A Bowtie for a Beast (Technical Appendix)

Overloading, Eta Expansion, and Extensible Data Types in $F_{\bowtie}$

ANONYMOUS AUTHOR(S)

The typed *merge operator* offers the promise of a compositional style of statically-typed programming in which solutions to the expression problem arise naturally. This approach, dubbed *compositional programming*, has recently been demonstrated by Zhang et al. [51].

Unfortunately, the merge operator is an unwieldy beast. Merging values from overlapping types may be ambiguous, so *disjointness relations* have been introduced to rule out undesired nondeterminism and obtain a well-behaved semantics. Past type systems using a disjoint merge operator rely on intersection types, but extending such systems to include union types or overloaded functions is problematic: naively adding either reintroduces ambiguity. In a nutshell: the elimination forms of unions and overloaded functions require values to be distinguishable by case analysis, but the merge operator can create exotic values that violate that requirement.

This paper presents $F_{\bowtie}$, a core language that demonstrates how unions, intersections, and overloading can all coexist with a tame merge operator. The key is an underlying design principle that states that any two types can support either the deterministic merging of their values, or the ability to distinguish their values, but never both. To realize this invariant, we decompose previously studied notions of disjointness into two new, dual relations that permit the operation that best suits each pair of types. This duality respects the polarization of the type structure, yielding an expressive language that we prove to be both type safe and deterministic.

1 INTRODUCTION

Given two programs e_1 and e_2, the merge operator $e_1 \shortparallel e_2$ combines them into one program that supports all the operations of both of its inputs.[1] This powerful language feature has been used to model a wide range of mechanisms. The canonical example is for record concatenation, which has long been used to model multiple inheritance [16, 47]. As an example, a multi-field record might be represented as the merge of three singleton records:

$$e = \{l_1 \mapsto 1\} \shortparallel \{l_2 \mapsto \mathbf{true}\} \shortparallel \{l_3 \mapsto 3\}$$

In a typed setting, one assigns merges intersection types. Thus, e has the type $\{l_1 : \mathbf{Int}\} \sqcap \{l_2 : \mathbf{Bool}\} \sqcap \{l_2 : \mathbf{Int}\}$. Here we are assuming the record fields are all distinct: $l_1 \neq l_2 \neq l_3$.

Unfortunately, the merge operator has a cost: merges of arbitrary values lead to nondeterminism. For example, while merging records with distinct fields, as in the program e above, is perfectly reasonable, merging the boolean values **true** and **false** would require the existence of a third boolean value that is both **true** and **false**. This violates programmers' expectation that only one branch of an if statement will execute: a program like **if true** $\shortparallel$ **false then** e_1 **else** e_2 is ambiguous. We would like a way to enable the power of the merge operator, while disallowing nondeterministic programs like this one.

Oliveira et al. [36] introduce a typing discipline for the merge operator that rules out such nondeterministic programs using *disjoint intersection types*. Under such a typing discipline, only expressions with disjoint types can be merged. Since **Bool** is not disjoint from itself, the merge

[1]This operator is sometimes written $e_1 \,,,\, e_2$ in the literature, but we prefer the more symmetric "$\shortparallel$" since the operator is commutative and associative.

2021. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

true $\shortparallel$ **false** is rejected. This line of work led to success in the modeling a number of highly compositional programming patterns including forms of *family polymorphism* [24] and *first-class traits* [5]. It has culminated in languages that support a style of programming known as *compositional programming* [51] which offers solutions to challenges such as the Expression Problem [46]. At the heart of these designs is the ability of the merge operator to support *nested composition* [6], which exploits subtyping relationships like:

$$(\mathbf{Int} \to \{age : \mathbf{Int}\}) \sqcap (\mathbf{Int} \to \{name : \mathbf{String}\}) \quad \leq \quad \mathbf{Int} \to \{age : \mathbf{Int}, name : \mathbf{String}\}$$

Here, the merge operator lets us combine/compose the behavior of multiple implementations. If we think of the two functions in the intersection as constructors of objects, what you get out is a new constructor with the combined behaviour of both.

At the same time, the merge operator has also been shown to support a form of *function overloading*, in which a single function can have several bodies and the choice of which to run is determined by its arguments [13, 21, 38]. The merge operator can combine multiple implementations of functions that take different argument types. Later on, the appropriate implementation can be dispatched based on the argument at the application site. However, overloading can also easily lead to nondeterminism if not carefully managed.

Both of these features, *nested composition* and *overloading*, are independently useful, but, so far, no system accommodates them simultaneously. The typing discipline used for the merge operator in Oliveira's line of work does not support traditional forms of overloading, and, conversely, the systems that support overloading do not support nested composition. This is the first problem that we address in this paper.

Another natural feature to consider alongside overloading is *union types* [3]. $A \sqcup B$ is the *untagged* union of A and B. Both A values and B values have this type (because, e.g., $A \leq A \sqcup B$), but, unlike for tagged unions, there is no extra dynamic information attached to the value to reveal which case it is. Nevertheless, the constructors of values of type A will often be distinct from those of type B, which is sufficient to tell them apart. For our purposes, we assume that the constructors of base types like **Int** are disjoint from those of other base types, like **Bool**, so 1 and **true** are distinct. In other words, a programmer can use a pattern-matching construct to eliminate a value e of type $\mathbf{Int} \sqcup \mathbf{Bool}$ as follows:

$$\mathbf{match}\ e\ \mathbf{with}\ \{(x_1 : \mathbf{Int}) \mapsto \mathbf{true}, (x_2 : \mathbf{Bool}) \mapsto \mathbf{false}\}$$

If $e = 1 : \mathbf{Int}$, then it is distinct from the constructors of **Bool**, so the match above will work as intended and evaluate to **true**. Likewise, if $e = \mathbf{true} : \mathbf{Bool}$, the match will evaluate to **false**, so pattern matching discriminates between **Int** and **Bool** values as expected. Unfortunately, in the presence of the merge operator there is a problem: the term $e = 1 \shortparallel \mathbf{true}$ has type $\mathbf{Int} \sqcap \mathbf{Bool}$, which is a subtype of $\mathbf{Int} \sqcup \mathbf{Bool}$. Now to decide whether e is an **Int** or a **Bool** during pattern matching yields the answer that it is *both*, leading again to nondeterministic evaluation. Safely incorporating union types with intersections and the merge operator is the second problem we address.

Disjointness and polarity. This paper gives a solution to both of the problems mentioned above. The key idea is that for certain types, like **Int** and **Bool** above, pattern matching provides a mechanism for distinguishing their values. When there is such a mechanism for types A and B, we say A and B are *distinguishable*, written $A \mathrel{\diamond} B$. Our language design requires that patterns in different branches of a match construct match distinguishable types. Types that are eliminated via pattern matching are naturally distinguishable from each other; such types are called *positive* in the literature on polarized type theory [2]. Intuitively, the values such types are defined completely by how they are constructed, and pattern matching extracts all of the information about them.

On the flip side, functions are characterized via a "strong" (i.e. non-matching) elimination form. They are characterized extensionally by the contexts in which they are used, not by how they are constructed. This means that function types are *negative*. Negative types are relevant because, to our knowledge, all practical uses of the merge operator in the literature concern merges of negative types. To apply the merge operator to two arguments of types A and B, our design requires that A and B are *mergeable*, written $A \bowtie B$.

Together, these two relations describe when it is unambiguous to merge two functions to create an overloaded function. For example, it makes sense for two function types such as **Int** $\to$ **String** and **Bool** $\to$ **String** to be mergeable because their inputs are distinguishable. Moreover, as typically (albeit not universally) described in lambda calculi, a function does not provide a means of dynamically asking whether it accepts integers or strings as inputs. Thus, from this perspective, we would *not* expect the type **Int** $\to$ **String** to be distinguishable from the type **Bool** $\to$ **String**, and in our system they are not distinguishable.

We refer to both distinguishability and mergeability as *disjointness relations*: the former is the disjointness of types as sets of values and the latter is the disjointness of types as sets of their contexts (i.e., the operations that can be carried out on their values).

A key invariant of our language design is that no two inhabited types are both distinguishable and mergeable—they can be only one or the other. This discipline rules out the issues of nondeterminism that arise in the presence of both intersection and union types. For instance, this principle rules out the ambiguous match for $e = 1 \shortparallel \textbf{true}$: since **Int** and **Bool** are distinguishable, but they are not mergeable, and e is ill typed. (Indeed there is no term of the type **Int** $\sqcap$ **Bool** and we do not even consider that type to be well formed.) The observations that we have made about polarity suggest a natural answer to the question of whether a pair of types should be distinguishable (positive) or mergeable (negative). Our type system exploits this observation to create an expressive yet deterministic programming language.

Contributions. Our primary result is the formalization of a core language $F_{\bowtie}$ (pronounced "F bow"). We prove the type system sound and demonstrate its support for compositional programming. $F_{\bowtie}$ showcases several important aspects:

- It is the first language to include disjoint intersection and union types, overloading, and a deterministic merge operator. These features combine to permit use of nested composition.
- It is also a step towards incorporating the work of Castagna et al. into the literature on disjoint intersection types [36].
- As in past treatments of the merge operator, the operational semantics is type-directed [29]. However, type information is *not* used for selecting which overload to execute at a call site. In $F_{\bowtie}$, dispatch is (co-)pattern matching. Consequently, we characterize the dynamic role of types as *runtime η-expansion*.
- The type system demonstrates the dual concepts of *mergeability* and *distinguishability*. Both have been studied independently [36, 40], but here it is shown that, in tandem, they enable well-typed deterministic overloading.

