax-rcdref}$$

$$\frac{}{\mathsf{Ref}\ A *_{\mathrm{ax}} A_1 \to A_2}\ \textsc{Dax-refarr} \qquad \frac{}{\mathsf{Ref}\ A *_{\mathrm{ax}} \forall X * A.\, B}\ \textsc{Dax-refall} \qquad \frac{}{\mathsf{Ref}\ A *_{\mathrm{ax}} \mathsf{Unit}}\ \textsc{Dax-refu}$$

$$\frac{}{\mathsf{Unit} *_{\mathrm{ax}} \mathsf{Ref}\ A}\ \textsc{Dax-uref}$$

$\boxed{\Gamma \vdash A * B}$ *(Disjointness)*

$$\frac{A *_{\mathrm{ax}} B}{\Gamma \vdash A * B}\ \textsc{D-ax} \qquad \frac{\rceil A \lceil}{\Gamma \vdash A * B}\ \textsc{D-topL} \qquad \frac{\rceil B \lceil}{\Gamma \vdash A * B}\ \textsc{D-topR}$$

$$\frac{A_1 \lhd A \rhd A_2 \qquad \Gamma \vdash A_1 * B \qquad \Gamma \vdash A_2 * B}{\Gamma \vdash A * B}\ \textsc{D-andL} \qquad \frac{\Gamma \vdash A * B}{\Gamma \vdash \{l : A\} * \{l : B\}}\ \textsc{D-rcdeq}$$

$$\frac{B_1 \lhd B \rhd B_2 \qquad \Gamma \vdash A * B_1 \qquad \Gamma \vdash A * B_2}{\Gamma \vdash A * B}\ \textsc{D-andR} \qquad \frac{X * A \in \Gamma \qquad A <: B}{\Gamma \vdash X * B}\ \textsc{D-varL}$$

$$\frac{X * A \in \Gamma \qquad A <: B}{\Gamma \vdash B * X}\ \textsc{D-varR} \qquad \frac{\Gamma, X * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Gamma \vdash \forall X * A_1.\, B_1 * \forall X * A_2.\, B_2}\ \textsc{D-for} \qquad \frac{\Gamma \vdash A * B}{\Gamma \vdash \mathsf{Ref}\ A * \mathsf{Ref}\ B}\ \textsc{D-ref}$$

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

$$\frac{}{\rceil \top \lceil}\ \textsc{TL-top} \qquad \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}\ \textsc{TL-and} \qquad \frac{\rceil B \lceil}{\rceil A \to B \lceil}\ \textsc{TL-arr} \qquad \frac{\rceil B \lceil}{\rceil \forall X * A.\, B \lceil}\ \textsc{TL-forall}$$

$$\frac{\rceil A \lceil}{\rceil \{l : A\} \lceil}\ \textsc{TL-rcd} \qquad \frac{\rceil A \lceil}{\rceil \mathsf{Ref}\ A \lceil}\ \textsc{TL-ref}$$

Fig. 13. Subtyping and disjointness (excerpt).

4.2 Subtyping and Disjointness

The extended subtyping rules are shown at the top of Figure 13. New rules S-var, S-forall, and S-rcd are for type variables, polymorphic types and record types. Importantly, extended subtyping is still proved to be reflexive and transitive (Lemma 4.1 and Lemma 4.2).

Lemma 4.1 (Reflexivity of subtyping). *$A <: A$.*

Lemma 4.2 (Transitivity of subtyping). *If $A <: B$ and $B <: C$ then $A <: C$.*

Disjointness is shown in Figure 13. Disjoint intersection types were proposed by Oliveira et al. [2016] to solve the ambiguity problem of merging expressions such as 1 and 2. In general, in calculi with a merge operator, disjointness of two types *ensures* that two types cannot have supertypes in common, except for top-like types. We extend the disjointness of F_i^+ with reference types. The rule D-AX shows that two different structural types are disjoint, and relies on an auxiliary relation $A *_{ax} B$. This relation captures simple disjointness axioms, such as an integer is disjoint to a function. The different structural types $A *_{ax} B$ for references are defined in the middle of Figure 13, while the full rules are shown in the extended version of the paper. The main disjointness relation has rules that deal with compound types, such as intersection types or records. Note that rules D-TOPL and D-TOPR have a special treatment for *top-like* types, which are shown at the bottom of Figure 13. Top-like types are disjoint to any type because they do not overlap with other types. Top-like types arise in calculi with merges and disjoint intersection types, and are covered in detail in previous works [Huang et al. 2021; Oliveira et al. 2016]. They are helpful to achieve determinism (see also Section 4.4). With the disjointness relation, we know, for example, that Int is not disjoint with Int&Bool, since Int is not disjoint with Int. Then Ref (Int&Bool) and Ref Int would not be disjoint either.

4.3 Bidirectional Typing

For typing, similarly to F_i^+ and the simplified system in Section 3, we use bidirectional typing. As explained by Huang et al. [2021] and Oliveira et al. [2016], a general subsumption rule can still cause ambiguity problems in the presence of the merge operator. For instance, 1 ⨟ True can have type Bool and 2 can have type Int, but if we merge these two expressions, ambiguity is introduced. Bidirectional typing avoids the ambiguity problem by using the weaker bidirectional subsumption rule. For 1 ⨟ True, we can infer the type Int&Bool. However, we can check the same term with type Bool and Int.

Moreover, types in programs should always be well-formed. The well-formedness relation is defined at the top of Figure 14. Type Int and $\top$ are well-formed. Arrow types, reference types and record types are well-formed if all their subcomponents are well-formed. The interesting case happens with intersection types $A\&B$: an intersection type is well-formed when types A and B are well-formed, and these types are disjoint (rule WEL-AND). If a type variable X is bound to a type in the context then it is well-formed (rule WEL-X). Our design is based on the original work on disjoint intersection types [Oliveira et al. 2016], which has a similar restriction. The work on F_i^+ [Fan et al. 2022], avoids this restriction but, as we shall discuss in Section 5, in the presence of references avoiding this restriction creates important technical difficulties.