The key technical results in this paper have been proven using a combination of pencil-and-paper proofs, for the results having to do with the term language constructs, and a Coq development. The Coq development formalizes certain type-level parts of the semantics, including subtyping, dispatch, and some key properties of disjointness. We include both components in the anonymous supplementary material.

2 OVERVIEW

This section gives an overview of this paper, starting with background on merges and disjoint intersection types, and then introducing the key ideas of our work.

2.1 Background

Intersection and union types. Intersection and union types [3, 17, 39] are widely used in diverse fields of programming languages. Intersection types were introduced to characterize exactly all strongly normalizing lambda terms. Union types were later introduced as the dual construct of intersection types [3]. Intersection types were first adopted for programming by work on Forsythe [41, 42] and subsequently employed to express key aspects of multiple inheritance [15] in object-oriented programming. The Scala language [34] and its DOT calculus [43], for example, make fundamental use of intersection types to express a class/trait that extends multiple other traits. Union types have also been adopted in programming languages. For instance they are widely used in TypeScript and Flow, and were also included in Scala 3.

The merge operator. The Forsythe language [41, 42] introduced a so-called *merge operator*, which allows building values that can have multiple types (expressed as intersection types). The merge operator has been studied more recently by Dunfield [21], who removed significant restrictions originally present in Reynolds' design. A simple example of a program using the merge operator is:

$$\textbf{let}\, f = \mathit{isDigit} \,\shortparallel\, \mathit{not}\,\textbf{in}\;(f\;\text{‘1’}, f\;\textbf{false})$$

Here f is an overloaded function that can take either a character or a boolean as an argument; it has the type $(\textbf{Char} \to \textbf{Bool}) \sqcap (\textbf{Bool} \to \textbf{Bool})$. The variable f is built using the merge operator and applying it extracts one of the functions from the merged value. In the body of the **let**, we see two applications of f: one to a character, and another to a boolean. This style of overloading is one of the major features of the merge operator that has been explored in earlier research [11, 21]. In addition to overloading, we can express multi-field records by merging single-field records (as already mentioned in the introduction).

Compositional programming and nested composition. Recent research on the merge operator shows that it also enables *first-class classes/traits* [5] and *compositional programming* [51]. Compositional programming supports extensible forms of datatypes and functions and offers a natural solution to hard modularity challenges, such as the Expression Problem [46]. At the heart of compositional programming is a mechanism, called *nested composition* [6], that composes behavior from multiple components in a merge. Nested composition differs from overloading or simple record projection, which select only one of the components in a merge. With nested composition we can write:

$$\textbf{let}\; \mathit{mkStudent} = (\lambda n : \textbf{Int}.\ \{\mathit{age} \mapsto n\}) \,\shortparallel\, (\lambda n : \textbf{Int}.\ \{\mathit{idNumber} \mapsto \ldots\})\;\textbf{in}\; \mathit{mkStudent}\; 25$$

In this case we combine two functions with a merge to get an expression with the type:

$$(\textbf{Int} \to \{\mathit{age} : \textbf{Int}\}) \sqcap (\textbf{Int} \to \{\mathit{idNumber} : \textbf{Int}\})$$

Using subtyping, *mkStudent* can be given the function type $\textbf{Int} \to \{\mathit{age} : \textbf{Int}, \mathit{idNumber} : \textbf{Int}\}$; operationally the semantics of merge combines the two functions. Thus, we can use *mkStudent* to build a new record that has both an *age* and a *idNumber* field.

Merges, Ambiguity, and Subtyping. The interaction between subtyping and the merge operator is subtle. To illustrate the issue, we use an example similar to one given by Cardelli and Mitchell [9]:

$$e_{\shortparallel} = \textbf{let}\; x : \{l_2 : \textbf{Bool}\} = \{l_1 \mapsto 1, l_2 \mapsto \textbf{true}\}\;\textbf{in}\;(\{l_1 \mapsto 2\} \,\shortparallel\, x).l_1 + 3$$

In this program, x has type $\{l_2 : \mathbf{Bool}\}$, despite also including a field l_1. The field l_1 is hidden due to subtyping, because $\{l_1 : \mathbf{Int}, l_2 : \mathbf{Bool}\} \leq \{l_2 : \mathbf{Bool}\}$. The merge $\{l_1 \mapsto 2\} \mathbin{,,} x$ appears to be safe, statically, because the type of x does not contain l_1. However, what should happen when we project l_1? If the original field l_1 is preserved in x then, when we later lookup l_1, there will be two l_1 fields. If we use a biased lookup, which returns either the first value from the left or the first value from the right in a merge, then the semantics of programs may lead to surprising behaviour. For instance, in the program above, if a right-biased lookup is used, then the program would return 4. However, a programmer may have expected 5 as a result, because the type of x appears to promise that no field l_1 is present. Moreover, if the types of the two l_1 fields are distinct, this program could lead to a runtime type-error (when the field of the wrong type is projected), unless special care in taken to prevent such a situation. In essence, we would like that information hidden via subtyping has no effect in later uses of values with hidden information. For this reason Cardelli and Mitchell argued that biased lookups should *not* be used. More detailed discussions about such issues can be found in work by Huang et al. [29].

Disjoint Intersection Types. To address the ambiguity problems, as well as the problems arising from the interactions between merges and subtyping, Oliveira et al. [36] proposed to restrict merges so that only mergeable types are accepted. Disjointness rejects ambiguous programs such as **true** $\mathbin{,,}$ **false**, because the types of the two values being merged are not disjoint. Moreover, disjointness ensures that the merge operator is symmetric (or unbiased), guaranteeing both the associativity and commutativity of the operator.

Originally, the semantics of the merge operator with disjoint intersection types was defined by elaboration, following the approach promoted by Dunfield [21]. More recently, Huang et al. [29] proposed a type-directed operational semantics. This approach gives a direct operational semantics to λ_i, which is a calculus with disjoint intersection types and the merge operator. In λ_i, programs can reduce without encountering ambiguities in the merges.

With a type-directed operational semantics, types are relevant at runtime, and they are used to enforce the information hiding promised by subtyping. Consider again $e_{,,}$, the program defined previously. In this case, we drop the field l_1 in x when the value is cast to the type $\{l_2 : \mathbf{Bool}\}$. Therefore, $(\{l_1 \mapsto 2\} \mathbin{,,} x).l_1$ would become $(\{l_1 \mapsto 2\} \mathbin{,,} \{l_2 \mapsto \mathbf{true}\}).l_1$ and the final result of the program would be 5. In other words, this solution to the problem of the interaction between merges and subtyping ensures that components of a merge that are hidden by subtyping are dropped from the value when upcasting.

While the existing approaches to disjointness can deal with programs that have merges of records or that use nested composition, they have restricted support for overloading. For instance, the merge used in the definition of the overloaded function f (i.e. *isDigit* $\mathbin{,,}$ *not*), would be rejected. In essence in the notion of disjointness proposed by Oliveira et al. [36], two functions are disjoint if their return types are disjoint. However $\mathbf{Char} \to \mathbf{Bool}$ and $\mathbf{Bool} \to \mathbf{Bool}$ have overlapping return types. The disjointness restriction contrasts with traditional approaches with overloading, where distinct input types are used to eliminate possible ambiguities for overloaded functions. In addition, none of the existing calculi with disjoint intersection types include union types, which introduce new ambiguity issues.

2.2 Challenges for Deterministic Merges with Overloading and Union Types

Union Types. In prior approaches, as exemplified by λ_i, the term $1 \mathbin{,,} \mathbf{true}$ is a well-typed merge. This is because the program contexts that can use an integer are distinct from boolean program contexts. For example, an integer context might be $[\cdot] + 3$, where $[\cdot]$ is a "hole" into which an integer value can be filled, but this context can never be confused with any boolean context, such as,

if $[\cdot]$ **then** e_1 **else** e_2. Therefore, no matter how a context uses the merged value, it is unambiguous whether the 1 or **true** must be projected. For instance, we have $(1 \,,, \, \mathbf{true}) + 3$ evaluates to $1 + 3$.

Now consider the union type $\mathbf{Int} \sqcup \mathbf{Bool}$. We assume that unions are *untagged*, meaning that, at runtime, there is no extra information added to indicate whether a value of this type has the type on the left side or the right side of the union. Nevertheless, as long as integer and boolean values have distinct runtime representations, i.e., they have separate constructors, it is possible to have a construct that can tell values of one type from the other as in this example from the introduction:

$$\mathbf{match}\ e\ \mathbf{with}\ \{(y : \mathbf{Bool}) \mapsto \mathbf{false}, (x : \mathbf{Int}) \mapsto \mathbf{true}\}$$

Given that, even without union types, pattern matching on boolean and integer values is a sensible operation, the behavior of the above match expression should be no surprise. It is, after all, equivalent to the (large) expression:

$$\mathbf{match}\ e\ \mathbf{with}\ \{\mathbf{true} \mapsto \mathbf{false}, \mathbf{false} \mapsto \mathbf{false}, 0 \mapsto \mathbf{true}, -1 \mapsto \mathbf{true}, 1 \mapsto \mathbf{true}, \ldots\}$$

However, now that we have introduced a single elimination form that works on the union type $\mathbf{Int} \sqcup \mathbf{Bool}$, merging becomes nondeterministic, as we saw in the case $e = 1 \,,, \, \mathbf{true}$. In other words, in λ_i, **Int** and **Bool** are not distinguishable because there is no context that can take a value that is either an **Int** or **Bool** and determine which type of value was provided. As a result, it is safe for these types to be mergeable in that setting. With unions and their elimination forms, this is no longer the case. Although permitted by λ_i, it is not clear from the literature, however, that merges like $1 \,,, \, \mathbf{true}$ have significant practical value. Thus in $\mathrm{F}_{\bowtie}$ we prefer to consider **Int** and **Bool** distinguishable in exchange for sacrificing the ability to merge them.