Typing. Figure 14 shows the typing rules of F_{im}^+. Compared to the calculus in Section 3, there are annotations for locations o (rule TYP-LOC). This design choice is essentially similar to what happens in the work with gradual typing and references [Toro and Tanter 2020]. In their work, reference values have a similar annotated form for references. An annotated reference type Ref B should always be a supertype of the reference type in the store Ref A. If there is no annotation, values in the store would need to be cast every time to preserve types. This alternative design would be significantly more complicated, since value stores should be considered during casting. For example, let's assume we have a value of 1 in the store, located at o, and this o is annotated with type Ref (Int&$\top$) (*i.e.*, o : (Ref Int&$\top$)). To preserve the type we would need to cast values in the store to have type Int&$\top$. However, if we keep one type annotation for o, there is no need to cast the value in the store. Consequently, stores do not need to be updated every time. We delve into a more detailed discussion about the complexities and difficulties that arise if stores are carried everywhere at runtime in Section 4.4.

Applicative distribution. In rules TYP-APP, TYP-PRJ, and TYP-TAPP, there is an applicative distribution relation $A \rhd B$, which increases the flexibility of typing, and is shown at the bottom of

$\boxed{\Gamma \vdash A}$ *(Well-Formed Types)*

$$\frac{}{\Gamma \vdash \top}\;\textsc{wel-top} \qquad \frac{}{\Gamma \vdash \mathsf{Unit}}\;\textsc{wel-unit} \qquad \frac{}{\Gamma \vdash \mathsf{Int}}\;\textsc{wel-int} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\;\textsc{wel-arrow}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{Ref}\ A}\;\textsc{wel-ref} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash A * B}{\Gamma \vdash A \,\&\, B}\;\textsc{wel-and} \qquad \frac{X * A \in \Gamma}{\Gamma \vdash X}\;\textsc{wel-X}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, X * A \vdash B}{\Gamma \vdash \forall X * A.\, B}\;\textsc{wel-all} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \{l : A\}}\;\textsc{wel-rcd}$$

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

$$\frac{}{\Sigma; \Gamma \vdash \top \Rightarrow \top}\;\textsc{Typ-top} \qquad \frac{}{\Sigma; \Gamma \vdash \mathsf{unit} \Rightarrow \mathsf{Unit}}\;\textsc{Typ-unit} \qquad \frac{}{\Sigma; \Gamma \vdash i \Rightarrow \mathsf{Int}}\;\textsc{Typ-lit}$$

$$\frac{\Gamma \vdash A \qquad x : A \in \Gamma}{\Sigma; \Gamma \vdash x \Rightarrow A}\;\textsc{Typ-var} \qquad \frac{\Gamma \vdash A \qquad \Sigma; \Gamma, x : A \vdash e \Leftarrow B}{\Sigma; \Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}\;\textsc{Typ-abs}$$

$$\frac{\Sigma; \Gamma \vdash e \Rightarrow A \qquad A \rhd \{l : B\}}{\Sigma; \Gamma \vdash e.l \Rightarrow B}\;\textsc{Typ-prj} \qquad \frac{\begin{array}{c}\Sigma; \Gamma \vdash e_1 \Rightarrow A \\ \Sigma; \Gamma \vdash e_2 \Leftarrow A_1 \qquad A \rhd A_1 \to A_2\end{array}}{\Sigma; \Gamma \vdash e_1\, e_2 \Rightarrow A_2}\;\textsc{Typ-app}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e_1 \Rightarrow A \\ \Sigma; \Gamma \vdash e_2 \Rightarrow B \qquad \Gamma \vdash A * B\end{array}}{\Sigma; \Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B}\;\textsc{Typ-merge} \qquad \frac{\Sigma; \Gamma \vdash e \Leftarrow A}{\Sigma; \Gamma \vdash e : A \Rightarrow A}\;\textsc{Typ-anno}$$

$$\frac{\Gamma \vdash B \qquad o : A \in \Sigma \qquad \mathsf{Ref}\ A <: \mathsf{Ref}\ B}{\Sigma; \Gamma \vdash o : \mathsf{Ref}\ B \Rightarrow \mathsf{Ref}\ B}\;\textsc{Typ-loc} \qquad \frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \{l = e\} \Rightarrow \{l : A\}}\;\textsc{Typ-rcd}$$

$$\frac{\Sigma; \Gamma \vdash e_1 \Rightarrow \mathsf{Ref}\ A \qquad \Sigma; \Gamma \vdash e_2 \Leftarrow A}{\Sigma; \Gamma \vdash e_1 := e_2 \Rightarrow \mathsf{Unit}}\;\textsc{Typ-ass} \qquad \frac{\Sigma; \Gamma \vdash e \Rightarrow \mathsf{Ref}\ A}{\Sigma; \Gamma \vdash !e \Rightarrow A}\;\textsc{Typ-deref}$$

$$\frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Rightarrow \mathsf{Ref}\ A}\;\textsc{Typ-ref} \qquad \frac{\Gamma \vdash A \qquad \Sigma; \Gamma, X * A \vdash e \Leftarrow B}{\Sigma; \Gamma \vdash \Lambda X.\, e : \forall X * A.\, B \Rightarrow \forall X * A.\, B}\;\textsc{Typ-tabs}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \qquad B \rhd \forall X * B_1.\, B_2 \\ \Gamma \vdash A * B_1 \qquad \Sigma; \Gamma \vdash e \Rightarrow B\end{array}}{\Sigma; \Gamma \vdash e\ A \Rightarrow B_2[X \mapsto A]}\;\textsc{Typ-tapp} \qquad \frac{\Sigma; \Gamma \vdash e \Rightarrow A \qquad A <: B \qquad \Gamma \vdash B}{\Sigma; \Gamma \vdash e \Leftarrow B}\;\textsc{Typ-sub}$$

$\boxed{A \rhd B}$ *(Applicative Distribution)*

$$\frac{}{A \rhd A}\;\textsc{apd-refl} \qquad \frac{A \rhd A_1 \to A_2 \qquad B \rhd A_1 \to B_2}{A \,\&\, B \rhd A_1 \to A_2 \,\&\, B_2}\;\textsc{apd-andarr}$$

$$\frac{A \rhd \forall X * A_1.\, A_2 \qquad B \rhd \forall X * B_1.\, B_2}{A \,\&\, B \rhd \forall X * (A_1 \,\&\, B_1).\, (A_2 \,\&\, B_2)}\;\textsc{apd-andall} \qquad \frac{A \rhd \{l : A_1\} \qquad B \rhd \{l : B_1\}}{A \,\&\, B \rhd \{l : A_1 \,\&\, B_1\}}\;\textsc{apd-andrcd}$$

Fig. 14. Type system of F^{+}_{im}.

Figure 14. This relation reflects the subtyping (and distributivity rules) in function application and record projection. Normally, rule Typ-app should only allow A to be a function type. However, due to subtyping, we additionally allow intersections of function types to act as a function. For

$\boxed{A \lessapprox B}$ *(Isomorphic Subtyping)*

$$\frac{}{A \lessapprox A}\ \text{IS-REFL} \qquad \frac{A \lessapprox B}{\mathsf{Ref}\ A \lessapprox \mathsf{Ref}\ B}\ \text{IS-REF} \qquad \frac{A \lessapprox B}{\{l : A\} \lessapprox \{l : B\}}\ \text{IS-RCD}$$

$$\frac{A_1 \lessapprox B_1 \qquad A_2 \lessapprox B_2 \qquad B_1 \lhd B \rhd B_2}{A_1 \,\&\, A_2 \lessapprox B}\ \text{IS-AND}$$

Fig. 15. Isomorphic subtyping.

example, suppose that A is $(\mathsf{Int} \to \mathsf{Int})\&(\mathsf{Bool} \to \mathsf{Bool})$. Since A is a subtype of both $(\mathsf{Int} \to \mathsf{Int})$ and $(\mathsf{Bool} \to \mathsf{Bool})$ it can act as any of these 2 functions, and the application should be allowed provided that a compatible argument type is provided. Similar handling happens in rule Typ-tapp and rule Typ-prj. Note that, in rule apd-andarr the two distributed function types should have the same input type. Unlike output types, input types cannot be intersected. Otherwise, we would generate types that are not well-formed. For instance, $(\mathsf{Int} \to \mathsf{Int})\&(\mathsf{Int} \to \mathsf{Bool})$ can be matched as $(\mathsf{Int}\&\mathsf{Int}) \to (\mathsf{Int}\&\mathsf{Bool})$, but $\mathsf{Int}\&\mathsf{Int}$ is not well-formed. Other typing rules are similar to F_i^+.

The typing rules infer unique types (Lemma 4.4). Furthermore, types that can be used to infer or check expressions must be well-formed (Lemma 4.3). Lemma 4.5 shows that if an expression can be checked by one type, then it can also be checked by a supertype.

Lemma 4.3 (Well-formedness of typing). *If* $\Sigma; \Gamma \vdash e \Leftrightarrow A$ *then* $\Gamma \vdash A$.

Lemma 4.4 (Uniqueness of type inference). *If* $\Sigma; \Gamma \vdash e \Rightarrow A_1$ *and* $\Sigma; \Gamma \vdash e \Rightarrow A_2$ *then* $A_1 = A_2$.

Lemma 4.5 (Subsumption of type checking). *If* $\Sigma; \Gamma \vdash e \Leftarrow A$ *and* $A <: B$ *then* $\Sigma; \Gamma \vdash e \Leftarrow B$.

Store typing. In the dynamic semantics, types should be split first and then values are cast under the split types in parallel. Thus we are not using the syntactic equality of types. Instead, we use a restricted form of equivalent types called isomorphic subtyping [Fan et al. 2022], which is shown in Figure 15. Lemma 4.7 shows that if two types are in an isomorphic subtyping relation then they are equivalent.

With the help of typing and isomorphic subtyping, we define the relation between value and type stores. Definition 4.6 defines well-formed stores (μ) with respect to the typing locations Σ:

Definition 4.6 (Well-formedness of the store with respect to Σ). $\Sigma \vdash \mu \triangleq$ if $dom(\mu) = dom(\Sigma)$ then $\forall o \in \mu, \Sigma; \cdot \vdash \mu(o) \Rightarrow A$ and $A \lessapprox \Sigma(o)$.

A store is well-formed with the typing location if the store and the typing location contain the same domains. For each location, which is in the store, the bounded value $\mu(o)$ can be inferred with a type, which is an isomorphic subtype of the type bound in the typing location ($\Sigma(o)$).

Lemma 4.7 (Isomorphic subtyping). *If* $A \lessapprox B$ *then* $A <: B$ *and* $B <: A$.

4.4 Dynamic Semantics

We employ a type-directed operational semantics (TDOS) [Huang et al. 2021] for the dynamic semantics. Following the TDOS approach, our dynamic semantics consists of two parts: casting and reduction. Moreover, small-step reduction uses an auxiliary parallel elimination relation.