In contrast, the type $(\mathbf{Int} \to \mathbf{Bool}) \sqcup (\mathbf{Bool} \to \mathbf{Bool})$ is quite different. Unlike integer and boolean values, functions generally do not support pattern matching; they are eliminated via application. Thus, from a type-theoretic point of view, a program like

$$\mathbf{match}\ e\ \mathbf{with}\ \{(x : \mathbf{Int} \to \mathbf{Bool}) \mapsto \mathbf{true}, (y : \mathbf{Bool} \to \mathbf{Bool}) \mapsto \mathbf{false}\} \tag{1}$$

is unusual. It represents an operation on a function—*other than application*—that provides information about which inputs the function accepts. Unlike the previous example with $\mathbf{Int} \sqcup \mathbf{Bool}$, there is no similar way to express this kind of typecase analysis with more primitive patterns.

Implementing such a matching construct would require that functional values are tagged with type information at runtime. Thanks to subtyping, the type tag of a value could not be directly compared with the type annotation in a pattern. Therefore, run-time execution of the subtyping algorithm would be necessary. Such an arrangement certainly has precedent in the literature [13] and in OOP language implementations, but $\mathrm{F}_{\bowtie}$ aims to introduce union types *without* changing the meaning of existing types. In other words, support for union types should not depend on adding new operations on the values of other types. Our approach requires neither additional type tags at runtime nor any type-tag comparison in the dispatch procedure. As we shall see, $\mathrm{F}_{\bowtie}$ will exploit type annotations on the merge operator at runtime, but those are part of the merge construct—they are not part of the representation of values.

Overloading. Overloading faces the same issues with determinism as pattern matching on unions. When the merge operator overloads functions, we can rewrite the problematic example as:

$$((\lambda x : \mathbf{Int}.\ \mathbf{true}) \,,, \, (\lambda y : \mathbf{Bool}.\ \mathbf{false}))\ (1 \,,, \, \mathbf{true})$$

The merged function can be given the type $(\mathbf{Int} \to \mathbf{Bool}) \sqcap (\mathbf{Bool} \to \mathbf{Bool})$, which, due to subtyping, is equivalent to $(\mathbf{Int} \sqcup \mathbf{Bool}) \to \mathbf{Bool}$. Semantically, these merged functions act like the match expression that we already saw:

$$\mathbf{match}\ e\ \mathbf{with}\ \{(x : \mathbf{Int}) \mapsto \mathbf{true}, (y : \mathbf{Bool}) \mapsto \mathbf{false}\}$$

Indeed, for this reason, $F_{\bowtie}$ unifies the syntax for pattern match expressions with the syntax for (potentially merged) lambda abstractions—they are the same thing.

2.3 Information Hiding and Pattern Matching

Overloaded Functions and Copattern Matching. A common notation to represent overloaded functions in core calculi is with λ abstractions containing a case for each overload such as:

$$\lambda\{((x : \mathbf{Int}) \hookrightarrow \mathbf{Bool}) \mapsto \mathbf{true}, ((x : \mathbf{Bool}) \hookrightarrow \mathbf{Bool}) \mapsto \mathbf{false}\} \tag{2}$$

This is the normalized form of the merge we saw above. In each case, the type to the right of the $\hookrightarrow$ is the type the function returns when that case is matched.

This notation is similar to GHC's `LambdaCase` and OCaml's `function` syntax, both of which combine the introduction of a function with pattern matching on its argument. It may also be seen as a form of *copattern matching* [1, 50] and is common in the literature on overloading [13].

Typical pattern matching involves destructuring a value according to a pattern describing the shape of an introduction form. In example (2), on the other hand, the λ-value destructures its *evaluation context* according to *elimination patterns* (i.e., copatterns) that describe the shape of elimination forms. In that example, the context $[\cdot]\ 1$, matches the first elimination pattern $(x : \mathbf{Int}) \hookrightarrow \mathbf{Bool}$, whereas the context $[\cdot]\ \mathbf{true}$ matches the second. In this way, *dispatch*—the process of deciding which overload of a function to execute—is completely subsumed by (co)pattern matching.

η-Expansion. Recall the definition of $e_{\shortparallel}$.

$$e_{\shortparallel} = \mathbf{let}\ x : \{l_2 : \mathbf{Bool}\} = \{l_1 \mapsto 1, l_2 \mapsto \mathbf{true}\}\ \mathbf{in}\ (\{l_1 \mapsto 2\} \shortparallel x).l_1\ +\ 3$$

Previously, we saw that the expected semantics should hide the l_1 field at the assignment of x. Type annotations ought to have the effect at runtime of hiding information (such as a field of a record or an overload of a function) in a term. $F_{\bowtie}$ implements this by η-expanding annotated terms. An expression $(v : \{l_2 : \mathbf{Bool}\})$ η-expands to $\{l_2 \mapsto (v.l_2 : \mathbf{Bool})\}$, hiding any information, other than the contents of the field l_2, that may be present in v.

In this way, η-expansion of a value at a type builds a wrapper that limits access to the value to the operations supported by the type. This technique is quite similar to how sound gradual type systems [44] build wrappers to catch runtime type errors caused by untyped code passing ill-typed values to typed contexts.

Formally, in $F_{\bowtie}$, a value v inside a wrapper for type A is written $\lambda\{(* \hookrightarrow A) \mapsto v\}$, so we would write the example above as $\lambda\{(* \hookrightarrow \{l_2 : \mathbf{Bool}\}) \mapsto v\}$. Here, the elimination pattern $* \hookrightarrow A$ matches any context valid for type A. When this occurs, v is placed inside this context. In other words, the value $\lambda\{(* \hookrightarrow A) \mapsto v\}$ behaves exactly as v but only in contexts valid for type A. The $*$ pattern is a sort of dual to variable patterns.

Pattern Expansion is η-Expansion. We have already seen that type annotations on pattern variables have significance at runtime. Their meaning is derived from η principals for positive types. Consider an $F_{\bowtie}$ expression like $\mathbf{match}\ e\ \mathbf{with}\ \{(x : \mathbf{Bool}) \mapsto e'\}$. Dually to η expansion for a value of record type, which re-builds an output and projects each field individually, η expansion for a positive type like **Bool** re-builds an input by matching against all possible values of a type and specializing each branch. Given the term e' above has a free variable $x : \mathbf{Bool}$, the general form of η expansion for booleans would be:

$$e' \to_{\eta} \mathbf{match}\ x\ \mathbf{with}\ \{\mathbf{true} \mapsto e'[\mathbf{true}/x], \mathbf{false} \mapsto e'[\mathbf{false}/x]\}$$

From this, one might informally expect an equivalence:

$$\textbf{match}\ e\ \textbf{with}\ \{(x : \textbf{Bool}) \mapsto e'\} \equiv \textbf{match}\ e\ \textbf{with}\ \{\textbf{true} \mapsto e'[\textbf{true}/x], \textbf{false} \mapsto e'[\textbf{false}/x]\}$$

In $F_{\bowtie}$, this expansion is not just an equivalence: it is the definition of the operational semantics of pattern matching. This is possible because similar expansions exist for all positive types. During elimination pattern matching, a dual kind of expansion occurs for $*$-patterns. The wrapper $\lambda\{(* \hookrightarrow \{l_2 : \textbf{Bool}\}) \mapsto v\}$ is equivalent to $\{l_2 \mapsto (v.l_2 : \textbf{Bool})\}$. In other words, the pattern $* \hookrightarrow A^-$ η-expands according to the negative type A^-. In summary, in $F_{\bowtie}$ we can characterize the type-directed component of the operational semantics as run-time η-expansion.

2.4 Compositional Programming in $F_{\bowtie}$

To see how $F_{\bowtie}$ supports the kind of (nested) compositional programming offered by the merge operator, and to introduce the notation used by the calculus, we next consider how to build a small extensible interpreter.

In $F_{\bowtie}$, we can represent a language of integer literals and addition expressions by defining the type **IntArithExpr** A as a record type as:

$$\textbf{IntArithExpr}\ A = \{\mathit{constant} : \textbf{Int} \to A, \mathit{add} : A \to A \to A\}$$

This type represents a simplified form of a *compositional interface* [51], and is closely related to the kind of interfaces used in techniques such as *finally tagless* [10] or *object algebras* [35]. Indeed this interface is essentially the fold (F-)Algebra [8] for a simple datatype of arithmetic expressions. An $F_{\bowtie}$ term of type **IntArithExpr** A describes how to interpret an expression in our object language as a value of type A. We represent one such object-language expression as follows:

$$\begin{aligned}
&\mathit{three}\ :\ \forall\alpha.\textbf{IntArithExpr}\ \alpha \to \alpha\\
&\mathit{three} = \Lambda\alpha.\lambda(x : \textbf{IntArithExpr}\ \alpha) \hookrightarrow \alpha.\ x.\mathit{add}\ (x.\mathit{constant}\ 1)\ (x.\mathit{constant}\ 2)
\end{aligned}$$

The notation $\Lambda\alpha.e$ binds a type variable α in the body e while $\lambda(x : B) \hookrightarrow C.\ e$ binds the term variable x of type B in the body e, which is ascribed the type C. Values such as *three* are defined by implementing their folds.