Our subtyping is essentially coercive, and casting implements coercions in the operational semantics provided a type as the casting target, which is why we use the term type-directed operational semantics. As a result, types cannot be erased at runtime because casting relies on types. The simplest way to trigger casting is with an explicit type annotation like $(1 \,,\, \mathsf{True}) : \mathsf{Int}$

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

$$\frac{\rceil A^\circ \lceil}{v \longmapsto_{A^\circ} \mathsf{TV}(A^\circ)}\ \textsc{cast-top} \qquad \frac{}{i \longmapsto_{\mathsf{Int}} i}\ \textsc{cast-i} \qquad \frac{}{\mathsf{unit} \longmapsto_{\mathsf{Unit}} \mathsf{unit}}\ \textsc{cast-unit}$$

$$\frac{\neg \rceil A \to B^\circ \lceil}{\mathsf{f} \longmapsto_{A \to B^\circ} \mathsf{f} : A \to B^\circ}\ \textsc{cast-f} \qquad \frac{B <: C^\circ \qquad \neg \rceil C^\circ \lceil}{p : B \longmapsto_{C^\circ} p : C^\circ}\ \textsc{cast-anno}$$

$$\frac{v \longmapsto_{A^\circ} v'}{\{l = v\} \longmapsto_{\{l : A^\circ\}} \{l = v'\}}\ \textsc{cast-rcd} \qquad \frac{v_1 \longmapsto_{A^\circ} v_1'}{v_1 \mathbin{,,} v_2 \longmapsto_{A^\circ} v_1'}\ \textsc{cast-mergel}$$

$$\frac{v_2 \longmapsto_{A^\circ} v_2'}{v_1 \mathbin{,,} v_2 \longmapsto_{A^\circ} v_2'}\ \textsc{cast-merger} \qquad \frac{B \lhd A \rhd C \qquad v \longmapsto_B v_1 \qquad v \longmapsto_C v_2}{v \longmapsto_A v_1 \mathbin{,,} v_2}\ \textsc{cast-and}$$

Fig. 16. Casting of F^+_{im}.

(which will trigger casting and return the value 1). Another example is function application:

$$((\lambda x.x + 1) : \mathsf{Int} \to \mathsf{Int})\ (1 \mathbin{,,} \mathsf{True})$$

The argument will be cast (or coerced) from $(1 \mathbin{,,} \mathsf{True})$ to 1 before addition (+1). This kind of casting (or coercion) is vital because of a semantic ambiguity issue. This issue arises when the merge operator interacts with subtyping, and it has been identified by Cardelli and Mitchell [1989]. Let us see an example in CP:

```
let f (r: {x : Bool}) = r, {y = 1} in (f {x = true; y = 2}).y
```

The function `f` expects the parameter `r` to have field `x`, but by width subtyping the argument can actually contain extra fields like `{y = 2}`. After merging `r` with `{y = 1}` in the function body, field `y` has two possible values, and the final result can be either `1` or `2`. Our solution is to cast the argument to exactly type `{x: Bool}`. That is, casting removes field `{y = 2}`. Therefore, we have to keep the function type at runtime to cast the argument. In this way, we avoid the semantic ambiguity, and the final result is always `1`. Thus, type annotations are computationally relevant in F^+_{im} and play an important role since type casting is essential for avoiding ambiguity problems.

Casting. Figure 16 shows the casting relation, which gives an interpretation to subtyping at runtime. The $v \longmapsto_A v'$ notation means that we cast value v under type A and return v'. Like subtyping, casting also needs to prioritize some rules over others in certain cases. Thus, casting also employs the notion of ordinary and splittable types to aid with such prioritization. If the cast type A is splittable, then values are cast by the splitting types separately and returning a merge of the two casting results (rule <span style="font-variant:small-caps">cast-and</span>). This rule ensures that we first decompose the intersections, as in the subtyping rule. Then most of the other rules only apply when the type is ordinary (i.e. it cannot be split further). Note that this rule showcases why annotations are needed for locations in rule Typ-loc. If the value store μ is carried during casting, the store μ would be updated in parallel to be μ_1 and μ_2 ($\mu; v \longmapsto_B \mu_1; v_1$ and $\mu; v \longmapsto_C \mu_2; v_2$). Thus, it would be hard to decide what the resulting store is, which could lead to a type unsoundness problem. Thus, we annotate locations o with an annotation to avoid carrying μ everywhere and to simplify the casting rules.

When the cast type A is an ordinary and a top-like type, we generate the top-like values with an auxiliary $\mathsf{TV}(A)$ function, which is shown in the extended version of the paper. A naive rule for casting under the top type could be:

$$\frac{A <: \top}{v : A \longmapsto_\top v : \top}$$

$\boxed{\mu; v \bullet \mathsf{arg} \longmapsto \mu'; e}$ *(Parallel Elimination)*

$$\frac{v \longmapsto_A v'}{\mu\,;\, \lambda x.\, e : A \to B \;\bullet\; v \;\longmapsto\; \mu\,;\, e[x \mapsto v'] : B}\ \textsc{pap-beta}$$

$$\frac{}{\mu\,;\, \mathsf{f} : A \to B \;\bullet\; v \;\longmapsto\; \mu\,;\, \mathsf{f}\,(v : A) : B}\ \textsc{pap-app} \qquad \frac{o = v_1 \in \mu \qquad v_2 \longmapsto_{\mathsf{ty}(v_1)} v_2'}{\mu\,;\, o : \mathsf{Ref}\ A \;\bullet\; v_2 \;\longmapsto\; \mu[o \mapsto v_2']\,;\, \mathsf{unit}}\ \textsc{pap-ass}$$

$$\frac{}{\mu\,;\, \Lambda X.\, e : \forall X * A.\, B \;\bullet\; C \;\longmapsto\; \mu\,;\, e[X \mapsto C] : B[X \mapsto C]}\ \textsc{pap-tapp}$$

$$\frac{}{\mu\,;\, \{l = v\} \;\bullet\; l \;\longmapsto\; \mu\,;\, v}\ \textsc{pap-pj} \qquad \frac{\mu\,;\, v_1 \;\bullet\; \mathsf{arg} \;\longmapsto\; \mu\,;\, e_1 \qquad \mu\,;\, v_2 \;\bullet\; \mathsf{arg} \;\longmapsto\; \mu\,;\, e_2}{\mu\,;\, v_1 \,{}_9\, v_2 \;\bullet\; \mathsf{arg} \;\longmapsto\; \mu\,;\, e_1 \,{}_9\, e_2}\ \textsc{pap-merge}$$