A natural way of interpreting object-language expressions as integers is by evaluating them. The interpretation *evalInt* describes how to do this, by defining a meaning for each field:

$$\begin{aligned}
&\mathit{evalInt}\ :\ \textbf{IntArithExpr Int}\\
&\mathit{evalInt} = \{\mathit{constant} \mapsto \lambda x : \textbf{Int} \hookrightarrow \textbf{Int}.\ x, \mathit{add} \mapsto (+)\}
\end{aligned}$$

Now the program *three* [**Int**] *evalInt*, of type **Int**, evaluates to 3.

Suppose that we wish to extend this language so that a constant may be either an integer or a floating point number. For simplicity, we will not presume any subtyping relationship between **Int** and **Float**, but instead rely on an explicit cast *toFloat* : **Int** $\to$ **Float**. What we wish is to obtain a combined language where both floating point numbers and integers can be used. Moreover, the language should automaticaly convert between integers and floating point numbers when necessary. Orchard and Schrijvers [37] tackle a similar problem in the setting of a typed object language. They illustrate that the problem is tricky to solve in Haskell using a finally tagless embedding. To this end, Orchard and Schrijvers proposed to extend Haskell with *constraint synonyms*, which later helped motivate the addition of the `ConstraintKinds` GHC extension. In contrast, in $F_{\bowtie}$, merge-based overloading with intersection and union types provides what we need.

We define the type **IntFloatArithExpr** A as an extension of **IntArithExpr** A using intersection. Observe that the intersection distributes over the record, augmenting the type of the *constant* field.

$$\begin{aligned}\mathbf{IntFloatArithExpr}\ A &= \mathbf{IntArithExpr}\ A\ \sqcap\ \{\mathit{constant} : \mathbf{Float} \to A\}\\ &\equiv \{\mathit{constant} : (\mathbf{Int} \to A) \sqcap (\mathbf{Float} \to A), \mathit{add} : A \to A \to A\}\\ &\equiv \{\mathit{constant} : \mathbf{Int} \sqcup \mathbf{Float} \to A, \mathit{add} : A \to A \to A\}\end{aligned}$$

In other words, subtyping gives us the equivalence $(\mathbf{Int} \to A) \sqcap (\mathbf{Float} \to A) \equiv (\mathbf{Int} \sqcup \mathbf{Float}) \to A$. Such equivalences are key to compositional programming [7].

To evaluate expressions in this language, we need to define an interpretation $\mathit{evalIntFloat}$: **IntFloatArithExpr** $(\mathbf{Int} \sqcup \mathbf{Float})$. A first step is to define the type expressions as well as the evaluation function for the sub-language containing floating point but not integer literals. (We use $+.$ for the floating-point addition primitive.)

$$\mathbf{FloatArithExpr}\ A = \{\mathit{constant} : \mathbf{Float} \to A, \mathit{add} : A \to A \to A\}$$

$$\begin{aligned}&\mathit{evalFloat}\ :\ \mathbf{FloatArithExpr}\ \mathbf{Float}\\ &\mathit{evalFloat} = \{\mathit{constant} \mapsto \lambda x : \mathbf{Float} \hookrightarrow \mathbf{Float}.\ x, \mathit{add} \mapsto (+.)\}\end{aligned}$$

It is now possible to merge our two evaluators.

$$\begin{aligned}&\mathit{partialEvalIntFloat}\ :\ (\mathbf{IntArithExpr}\ \mathbf{Int}) \sqcap (\mathbf{FloatArithExpr}\ \mathbf{Float})\\ &\mathit{partialEvalIntFloat} = \mathit{evalInt} \shortparallel \mathit{evalFloat}\end{aligned}$$

But, we are not finished: we would like an expression of type **IntFloatArithExpr** $(\mathbf{Int} \sqcup \mathbf{Float})$, but only have one of type $(\mathbf{IntArithExpr}\ \mathbf{Int}) \sqcap (\mathbf{FloatArithExpr}\ \mathbf{Float})$. This is not enough because the former type requires the *add* operation to support addition of integers with floating point numbers, which the latter does not. As a result, trying to use *partialEvalIntFloat* to evaluate *four* (defined below) would be ill typed.

$$\begin{aligned}\mathit{four} = \Lambda\alpha.\lambda(x : \mathbf{IntFloatArithExpr}\ \alpha) \hookrightarrow \alpha.&\\ x.\mathit{add}\ (x.\mathit{constant}\ 2.0)\ (x.\mathit{constant}\ 2)&\end{aligned}$$

In order to complete the evaluator, we need an addition extension that handles the missing cases. Its type is given below.

$$\mathbf{ArithExt} = \{\mathit{add} : (\mathbf{Int} \to \mathbf{Float} \to \mathbf{Float}) \sqcap (\mathbf{Float} \to \mathbf{Int} \to \mathbf{Float})\}$$

We implement the extension by coercing integers to floats and using floating point addition, as shown in the code below. Note that the *add* field of the record contains a "function" with two bodies, distinguished by the input type of the argument x.

$$\begin{aligned}&\mathit{evalExt}\ :\ \mathbf{ArithExt}\\ &\mathit{evalExt} = \{\mathit{add} \mapsto \lambda\{\ (x : \mathbf{Int}) \mapsto \lambda(y : \mathbf{Float}) \hookrightarrow \mathbf{Float}.\ \mathit{toFloat}\ x\ +.\ y,\\ &\qquad\qquad\qquad\qquad (x : \mathbf{Float}) \mapsto \lambda(y : \mathbf{Int}) \hookrightarrow \mathbf{Float}.\ x\ +.\ \mathit{toFloat}\ y\ \}\ \}\end{aligned}$$

Applying distributivity, we can show that

$$(\mathbf{IntArithExpr}\ \mathbf{Int}) \sqcap (\mathbf{FloatArithExpr}\ \mathbf{Float}) \sqcap \mathbf{ArithExt} \leq \mathbf{IntFloatArithExpr}\ (\mathbf{Int} \sqcup \mathbf{Float})$$

This tells us that merging *partialEvalIntFloat* with *evalExt* yields an expression of the desired type.

$$\begin{aligned}&\mathit{evalIntFloat}\ :\ \mathbf{IntFloatArithExpr}\ (\mathbf{Int} \sqcup \mathbf{Float})\\ &\mathit{evalIntFloat} = \mathit{partialEvalIntFloat} \shortparallel \mathit{evalExt}\end{aligned}$$

Now, evaluating *four* $[\mathbf{Int} \sqcup \mathbf{Float}]$ *evalIntFloat* results in the floating-point value 4.0 as expected.

expressions	e	$::=$	$x \mid c\ e \mid e_1\ e_2 \mid e\,[A]$ $\mid (e : A)$ $\mid (e_1 : A_1) \shortparallel (e_2 : A_2)$ $\mid \lambda\{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\}$	*types*	A, B, C	$::=$	$\alpha \mid c\ A$ $\mid A \to B \mid \forall\,\alpha.B$ $\mid \bot \mid A_1 \sqcup A_2$ $\mid \top \mid A_1 \sqcap A_2$
				neg. types	A^-, B^-	$::=$	$A \to B \mid \forall\,\alpha.B$ $\mid A_1^- \sqcup A_2^-$ $\mid \top \mid A_1^- \sqcap A_2^-$
values	v	$::=$	$x \mid c\ v$ $\mid \lambda\{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\}$				
value patterns	p	$::=$	$x : A \mid c\ p \mid (p_1 \vert p_2)$ $\mid p_1 \& p_2$	*elim. types*	$\hat{A}, \hat{B}$	$::=$	$A \mid [A]$
elim. frames	F	$::=$	$[\cdot]\,v \mid [\cdot]\,[A]$	*environments*	Γ, Δ	$::=$	$\cdot \mid \Gamma, \alpha \mid \Gamma, x : A$
elim. patterns	$\hat{p}$	$::=$	$* \hookrightarrow A^- \mid p \hookrightarrow B \mid \alpha \hookrightarrow B$				

Fig. 1. $F_{\bowtie}$ Syntax

3 SYSTEM $F_{\bowtie}$

We will now make the intuitions from earlier precise. This section presents the syntax and operational semantics of $F_{\bowtie}$ in full.

3.1 Syntax

Expressions. Figure 1 presents the syntax of $F_{\bowtie}$. Expressions include standard syntax such as variables, application of functions to arguments, and application of terms to types. An underscore is used in place of a variable name when the variable is never referenced. The $c\ e$ form, tags the expression e with constructor c. The constructor c comes from some predetermined set of symbols; the metavariables l and k also range over this set, particularly when the constructor represents a record label. The notation $(e : A) \shortparallel (e' : B)$ denotes a merge of two type-annotated expressions.

We denote a subset of expressions as values: variables, constructors with values as arguments, and λ forms. We also describe *elimination frames* which are contexts in the shape of an elimination form with a hole in head position. Elimination frames play a role dual to that of values in pattern matching.

Types. The types of $F_{\bowtie}$ include type variables α, constructors applied to an argument $c\ A$, functions $A \to B$, and universal quantifiers $\forall\,\alpha.B$. Additionally, the $\bot$ type is uninhabited while $\top$ is inhabited by all well-typed values. Finally, $F_{\bowtie}$ types include unions, $A \sqcup B$, and intersections, $A \sqcap B$. We sometimes write n-ary intersections and unions such as $A_1 \sqcap \ldots \sqcap A_n$. This notation does not preclude the possibility that $n = 0$, in which case the type is $\top$ (or $\bot$ in the case of 0-ary unions). We will in §4.2 introduce some restrictions on which unions and intersections are considered to be well-formed.

Negative types are those that are introduced by λ and eliminated with strong (non-matching) elimination forms. These include functions, quantifiers, $\top$, and unions and intersections of negative types. Note that $\lambda\{\,\}$ is the trivial value of type $\top$. Elimination types are associated with elimination forms. An elimination type $\hat{A} = A$ corresponds to an application $[\cdot]\ e$ in which the argument e has type A; the elimination type $\hat{A} = [A]$ corresponds to a polymorphic instantiation $[\cdot]\,[A]$.

The typing algorithm makes use of unordered environments Γ which both contain type variables and map term variables to their types. The type associated with a term variable in an environment Γ may reference any free type variable in Γ.

Encodings. Various standard syntax can be encoded in terms of $F_{\bowtie}$'s constructs. For example, one might in theory encode **Int**, the type of 32-bit integers, as a union of 2^{32} distinct constructors.

$$A + B \;=\; \text{left } A \sqcup \text{right } B \qquad \textbf{Bool} \;=\; \top + \top \qquad \{l : A\} \;=\; l\,\top \to A$$
$$\{l_1 : A_1, \ldots, l_n : A_n\} \;=\; \{l_1 : A_1\} \sqcap \ldots \sqcap \{l_n : A_n\}$$

$$\textbf{true} \;=\; \text{left } \lambda\{\,\} \qquad \textbf{false} \;=\; \text{right } \lambda\{\,\} \qquad e.l \;=\; e\,(l\,\lambda\{\,\})$$
$$\lambda x : A.\ e \;=\; \lambda\{x : A \mapsto e\} \qquad \Lambda\alpha.e \;=\; \lambda\{\alpha \mapsto e\}$$
$$\{l_1 \mapsto e_1, \ldots, l_n \mapsto e_n\} \;=\; \lambda\{(\_ : l_1\,\top) \mapsto e_1, \ldots, (\_ : l_n\,\top) \mapsto e_n\}$$
$$\textbf{match } e \textbf{ with } \{p_1 \mapsto e_1, \ldots, p_n \mapsto e_n\} \;=\; \lambda\{p_1 \mapsto e_1, \ldots, p_n \mapsto e_n\}\ e$$

Fig. 2. Useful abbreviations

Figure 2 contains other useful abbreviations. This figure omits (as we often do) return type annotations $\hookrightarrow A$ where they are to be inferred from context. Sum types are represented by unions of the presumed-distinct left and right constructors. Records types are also encodable. The type of a field label is represented as $l\,\top$ (the constructor l with an argument of type $\top$), and a single-field record is just a function accepting that type.

Patterns. Pattern matching is a distinguishing feature of $\mathrm{F}_{\bowtie}$. There are two types of patterns. The metavariable p ranges over *value patterns*, which are essentially the patterns of Haskell or ML. In a match construct **match** v **with** $\{p_1 \mapsto e_1, \ldots, p_n \mapsto e_n\}$, the discriminee v will be deconstructed by the patterns of each $p_i \mapsto e_i$ clause.

As we have seen, the λ form contains a number of overloaded implementations of a computation, with the correct one chosen dynamically based on how the value is used by its context. Each overload is represented as a clause $\hat{p} \mapsto e$ where $\hat{p}$ is an elimination pattern (or copattern [1, 50]). We can think of λ as using these patterns to build a computation by deconstructing its immediate context—specifically, the elimination frame F. For example, consider the following value:

$$\lambda\{((x : A_1) \hookrightarrow B_1) \mapsto e_1, ((x : A_2) \hookrightarrow B_2) \mapsto e_2, (\alpha \hookrightarrow B_3) \mapsto e_3\}$$

In a frame of shape $[\cdot]\ v'$, the e_1 or e_2 overloads are selected when v' has type A_1 or A_2 respectively. (In fact, both may be be selected and merged together if A_1 and A_2 are not distinguishable.) In a context of shape $[\cdot]\,[A]$, the third clause is matched instead. Each B_i in the value is an output type annotation that describes the type of the corresponding expression e_i. Note that α is bound in B_3.

A clause $(* \hookrightarrow A^-) \mapsto v$ matches any elimination F that is valid for a value of type A^-. The elimination frame F, then becomes the context for v. In this way, the elimination pattern $*$ is a sort of "pattern variable" that places the current context around the expression in which $*$ is "bound". The point is purely to enforce that v has precisely the type A^-. This may seem unusual, but the semantics are straightforward. A term $\lambda\{(* \hookrightarrow \{l_1 : A_1\}) \mapsto \{l_1 \mapsto e_1, l_2 \mapsto e_2\}\}$ should be thought of as a wrapper around $\{l_1 \mapsto e_1, l_2 \mapsto e_2\}$ that enforces the type $\{l_1 : A_1\}$ by hiding the field l_2.

As we have seen, this hiding is achieved using η-expansion. In other words, we are taking advantage of the fact that $\lambda\{(* \hookrightarrow \{l_1 : A\}) \mapsto v\}$ is equivalent to $\lambda\{(x : l_1\,\top) \hookrightarrow A \mapsto v.l_1\}$. This explains why A^- must be a negative type: only negative types have more primitive elimination patterns to introduce them with. The syntax of value patterns also includes or- and and-patterns written $p_1|p_2$ and $p_1\&p_2$. These provide η-principles for union and intersection types. Variables in patterns are annotated with the type of value they are expected to match. They are bound in the right hand side of a clause.

$$\begin{array}{lcl} M & ::= & \{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\} \\ E & ::= & [\cdot] \mid c\,E \mid E\,e \mid v\,E \mid E[A] \\ & & \mid (E : A) \mid (E : A) \shortparallel (e : B) \\ & & \mid (v : A) \shortparallel (E : B) \end{array}$$

$$\boxed{e \mapsto e'} \qquad \textit{(small-step reduction)}$$

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']} \qquad \frac{}{F[\lambda M] \mapsto \mathrm{disp}(M, F)} \qquad \frac{}{(v : A) \mapsto v}$$

$$\frac{}{(v_1 : A_1^-) \shortparallel (v_2 : A_2^-) \mapsto \lambda\{* \hookrightarrow A_1^- \mapsto v_1, * \hookrightarrow A_2^- \mapsto v_2\}}$$

$$\mathrm{disp}(M, F) = \bigsqcup_{\substack{\hat{p} \mapsto e \in M \\ F/\hat{p} \Rightarrow \sigma \hookrightarrow B}} (\sigma(e) : B)$$

Fig. 3. $\mathsf{F}_{\bowtie}$ Operational Semantics and Auxiliary Definitions

3.2 Operational Semantics

A small-step, call-by-value operational semantics is given in Figure 3. We use M to range over λ-bodies and E to range over evaluation contexts. Evaluation proceeds under all evaluation contexts as is standard. The metavariable σ ranges over substitutions: mappings from term and type variables to values and types. A substitution may also contain at most one special mapping from $*$ to an elimination frame. The notation $\sigma(e)$ applies the substitution: every variable in the domain of σ is replaced with the corresponding type or value in e. The result is then placed in the frame F if such a frame exists in σ.

The usual β rules for type and term application are subsumed into a single rule which evaluates terms of the form $F[\lambda M]$ (strong elimination forms applied to a value) using the dispatch meta-function, disp. This procedure looks at every case $\hat{p} \mapsto e$ in M and checks whether F matches $\hat{p}$. If so, a substitution σ and output type B are obtained. The result of disp is then a merge of the expression e of every matching case with an appropriate substitution applied for pattern variables. The notation $F/\hat{p} \Rightarrow \sigma \hookrightarrow B$ used here indicates that elimination frame F matches elimination pattern $\hat{p}$; we will shortly present this process in more detail. A merge of two values evaluates to a λ-expression with two corresponding cases. Note that the type that each of the values is annotated with is remembered; the λ acts as a wrapper to hide access to any part of either value not described by its type annotation. Type annotations on expressions other than merges can be safely dropped during evaluation, since if the expression gets merged it will be annotated again in the merge.

Pattern Matching. Figure 4 gives the pattern matching algorithm. We write $v/p \Rightarrow \sigma$ to mean the value v matches the value pattern p, where σ describes the corresponding bindings for pattern variables. During the matching process, constructors are compared to a pattern structurally. Or-patterns match a value when either one of the two component patterns match. The type system will ensure that the two patterns are mutually exclusive, so there is no need for a rule that handles both patterns matching. An and-pattern matches a value when both of its subpatterns match. The results of the two matches are combined with the partial $\sigma_1 \sqcap \sigma_2$ operation. On substitutions with disjoint domains, this operation is concatenation. However, when the domains overlap it is defined only when any variables present in both results are assigned the same value.

Variable patterns annotated with negative types trivially match any value because values of negative type have no discerning tags to take advantage of in the pattern matching process. On the other hand, when the type is positive we can perform η-expansion. The last three cases of the value pattern matching definition define matching of a variable pattern in terms of patterns for the corresponding type. In this way, these variable patterns are essentially just shortcuts for more primitive patterns.