$\boxed{\mu; e \longmapsto \mu'; e'}$ *(Small-Step Semantics)*

$$\frac{\mu; e \;\longmapsto\; \mu'; e'}{\mu; F[e] \;\longmapsto\; \mu'; F[e']}\ \textsc{Step-eval} \qquad \frac{v : A \neq \mathsf{f} \qquad v \longmapsto_A v'}{\mu; v : A \;\longmapsto\; \mu; v'}\ \textsc{Step-annov}$$

$$\frac{o \notin \mu}{\mu; \mathsf{ref}\ v \;\longmapsto\; \mu, o = v; o : \mathsf{Ref}\ \mathsf{ty}(v)}\ \textsc{Step-ref} \qquad \frac{\mu\,;\, v_1 \;\bullet\; v_2 \;\longmapsto\; \mu\,;\, e}{\mu; v_1\, v_2 \;\longmapsto\; \mu; e}\ \textsc{Step-beta}$$

$$\frac{o = v \in \mu}{\mu;\ !(o : \mathsf{Ref}\ A) \;\longmapsto\; \mu; v : A}\ \textsc{Step-deref} \qquad \frac{\mu\,;\, v_1 \;\bullet\; v_2 \;\longmapsto\; \mu'\,;\, e}{\mu; v_1 := v_2 \;\longmapsto\; \mu'; e}\ \textsc{Step-ass}$$

$$\frac{\mu\,;\, v \;\bullet\; A \;\longmapsto\; \mu\,;\, e}{\mu; v\, A \;\longmapsto\; \mu; e}\ \textsc{Step-tapp} \qquad \frac{\mu\,;\, v \;\bullet\; l \;\longmapsto\; \mu\,;\, e}{\mu; v.l \;\longmapsto\; \mu; e}\ \textsc{Step-pj}$$

Fig. 17. Small-step semantics of F^+_{im}.

but this causes a problem, because for the above merge, we could either have $1 : \top \,{}_9\, 2 : \top \longmapsto_\top 1 : \top$ or $1 : \top \,{}_9\, 2 : \top \longmapsto_\top 2 : \top$. So casting would not be deterministic. In a coercive semantics like ours, the correct way to deal with $\top$ is to view $\top$ as the unit type that has a single value. Then casting under the top type returns that value and then this recovers determinism. However, there are other top-like types that cause similar issues and have to be dealt with accordingly, which is why this notion appears in the literature of disjoint intersection types [Huang et al. 2021; Oliveira et al. 2016]. If the cast type A is not an ordinary and top-like type, they are covered by case analysis on the value. An integer i cast under type Int results in itself (rule CAST-I). Rule CAST-F shows that a functional value, cast under an ordinary function type, simply adds the annotation to the value. Annotated values replace the annotation by the cast type (rule CAST-ANNO). Rule CAST-RCD states that records cast under ordinary record types update the field value by casting under the field type of records. When merged values are cast under ordinary types, the result of casting comes from either the left or the right branch (rules CAST-MERGEL and CAST-MERGER).

Parallel elimination. The parallel elimination relation deals with elimination forms requiring an extra *argument* during the elimination process. It gives an interpretation to applicative distribution at runtime. An argument can be a value, a type or a label. There are four expressions that benefit from parallel elimination: applications, reference assignments, record projections and type applications. The definition of parallel elimination is shown in Figure 17. The $\mu; v \bullet \mathsf{arg} \longmapsto \mu'; e$ notation means that value v takes an argument with value store μ and returns expression e with store μ'.

Rules PAP-BETA and PAP-APP are used for expression applications. The annotations of functional values are erased one by one by rule PAP-APP. Before doing beta reduction, the argument values v are cast under the input type of functions, and the lambda body is annotated with the output type of function rule PAP-BETA. Rule PAP-ASS casts the value argument v_2 to have the type of value v_1 in the store and then updates values in the store. The type of values is obtained from the function $\mathsf{ty}(v)$:

$$\mathsf{ty}(i) = \mathsf{Int} \quad \mathsf{ty}(\{l = v\}) = \{l : \mathsf{ty}(v)\} \quad \mathsf{ty}(p : A) = A \quad \mathsf{ty}(v_1 \,,\, v_2) = \mathsf{ty}(v_1) \,\&\, \mathsf{ty}(v_2)$$
$$\mathsf{ty}(\top) = \top \quad \mathsf{ty}(\mathsf{unit}) = \mathsf{Unit} \quad \mathsf{ty}(f : A \to B) = A \to B \quad \mathsf{ty}((\lambda x.e) : A \to B) = A \to B$$

Type applications are performed in rule PAP-TAPP. Rule PAP-PJ projects the field value of a record by its label. For merged values $(v_1 \,,\, v_2)$, both values reduce in parallel (rule PAP-MERGE).