$\boxed{v/p \Rightarrow \sigma}$ *(value pattern matching)*

$$\frac{v/p \Rightarrow \sigma}{c\;v/c\;p \Rightarrow \sigma} \qquad \frac{v/p_1 \Rightarrow \sigma}{v/(p_1|p_2) \Rightarrow \sigma} \qquad \frac{v/p_2 \Rightarrow \sigma}{v/(p_1|p_2) \Rightarrow \sigma} \qquad \frac{v/p_1 \Rightarrow \sigma_1 \qquad v/p_2 \Rightarrow \sigma_2 \qquad \sigma = \sigma_1 \sqcap \sigma_2}{v/(p_1 \& p_2) \Rightarrow \sigma}$$

$$\frac{}{v/(x : A^-) \Rightarrow [x \mapsto v]} \qquad \frac{v/((x : A_1)\&(x : A_2)) \Rightarrow \sigma}{v/(x : A_1 \sqcap A_2) \Rightarrow \sigma} \qquad \frac{v/c\;(x : A) \Rightarrow [x \mapsto v']}{v/(x : c\;A) \Rightarrow [x \mapsto c\;v']}$$

$$\frac{v/((x : A_1)|(x : A_2)) \Rightarrow \sigma}{v/(x : A_1 \sqcup A_2) \Rightarrow \sigma}$$

$\boxed{F/\hat{p} \Rightarrow \sigma \hookrightarrow B}$ *(elimination pattern matching)*

$$\frac{v/p \Rightarrow \sigma}{[\cdot]\,v/(p \hookrightarrow B) \Rightarrow \sigma \hookrightarrow B} \qquad \frac{}{[\cdot][A]/(\alpha \hookrightarrow B) \Rightarrow [\alpha \mapsto A] \hookrightarrow B[A/\alpha]}$$

$$\frac{F/((x : A) \hookrightarrow B) \Rightarrow [x \mapsto v] \hookrightarrow B'}{F/(* \hookrightarrow (A \to B)) \Rightarrow [* \mapsto [\cdot]v] \hookrightarrow B'} \qquad \frac{F/(\alpha \hookrightarrow B) \Rightarrow [\alpha \mapsto A'] \hookrightarrow B'}{F/(* \hookrightarrow \forall\,\alpha.B) \Rightarrow [* \mapsto [\cdot][A']] \hookrightarrow B'}$$

$$\frac{\begin{array}{c} F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_1 \\ F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_2 \qquad \sigma = \sigma_1 \sqcap \sigma_2 \end{array}}{F/(* \hookrightarrow A_1^- \sqcup A_2^-) \Rightarrow \sigma \hookrightarrow B_1 \sqcup B_2} \qquad \frac{\begin{array}{c} F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_1 \\ F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_2 \qquad \sigma = \sigma_1 \sqcup \sigma_2 \end{array}}{F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow B_1 \sqcap B_2}$$

$$\frac{F/(* \hookrightarrow A_1^-) \Rightarrow \sigma \hookrightarrow B \qquad F/(* \hookrightarrow A_2^-) \not\Rightarrow}{F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow B} \qquad \frac{F/(* \hookrightarrow A_1^-) \not\Rightarrow \qquad F/(* \hookrightarrow A_2^-) \Rightarrow \sigma \hookrightarrow B}{F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow B}$$

$\boxed{\sigma = \sigma_1 \sqcap \sigma_2}$

$$\begin{array}{lcll}
(\sigma_1 \sqcap \sigma_2)(\alpha) & = & A_1 & \text{where } \sigma_1(\alpha) = A_1 \text{ and } \alpha \text{ not in } \sigma_2 \\
(\sigma_1 \sqcap \sigma_2)(\alpha) & = & A_2 & \text{where } \sigma_2(\alpha) = A_2 \text{ and } \alpha \text{ not in } \sigma_1 \\
(\sigma_1 \sqcap \sigma_2)(\alpha) & = & A & \text{where } \sigma_1(\alpha) = \sigma_2(\alpha) = A \\
\\
(\sigma_1 \sqcap \sigma_2)(x) & = & v_1 & \text{where } \sigma_1(x) = v_1 \text{ and } x \text{ not in } \sigma_2 \\
(\sigma_1 \sqcap \sigma_2)(x) & = & v_2 & \text{where } \sigma_2(x) = v_2 \text{ and } x \text{ not in } \sigma_1 \\
(\sigma_1 \sqcap \sigma_2)(x) & = & v & \text{where } \sigma_1(x) = \sigma_2(x) = v \\
\\
(\sigma_1 \sqcap \sigma_2)(*) & = & F_1 & \text{where } \sigma_1(\,*\,) = F_1 \text{ and } * \text{ not in } \sigma_2 \\
(\sigma_1 \sqcap \sigma_2)(*) & = & F_2 & \text{where } \sigma_2(\,*\,) = F_2 \text{ and } * \text{ not in } \sigma_1 \\
(\sigma_1 \sqcap \sigma_2)(*) & = & F & \text{where } \sigma_1(*) = \sigma_2(*) = F
\end{array}$$

$\boxed{\sigma = \sigma_1 \sqcup \sigma_2}$

$$\begin{array}{lcl}
(\sigma_1 \sqcup \sigma_2)(\alpha) & = & A \\
 & & \text{where } \sigma_1(\alpha) = \sigma_2(\alpha) = A \\
(\sigma_1 \sqcup \sigma_2)(x) & = & v \\
 & & \text{where } \sigma_1(x) = \sigma_2(x) = v \\
(\sigma_1 \sqcup \sigma_2)(*) & = & F \\
 & & \text{where } \sigma_1(*) = \sigma_2(*) = F
\end{array}$$

Fig. 4. Pattern Matching

A value may match a pattern variable annotated with a negative intersection or union type either via the negative variable case or via the intersection/union case. The following lemma makes clear that we get the same substitution as output either way.

Lemma 3.1. *If $v/(x : A) \Rightarrow \sigma$ then $\sigma = [x \mapsto v]$.*

Thus, this issue is not a source of nondeterminism. However, the treatment of or-patterns does introduce nondeterminism. A value v that matches the patterns p_1 and p_2 in two different ways can also match $p_1|p_2$ in both ways. The type system will rule out such nondeterministic examples

$\boxed{A \leq B}$ *(declarative subtyping)*

$$\frac{}{A \leq A} \qquad \frac{A_1 \leq A_2 \qquad A_2 \leq A_3}{A_1 \leq A_3} \qquad \frac{A \leq B}{c\,A \leq c\,B} \qquad \frac{A \leq B}{C \to A \leq C \to B} \qquad \frac{A \leq B}{\forall\alpha.A \leq \forall\alpha.B}$$

$$\frac{A_2 \leq A_1}{A_1 \to B \leq A_2 \to B} \qquad \frac{A_1 \leq B}{A_1 \sqcap A_2 \leq B} \qquad \frac{A_2 \leq B}{A_1 \sqcap A_2 \leq B} \qquad \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \sqcap B_2}$$

$$\frac{}{(C \to A) \sqcap (C \to B) \leq (C \to A \sqcap B)} \qquad \frac{}{(c\,A) \sqcap (c\,B) \leq c\,(A \sqcap B)}$$

$$\frac{}{(\forall\alpha.A) \sqcap (\forall\alpha.B) \leq (\forall\alpha.A \sqcap B)} \qquad \frac{}{\forall\alpha.(A \sqcup B) \leq (\forall\alpha.A) \sqcup (\forall\alpha.B)} \qquad \frac{}{c\,(A \sqcup B) \leq (c\,A) \sqcup (c\,B)}$$

$$\frac{}{(A_1 \to B) \sqcap (A_2 \to B) \leq (A_1 \sqcup A_2) \to B} \qquad \frac{A \leq B_1}{A \leq B_1 \sqcup B_2} \qquad \frac{}{\bot \leq B} \qquad \frac{A \leq B_2}{A \leq B_1 \sqcup B_2} \qquad \frac{}{A \leq \top}$$

$$\frac{}{(A \sqcup B) \sqcap C \leq (A \sqcap C) \sqcup (B \sqcap C)} \qquad \frac{A_1 \leq B \qquad A_2 \leq B}{A_1 \sqcup A_2 \leq B}$$

Fig. 5. Declarative subtyping

by ensuring only one side of an or-pattern ever matches. In other words, well-typed or-patterns are exclusive and pattern matching is deterministic over well-typed patterns.

Dually, $F/\hat{p} \Rightarrow \sigma \hookrightarrow B$ means that the elimination frame F matches the pattern $\hat{p}$, where the substitution σ contains bindings for pattern variables and may additionally contain an elimination frame. The negation of this defintion, $F/\hat{p} \not\Rightarrow$ means that F does not match $\hat{p}$. The type B is the expected return type of the case of the λ-expression in which $\hat{p}$ appears.

The $*$ elimination patterns behave in a dual way to variable patterns. To understand them, it is helpful to see an example. Consider the term $e = \lambda\{(* \hookrightarrow (A \to B)) \mapsto v\}\ v'$. Here, $F = [\cdot]v'$ and $\hat{p} = * \hookrightarrow A \to B$. The matching procedure proceeds by recursively matching F against the pattern $(x : A) \hookrightarrow B$. In the end, the result is $[\, * \mapsto [\cdot]v']$ with a return type of B. Thus, $e \mapsto (v\ v' : B)$.

As with variable patterns for positive types, $*$ patterns for function and polymorphic types are defined in terms of more primitive patterns. To keep $F_{\bowtie}$'s typechecking deterministic, however, the language includes neither and-elimination patterns nor or-elimination patterns. Thus, $*$ patterns for union and intersection types must be treated specially. This in effect offloads the nondeterminism to the subtyping algorithm. The dual role of elimination patterns when compared with value patterns can be counterintuitive when it comes to union and intersection types. For example, a value matches a value pattern $x : A_1 \sqcup A_2$ when it matches one of either $x : A_1$ or $x : A_2$ but an elimination frame matches an elimination pattern $* \hookrightarrow A_1 \sqcup A_2$ when it matches *both* $* \hookrightarrow A_1$ and $* \hookrightarrow A_2$. Similarly, $\sigma_1 \sqcap \sigma_2$ is used to combine substitutions during elimination matching for union types while the dual $\sigma_1 \sqcup \sigma_2$ operation is used for intersection types.

4 TYPE SYSTEM

We now present the type system of $F_{\bowtie}$. It aims to guarantee both a conventional type soundness property as well as determinism of reduction.