An example showing how parallel elimination works is:

$$\cdot; ((\lambda x.(x \,,\, \mathsf{False}) : \mathsf{Int} \to \mathsf{Int\&Bool} : \mathsf{Int\&Bool} \to \mathsf{Bool}) \,,\, (\lambda x.x : \mathsf{Int\&Bool} \to \mathsf{Int})) \bullet (1 \,,\, \mathsf{True})$$
$$\longmapsto^* \{\text{by rules PAP-MERGE, PAP-APP, and PAP-BETA}\}$$
$$\cdot; ((1 \,,\, \mathsf{False}) : \mathsf{Int\&Bool} : \mathsf{Bool}) \,,\, ((1 \,,\, \mathsf{True}) : \mathsf{Int})$$

In this example, we consider an empty value store. Two merged functional values $(\lambda x.(x \,,\, \mathsf{False}) : \mathsf{Int} \to \mathsf{Int\&Bool} : \mathsf{Int\&Bool} \to \mathsf{Bool})$ and $(\lambda x.x : \mathsf{Int\&Bool} \to \mathsf{Int})$ are reduced in parallel under the argument $(1 \,,\, \mathsf{True})$ to be $((1 \,,\, \mathsf{False}) : \mathsf{Int\&Bool} : \mathsf{Bool})$ and $((1 \,,\, \mathsf{True}) : \mathsf{Int})$, respectively.

Reduction. Reduction rules are shown at the bottom of Figure 17. Reduction has the form $\mu; e \longmapsto \mu'; e'$. An expression e with the value store μ reduces to e' and store μ'. Rule STEP-EVAL evaluates the expressions with different contexts. Casting is triggered by annotations (rule STEP-ANNOV). Rules STEP-BETA, STEP-ASS, STEP-TAPP, and STEP-PJ trigger the parallel application to get a resulting expression. Reference values allocate a fresh location and save the value in the store (rule STEP-REF). Rule STEP-DEREF gets the corresponding value in the store by the location address.

Importantly, the reduction of F^{+}_{im} is deterministic. For well-typed expressions with well-formed stores, they reduce to the same expressions and same stores:

THEOREM 4.8 (DETERMINISM OF REDUCTION). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, $\mu; e \longmapsto \mu_1; e_1$, and $\mu; e \longmapsto \mu_2; e_2$ then $e_1 = e_2$ and $\mu_1 = \mu_2$.*

Type soundness. Moreover, F^{+}_{im} is type sound. Theorem 4.9 says that every well-type expression with well-formed stores is either or a value or can be reduced. While Corollary 4.11 shows that the reduction of well-typed expressions e with well-formed stores μ can be checked by the starting type A and the resulting stores μ' are well-formed with the extended typing store Σ'. The corollary of preservation is concluded from Theorem 4.10.

THEOREM 4.9 (PROGRESS OF REDUCTION). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$ and $\Sigma \vdash \mu$ then either e is a value or $\exists\, e'$ μ', $\mu; e \longmapsto \mu'; e'$.*

THEOREM 4.10 (TYPE PRESERVATION WITH RESPECT TO ISOMORPHIC SUBTYPING). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, and $\mu; e \longmapsto \mu'; e'$ then $\exists\, B\ \Sigma'$, $\Sigma' \supseteq \Sigma$, $\Sigma'; \cdot \vdash e' \Leftrightarrow B$, $B \lessapprox A$, and $\Sigma' \vdash \mu'$.*

COROLLARY 4.11 (TYPE PRESERVATION). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, and $\mu; e \longmapsto \mu'; e'$ then $\exists\, \Sigma'$, $\Sigma' \supseteq \Sigma$, $\Sigma'; \cdot \vdash e' \Leftarrow A$, and $\Sigma' \vdash \mu'$.*

5 Related Work

Intersection types with references. As discussed in Section 3, Davies and Pfenning identified that a naive use of intersection types is type unsound in the presence of references, and proposed a set of restrictions. There are some other solutions in the literature. Dezani-Ciancaglini and Della Rocca

[2007] solved the problem by only intersecting non-reference types. With this restriction, subtyping is allowed to have distributivity and an unrestricted intersection introduction rule. Both counter-examples are rejected, since **ref** 1 cannot be typed with **Ref Nat** & **Ref Pos**. Nevertheless, this approach forbids reference types to be intersected even with non-reference types. For instance, (**Ref Nat**) & **Pos** is not allowed. Dezani-Ciancaglini et al. [2009], refined the previous proposal to allow intersections between reference and non-reference types. They proposed a *kind* for types to track whether the type is a reference type or contains reference types in case of intersection. Then the type soundness is achieved by restricting intersection elimination rule to intersection types not containing references. While both works put restrictions on types, our solution does not rely on the sophisticated restriction of no reference intersections. Furthermore, this restriction seems to prevent some practical programs, such as records/objects with multiple reference fields. Blaauwbroek [2017] used a pass-by-sharing [Liskov et al. 1981] approach to model computational effects. In their approach, no reference type exists, and referencing and dereferencing are implicit, while variables are mutable. For assignments, an invariant typing rule is required.

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash e : A}{\Gamma \vdash x := e : A}$$

In this rule, the left expression can only be a variable and the e should be of the same type as the variable. Counter-examples are forbidden because the type of e is invariant. Our solution adopts a conventional form of references, and the restriction that we impose is lightweight, while allowing both distributivity and unrestricted intersection introduction. Furthermore, the restriction does not seem to be restrictive for practical programs.