4.1 Subtyping

Types form a bounded distributive lattice under the subtyping relation (written $A \leq B$) defined in Figure 5. The standard subtyping rules for function, union, and intersection types are included in

$\boxed{A \bowtie B}$ *(mergeability)* $\boxed{A \bowtie_{\mathrm{ax}} B}$ *(merg. axioms)*

$$\frac{A \diamond B}{A \to A' \bowtie B \to B'} \qquad \frac{B \bowtie B'}{A \to B \bowtie A \to B'} \qquad \frac{A \bowtie B \qquad A' \bowtie B}{A \sqcap A' \bowtie B} \qquad \frac{}{\top \bowtie_{\mathrm{ax}} B}$$

$$\frac{A \bowtie B}{\forall\,\alpha.A \bowtie \forall\,\alpha.B} \qquad \frac{A \bowtie B \qquad A' \bowtie B}{A \sqcup A' \bowtie B} \qquad \frac{A \bowtie_{\mathrm{ax}} B}{A \bowtie B} \qquad \frac{B \bowtie A}{A \bowtie B} \qquad \frac{}{A \to A' \bowtie_{\mathrm{ax}} \forall\,\alpha.B}$$

$\boxed{A \diamond B}$ *(distinguishability)* $\boxed{A \diamond_{\mathrm{ax}} B}$ *(dist. axioms)*

$$\frac{A \diamond B}{c\ A \diamond c\ B} \qquad \frac{A \diamond B \qquad A' \diamond B}{A \sqcup A' \diamond B} \qquad \frac{A \diamond_{\mathrm{ax}} B}{A \diamond B} \qquad \frac{B \diamond A}{A \diamond B} \qquad \frac{}{\bot \diamond_{\mathrm{ax}} B}$$

$$\frac{A \diamond B' \qquad B \leq B'}{A \diamond B} \qquad \frac{c_1 \neq c_2}{c_1\ A \diamond_{\mathrm{ax}} c_2\ B}$$

Fig. 6. Disjointness Relations

this definition or derivable from the rules that are. For example, the rule

$$\frac{B \leq A \qquad A' \leq B'}{A \to A' \leq B \to B'}$$

is derivable from transitivity and the co- and contravariance rules for the function type constructor. Subtyping is defined to be reflexive and transitive. Additionally, unions and intersections distribute over most other types. The following type equivalences hold:

$$\begin{array}{rclcrcl} c\ (A \sqcup B) & \equiv & (c\ A) \sqcup (c\ B) & \qquad & \forall\,\alpha.(A \sqcap B) & \equiv & (\forall\,\alpha.A) \sqcap (\forall\,\alpha.B) \\ (C \to A) \sqcap (C \to B) & \equiv & C \to (A \sqcap B) & & (A \to C) \sqcap (B \to C) & \equiv & (A \sqcup B) \to C \end{array}$$

The notation $A \equiv B$ denotes subtyping in both directions.

Due to the presence of overloading, we do not in general have $(A \to A') \sqcap (B \to B') \leq (A \sqcup B) \to (A' \sqcap B')$. Consider an overloaded function which produces an integer when given an integer and produces a boolean when given a boolean. It has type $(\mathbf{Int} \to \mathbf{Int}) \sqcap (\mathbf{Bool} \to \mathbf{Bool})$. It cannot be cast to the type $(\mathbf{Int} \sqcup \mathbf{Bool}) \to (\mathbf{Int} \sqcap \mathbf{Bool})$ because given only one of an integer or boolean, it is able to produce a value of only one of those types as a result—not both.

4.2 Disjoint and Well-Formed Types

The mergeability and distinguishability relations are defined inductively in Figure 6.

Mergeability. The idea of mergeability is to relate two types A and B when there is no potentially ambiguous operation shared between them. Consider the following derivable rules.

$$\frac{l \neq k}{\{l : A\} \bowtie \{k : B\}} \qquad \frac{A \bowtie B}{\{l : A\} \bowtie \{l : B\}}$$

The first states that any records with different labels are mergeable, because such types share no operations in common. The second states that even when two types both contain the field l, they may be merged if the contents of the field themselves are mergeable. In this case, some later part of every operation on the two types (after the projection of l) will disambiguate a use of this type.

These two rules follow directly from the first two rules of Figure 6. Functions can be merged when their arguments are distinguishable to form an overloaded function. Two functions that take the same argument type can also be merged when their outputs can be. We previously saw an example in which merging a constructor of type $\mathbf{Int} \to \{age : \mathbf{Int}\}$ with another of type $\mathbf{Int} \to \{name : \mathbf{String}\}$ yields a combined constructor of type $\mathbf{Int} \to \{age : \mathbf{Int}, name : \mathbf{String}\}$

This style of merge supports nested composition and is also possible for universally quantified types. A merge of values of types A and A' (itself having type $A \sqcap A'$) can be merged with a third value of type B when both A and A' are mergeable with B. This ensures that operations on B do not conflict with those of A or A'. Conservatively, a union is also mergeable with another type when both of its components is. Since a value of type $\top$ cannot be used in any way, this type can trivially be merged with any other type. Finally, because application of terms to types is explicit in $F_{\bowtie}$, it is safe for function types and universal quantifiers to be mergeable with each other. In systems where universal quantifiers have no explicit elimination form, this may not be possible.

Distinguishability. Figure 6 also gives the distinguishability relation. Two types A and B are distinguishable when patterns of each type do not match values of the other type. In other words, distinguishability means that the pattern matching procedure can tell values of one type from values of the other. Most of the distinguishability rules are straightforward or precisely dual to those of mergeability. A key rule states that unequal type constructors are distinguishable. Another rule closes distinguishability over the subtyping relation. As shown in §6.2, this property is important for soundness of the type system.

Well-Formed Types. The well-formedness relation on types, written $\Gamma \vdash A$ ensures that:

(1) Only the type variables in Γ appear free in A.
(2) The components of every intersection in A are mergeable.
(3) The components of every union in A are distinguishable.

The cases for unions and intersection types are the only non-standard part of the definition of this relation, which is given in the appendix.

The disjointness restrictions considerably simplify the type soundness argument by enabling inversion principles that we will see in §6. In the case of unions, the restrictions similarly simplify the pattern matching procedure. Consider matching the value $v = \lambda\{(* \hookrightarrow A \sqcup B) \mapsto v'\}$ against the pattern $p = (x : A)|(x : B)$ The matching procedure we saw earlier would try to match v against $x : A$ and $x : B$ separately. But these matches are potentially ill typed: $A \sqcup B$ is in general a subtype of neither A nor B. With the disjointness restriction on unions, it can be proven that every value of type $A \sqcup B$ either has type A or type B. Thus, one of the two matches is well typed.

A notable drawback of the disjointness restriction is that it precludes employing intersections as type refinements [25]. It appears that, under a notion of subtyping with the right distributivity rules, non-distinguishable unions (and their patterns) can often be simplified to distinguishable ones. Future work may study such an approach to support arbitrary union and intersection types in $F_{\bowtie}$. For our present purposes, disjoint intersections and unions capture the essence of what is necessary to investigate overloading and extensible data types.

4.3 Typing

Figure 7 defines $F_{\bowtie}$'s type system. The presentation uses the notation of bidirectional typing to ensure that it is algorithmic. The judgment $\Gamma \vdash e \Rightarrow A$ means that A is the least type which can be assigned to e. In other words, A is the principal type of e. Meanwhile, $\Gamma \vdash e \Leftarrow A$ signifies that e has the type A (and potentially some subtypes of A). The rules themselves—in particular for function application—do not follow the standard recipe of bidirectional typing [22], however. This is because in a setting with rich enough types, more conventional typing rules are not closed under η-expansion. Since η-expansion occurs at runtime in $F_{\bowtie}$, this would ultimately break the type preservation lemma.

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(expression type synthesis/checking)*

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \;\Rightarrow\; A} \qquad \frac{\Gamma \vdash e \;\Rightarrow\; A}{\Gamma \vdash c\, e \;\Rightarrow\; c\, A} \qquad \frac{\Gamma \vdash e \;\Leftarrow\; A}{\Gamma \vdash (e : A) \;\Rightarrow\; A} \qquad \frac{\Gamma \vdash e \;\Rightarrow\; A \qquad A \leq B}{\Gamma \vdash e \;\Leftarrow\; B}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_2 \;\Rightarrow\; A_2 \\ \Gamma \vdash e_1 \;\Rightarrow\; A_1 \qquad \mathrm{disp}(A_1, A_2) \Rightarrow B\end{array}}{\Gamma \vdash e_1\; e_2 \;\Rightarrow\; B} \qquad \frac{\begin{array}{c}\Gamma \vdash A_2 \\ \Gamma \vdash e \;\Rightarrow\; A_1 \qquad \mathrm{disp}(A_1, [A_2]) \Rightarrow B\end{array}}{\Gamma \vdash e\,[A_2] \;\Rightarrow\; B}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1^- \qquad \Gamma \vdash A_2^- \qquad A_1^- \bowtie A_2^- \\ \Gamma \vdash e_1 \;\Leftarrow\; A_1^- \qquad \Gamma \vdash e_2 \;\Leftarrow\; A_2^-\end{array}}{\Gamma \vdash (e_1 : A_1^-) \parallel (e_2 : A_2^-) \;\Rightarrow\; A_1^- \sqcap A_2^-} \qquad \frac{\begin{array}{c}\Gamma; \hat{p}_1 \vdash e_1 \Rightarrow A_1, \ldots, \Gamma; \hat{p}_n \vdash e_n \Rightarrow A_n \\ A_1 \bowtie \ldots \bowtie A_n\end{array}}{\Gamma \vdash \lambda\{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\} \;\Rightarrow\; A_1 \sqcap \ldots \sqcap A_n}$$