Merge operator. Reynolds [1997] introduced a calculus with a restricted merge operator and intersection types. Dunfield [2014] proposed a calculus with an unrestricted merge operator, but the semantics is non-deterministic. To solve the source of non-determinism, Oliveira et al. [2016] introduced the notion of disjoint intersection types. Two types can be intersected if they are disjoint. The semantics of calculi with the merge operator is type-directed. Huang et al. [2021] proposed a new form of semantics called *Type-Directed Operational Semantics* (TDOS), which we also employ in F^+_{im}. Over the years several extensions have been proposed for calculi with disjoint intersection types. Alpuim et al. [2017] proposed *disjoint polymorphism* in the calculus F_{i}. Bi et al. [2019] improved F_{i} with distributive subtyping and unrestricted intersections. Later, Fan et al. [2022] improved the work of Bi et al. by employing a TDOS to obtain a F^+_{i} calculus with simpler proofs. Compared to F^+_{i}, F^+_{im} is extended with references. So imperative programming is supported. However, F^+_{im} does not support unrestricted intersections and two intersected types should be disjoint. To allow programs such as $1 : \mathsf{Int}\&\mathsf{Int}$, a notion called *consistency* is required. Consistency states that two values are consistent when casting them with the same type returns the same value. To preserve types, consistency should be preserved at runtime. To achieve this, merge expressions should be reduced in parallel when distributivity is present. But this introduces difficulties with references, since two value stores would be evaluated in parallel. Thus we do not employ consistency in F^+_{im}, and resort to restricting all intersections to be disjoint as in Oliveira et al.'s work.

Language designs for modularity. Family polymorphism [Ernst 2001] is a well-known idea in OOP. A family of related virtual classes is refined simultaneously via inheritance in family polymorphism. Family polymorphism provides an elegant solution to the expression problem [Ernst 2004]. There have been several languages and calculi that incorporate the idea of family polymorphism and virtual classes [Aracic et al. 2006; Clarke et al. 2007; Ernst 1999; Ernst et al. 2006; Jolly et al. 2004; Madsen and Møller-Pedersen 1989; Nystrom et al. 2004, 2006; Zhang and Myers 2017]. Among these approaches, J& [Nystrom et al. 2006] and Familia [Zhang and Myers 2017] share common features with CP. J& uses intersections to compose related packages or classes. Familia, which is a Java-like language with family polymorphism, also integrates subtype polymorphism and parametric polymorphism successfully. However, both J& and Familia do not support the merge operator, which allows for a more dynamic form of composition. All of the aforementioned approaches aim at modelling a traditional OOP language (usually Java-like) with family polymorphism.

CP is inspired by some of the ideas of family polymorphism. In particular, CP adopts subtyping and nested composition to enable powerful forms of multiple trait-based inheritance, which can model inheritance of whole sets of traits (that are similar to classes). However, CP is both technically very different from traditional calculi with family polymorphism, and it also employs a different programming style, based on a form of open pattern matching. As described in Section 1 compositional programming fits within a more general and recent theme of work [Fan and Parreaux 2023; Jin et al. 2023; van der Rest and Poulsen 2022, 2023], which aims at addressing modularity problems of functional programming. All these approaches promote a programming style based on open forms of pattern matching, although the technical details differ significantly. Previous work has mostly focused on purely functional settings. Closer to us, van der Rest and Poulsen [2022, 2023] also explore algebraic effects and handlers [Plotkin and Pretnar 2009], which are a popular approach to introduce effects into purely functional languages. However, they do not fully consider algorithmic aspects, or provide an implementation. In our work, we aim at supporting a more classical ML-style impure (functional) programming style, which does not track effects. As we illustrate in our work, this allows us to model mutable objects, as well as programs that create and traverse cyclic structures, which are difficult in a pure setting.

The motivation and inspiration for new language designs supporting open forms of pattern matching comes partly from previous work on design patterns to support extensibility and modularity. Such design patterns include, among others: *tagless-final embeddings* [Carette et al. 2009] and *data types à la carte* [Swierstra 2008] in functional programming; and *polymorphic embeddings* [Hofer et al. 2008] and *object algebras* [Oliveira and Cook 2012a; Rendel et al. 2014] in OOP languages. While these design patterns enable extensible and modular designs, they typically require boilerplate code, the use of sophisticated type-level features, and an unconventional programming style. By adopting a dedicated programming language design these issues can be overcome. CP is inspired by the work on object algebras and extensible Church encodings [Oliveira 2009; Oliveira et al. 2006]. However, CP has significantly improved linguistic support over object algebras. As discussed extensively by Zhang et al. [2021], nested composition and modular dependencies require sophisticated and cumbersome encodings in languages like Scala. In CP, these concepts are naturally supported.

6 Conclusion

In this paper, we propose two calculi. One solves the type soundness problem identified by Davies and Pfenning [2000] with lighter restrictions, based on bidirectional typing. The other calculus, named F_{im}^{+}, is a variant of F_{i}^{+} with references. We augment the CP language with references based on F_{im}^{+} as the core calculus. With references, CP enables a modular imperative programming style. We illustrate how graph structures can be encoded by CP with a live-variable analysis example. A direction for future work is to study whether disjoint union types [Rehman et al. 2022] are compatible with references, and add support for them in CP. Moreover, it would be interesting to explore whether effect handlers and algebraic effects can be encoded in CP, perhaps with some extensions.

Acknowledgments

We are grateful to anonymous reviewers and our colleagues at the HKU PL group. This work has been sponsored by Hong Kong Research Grants Council project number 17209821.

Data-Availability Statement

The artifact that supports the paper is available on Zenodo [Ye et al. 2024].

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques and Tools*. Pearson Education.

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_1

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development* 1 (2006). https://doi.org/10.1007/11687061_5

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic* 48 (1983). https://doi.org/10.2307/2273659

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.9

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.22

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *ESOP*. https://doi.org/10.1007/978-3-030-17184-1_14

Lasse Blaauwbroek. 2017. *On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects*. Master's thesis. Technical University Eindhoven.

Luca Cardelli and John C. Mitchell. 1989. Operations on records. *Mathematical Structures in Computer Science* 1 (1989). https://doi.org/10.1017/S0960129500000049

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (2009). https://doi.org/10.1017/S0956796809007205

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *AOSD*. https://doi.org/10.1145/1218563.1218578

William Cook and Jens Palsberg. 1989. A Denotational Semantics of Inheritance and Its Correctness. In *OOPSLA*. https://doi.org/10.1145/74878.74922

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 27 (1981). https://doi.org/10.1002/malq.19810270205

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *ICFP*. https://doi.org/10.1145/351240.351259

Mariangiola Dezani-Ciancaglini and Simona Ronchi Della Rocca. 2007. Intersection and Reference Types. In *Reflections on Type Theory, Lambda Calculus, and the Mind*. Radboud University Nijmegen.

Mariangiola Dezani-Ciancaglini, Paola Giannini, and Simona Ronchi Della Rocca. 2009. Intersection, Universally Quantified, and Reference Types. In *CSL*. https://doi.org/10.1007/978-3-642-04027-6_17

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-Grained Reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (2006). https://doi.org/10.1145/1119479.1119483

Jana Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming* 24, 2–3, 133–165.

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2021). https://doi.org/10.1145/3450952

Erik Ernst. 1999. *gbeta - a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph. D. Dissertation. University of Aarhus.

Erik Ernst. 2001. Family Polymorphism. In *ECOOP*. https://doi.org/10.1007/3-540-45337-7_17

Erik Ernst. 2004. The Expression Problem, Scandinavian Style. In *MASPEGHI@ECOOP*. https://doi.org/10.1007/978-3-540-30554-5_11

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *POPL*. https://doi.org/10.1145/1111037.1111062

Martin Erwig. 2001. Inductive graphs and functional graph algorithms. *Journal of Functional Programming* (2001). https://doi.org/10.1017/S0956796801004075

Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2022. Direct Foundations for Compositional Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2022.18

Andong Fan and Lionel Parreaux. 2023. `super`-Charging Object-Oriented Programming Through Precise Typing of Open Recursion. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.11

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *POPL*. https://doi.org/10.1145/2837614.2837670

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris VII.

Robert Harper and Benjamin Pierce. 1991. A record calculus based on symmetric concatenation. In *POPL*. https://doi.org/10.1145/99583.99603

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010). https://doi.org/10.1007/s10990-011-9066-z

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of DSLs. In *GPCE*. https://doi.org/10.1145/1449913.1449935

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021), e28. https://doi.org/10.1017/S0956796821000186

Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. In *PLDI*. https://doi.org/10.1145/3591286

Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. 2004. Simple Dependent Types: Concord. In *FTfJP@ECOOP*.

Daan Leijen. 2005. Extensible Records with Scoped Labels. In *TFP*.

Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. 1981. *CLU reference manual.* Springer. https://doi.org/10.1007/BFb0035014

Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*. https://doi.org/10.1145/74877.74919

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable extensibility via nested inheritance. In *OOPSLA*. https://doi.org/10.1145/1028976.1028986

Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: nested intersection for scalable software composition. In *OOPSLA*. https://doi.org/10.1145/1167473.1167476

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components: A Practical Solution to the Expression Families Problem. In *ECOOP*. https://doi.org/10.1007/978-3-642-03013-0_13

Bruno C. d. S. Oliveira and William R. Cook. 2012a. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *ECOOP*. https://doi.org/10.1007/978-3-642-31057-7_2

Bruno C. d. S. Oliveira and William R. Cook. 2012b. Functional programming with structured graphs. In *ICFP*. https://doi.org/10.1145/2398856.2364541

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006. Extensible and Modular Generics for the Masses. In *TFP*.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *ICFP*. https://doi.org/10.1145/2951913.2951945

Benjamin C. Pierce. 1993. Intersection Types and Bounded Polymorphism. In *TLCA*. https://doi.org/10.1007/BFb0037117

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *ESOP*. https://doi.org/10.1007/978-3-642-00590-9_7

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism.*

Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. 2022. Union Types with Disjoint Switches. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2022.25

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *OOPSLA*. https://doi.org/10.1145/2660193.2660237

John C. Reynolds. 1974. Towards a theory of type structure. In *Colloque sur la Programmation*. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1997. Design of the Programming Language Forsythe. In *Algol-like Languages*. Chapter 8. https://doi.org/10.1007/978-1-4612-4118-8_9

Mark Shields and Erik Meijer. 2001. Type-Indexed Rows. In *POPL*. https://doi.org/10.1145/360204.360230

Jeremy G. Siek and Philip Wadler. 2009. Threesomes, with and without Blame. In *STOP@ECOOP*. https://doi.org/10.1145/1570506.1570511

Wouter Swierstra. 2008. Data Types à la Carte (Functional Pearl). *J. Funct. Program.* 18, 4 (2008). https://doi.org/10.1017/S0956796808006758

Matías Toro and Éric Tanter. 2020. Abstracting Gradual References. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.33

Cas van der Rest and Casper Bach Poulsen. 2022. Towards a Language for Defining Reusable Programming Language Components (Project Paper). In *TFP*. https://doi.org/10.1007/978-3-031-21314-4_2

Cas van der Rest and Casper Bach Poulsen. 2023. Types and Semantics for Extensible Data Types. In *APLAS*. https://doi.org/10.1007/978-981-99-8311-7_3

Philip Wadler. 1998. The Expression Problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *ESOP*. https://doi.org/10.1007/978-3-642-00590-9_1

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation* 8, 4 (1995). https://doi.org/10.1007/BF01018828

Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2024. *Imperative Compositional Programming (Artifact)*. https://doi.org/10.5281/zenodo.13373228

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021). https://doi.org/10.1145/3460228

Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. In *OOPSLA*. https://doi.org/10.1145/3133894

Yaoda Zhou, Bruno C. d. S. Oliveira, and Andong Fan. 2022. A Calculus with Recursive Types, Record Concatenation and Subtyping. In *APLAS*. https://doi.org/10.1007/978-3-031-21037-2_9

Received 2024-04-04; accepted 2024-08-18