$\boxed{\Gamma \vdash p \Rightarrow A \dashv \Delta}$ *(value pattern typing)*

$$\frac{\Gamma \vdash A}{\Gamma \vdash (x : A) \Rightarrow A \dashv x : A} \qquad \frac{\Gamma \vdash p_1 \Rightarrow A_1 \dashv \Delta_1 \qquad \Gamma \vdash p_2 \Rightarrow A_2 \dashv \Delta_2 \qquad A_1 \diamond A_2}{\Gamma \vdash p_1 | p_2 \Rightarrow A_1 \sqcup A_2 \dashv \Delta_1 \sqcup \Delta_2}$$

$$\frac{\Gamma \vdash p \Rightarrow A \dashv \Delta}{\Gamma \vdash c\, p \Rightarrow c\, A \dashv \Delta} \qquad \frac{\Gamma \vdash p_1 \Rightarrow A_1 \dashv \Delta_1 \qquad \Gamma \vdash p_2 \Rightarrow A_2 \dashv \Delta_2 \qquad A_1 \bowtie A_2 \qquad p_1 \sim p_2}{\Gamma \vdash p_1 \& p_2 \Rightarrow A_1 \sqcap A_2 \dashv \Delta_1 \sqcap \Delta_2}$$

$\boxed{\Gamma; \hat{p} \vdash e \Rightarrow B}$ *(λ clause typing)* $\boxed{\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta}$ *(elimination pattern typing)*

$$\frac{\begin{array}{c}\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta \\ \Gamma, \Delta \vdash e \;\Leftarrow\; B\end{array}}{\Gamma; \hat{p} \vdash e \Rightarrow A} \qquad \frac{\Gamma \vdash B^-}{\Gamma \vdash (* \hookrightarrow B^-) \Rightarrow B^- \hookrightarrow B^- \dashv \cdot}$$

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash p \Rightarrow A \dashv \Delta}{\Gamma \vdash (p \hookrightarrow B) \Rightarrow (A \to B) \hookrightarrow B \dashv \Delta} \qquad \frac{\Gamma, \alpha \vdash B}{\Gamma \vdash (\alpha \hookrightarrow B) \Rightarrow \forall \alpha.B \hookrightarrow B \dashv \alpha}$$

$\boxed{p_1 \sim p_2}$ *(value pattern consistency)*

$$\frac{}{(x : A) \sim (y : B)} \qquad \frac{p_1 \sim p_2}{c\, p_1 \sim c\, p_2} \qquad \frac{p_1 \sim p_2 \qquad p_1' \sim p_2}{p_1 \& p_1' \sim p_2} \qquad \frac{p_1 \sim p_2 \qquad p_1 \sim p_2'}{p_1 \sim p_2 \& p_2'}$$

Fig. 7. Typing Rules

Instead, applications are typed with the help of a type-level dispatch operator $\mathrm{disp}(A, \hat{B}) \Rightarrow C$; this is similar in style to the *apptype* function in the system of Freeman and Pfenning [25]. The disp metafunction takes the type of a function A and a context $\hat{B}$ and computes the output type C.

Lemma 4.1 (Soundness and Completeness of Type-level Dispatch).

(1) *We have* $\mathrm{disp}(A, B) \Rightarrow C$ *iff* C *is the least type such that* $A \leq B \to C$.
(2) *We have* $\mathrm{disp}(A, [B]) \Rightarrow C$ *iff* C *is the least type such that there exists* A' *where* $A \leq \forall \alpha.A'$ *and* $C \leq A'[B/\alpha]$.

Due to overloading and union types, the implementation of this operation is non-trivial. It must statically determine which overloads of a function will execute. We will cover this in §5. Merges are typed by taking the intersection of the types they are annotated with. These types must be mergeable. Types are assigned to a λ-value by obtaining a type for each clause in its body, ensuring that the inferred types of all clauses are disjoint, and constructing an intersection of those types. This involves an auxiliary judgment $\Gamma; \hat{p} \vdash e \Rightarrow A$ which infers the type A of a single clause $\hat{p} \mapsto e$.

Inferring the type of a clause involves determining the types of the bound variables in $\hat{p}$ as well as this pattern's return type annotation. These are used to type the expression e.

The next two judgments type patterns. Value patterns are typed with the judgment $\Gamma \vdash p \Rightarrow A \dashv \Delta$. The environment Γ contains the type variables free in p (as well as A, B, and Δ). The type A is the type of value that p matches (i.e. the type that p eliminates). Lastly, Δ contains the type and term variables bound by p.

An or-pattern binds only the variables that around bound by both subpatterns. This ensures that at runtime there is always a value to assign to every bound variable, even if only one subpattern matches. On the other hand, an and-pattern binds the variables that are mentioned on either side. The $\Delta_1 \sqcup \Delta_2$ and $\Delta_1 \sqcap \Delta_2$ operation take care of building the output environments in the typing of these kinds of patterns. Their full definitions, reminiscent of classical record subtyping, are given in the appendix.

One subtle point in the definition of value pattern matching is required to ensure determinism. We need to rule out certain patterns like $(x : \top) \& c_0\ (x : (c_1\ \top \sqcup c_2\ \top))$ that bind the same variable twice to distinct values. This is achieved by requiring that the two sides of an and-pattern are *consistent*, written $p_1 \sim p_2$. Elimination patterns are typed with the judgment $\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta$. The environments Γ and Δ perform the same functions as for value patterns. The type A, this time, is the type of context that $\hat{p}$ matches (i.e. the type that $\hat{p}$ introduces). Since elimination patterns contain output type annotations, B is type of the expression bound under $\hat{p}$.

Example. Consider the value $\lambda\{(c\ (f : \mathbf{Bool} \to \mathbf{Bool}) \hookrightarrow \mathbf{Int}) \mapsto \mathbf{if}\ f\ \mathbf{true}\ \mathbf{then}\ 1\ \mathbf{else}\ 0\}$ of type $(c\ (\mathbf{Bool} \to \mathbf{Bool})) \to \mathbf{Int}$. This function pattern matches on its input to extract a function f from underneath a constructor c. Then it calls the function and returns an integer based on the result.

The elimination pattern $\hat{p} = c\ (f : \mathbf{Bool} \to \mathbf{Bool}) \hookrightarrow \mathbf{Int}$ is typed with the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash (f : \mathbf{Bool} \to \mathbf{Bool}) \Rightarrow \mathbf{Bool} \to \mathbf{Bool} \dashv f : \mathbf{Bool} \to \mathbf{Bool}}}{\cdot \vdash c\ (f : \mathbf{Bool} \to \mathbf{Bool}) \Rightarrow c\ (\mathbf{Bool} \to \mathbf{Bool}) \dashv f : \mathbf{Bool} \to \mathbf{Bool}}}{\cdot \vdash \hat{p} \Rightarrow (c\ (\mathbf{Bool} \to \mathbf{Bool}) \to \mathbf{Int}) \hookrightarrow \mathbf{Int} \dashv f : \mathbf{Bool} \to \mathbf{Bool}}$$

At the bottom, in the elimination pattern judgment, note that the type **Int** appears twice. The first occurrence is to the right of the $\to$ in the type, which becomes the type of the entire function. The second is to the right of the $\hookrightarrow$ as the return type—that is, the type of the body $\mathbf{if}\ f\ \mathbf{true}\ \mathbf{then}\ 1\ \mathbf{else}\ 0$.

5 TYPING ALGORITHM

The subtyping and type-level dispatch relations have thus far been presented only in a declarative manner. We here describe the algorithms required to compute these relations. As both of these play crucial roles in the type system, their implementations are necessary to show the typing relation is computable.

5.1 Subtyping and Splitting

The main difficulty in our subtyping algorithm design arises from the distributivity over intersections and unions. A conventional technique adapted by many of the systems that support both intersection and union types is to convert types to standard forms before analyzing them [12, 25]. We instead follow the idea of *splitting types* [28], which normalizes types on the fly, and extend its subtyping to universally quantified and record types. The two splitting algorithms shown at the top of Figure 8 each take a type and transform it into an equivalent union or intersection type, respectively. The type $C \to A \sqcap B$, for instance, is equivalent to $(C \to A) \sqcap (C \to B)$, as we can

$\boxed{\mathrm{split_u}(\hat{A}) \Rightarrow A_1 \sqcup A_2}$ *(union splitting)*

$$\frac{}{\mathrm{split_u}(A \sqcup B) \Rightarrow A \sqcup B} \qquad \frac{\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2}{\mathrm{split_u}(\forall\,\alpha.A) \Rightarrow (\forall\,\alpha.A_1) \sqcup (\forall\,\alpha.A_2)} \qquad \frac{\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2}{\mathrm{split_u}(c\ A) \Rightarrow c\ A_1 \sqcup c\ A_2}$$

$$\frac{\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2}{\mathrm{split_u}(A \sqcap B) \Rightarrow (A_1 \sqcap B) \sqcup (A_2 \sqcap B)} \qquad \frac{\mathrm{split_u}(A) \not\Rightarrow \qquad \mathrm{split_u}(B) \Rightarrow B_1 \sqcup B_2}{\mathrm{split_u}(A \sqcap B) \Rightarrow (A \sqcap B_1) \sqcup (A \sqcap B_2)}$$

$\boxed{\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2}$ *(intersection splitting)*

$$\frac{}{\mathrm{split_i}(A \sqcap B) \Rightarrow A \sqcap B} \qquad \frac{\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2}{\mathrm{split_i}(\forall\,\alpha.A) \Rightarrow (\forall\,\alpha.A_1) \sqcap (\forall\,\alpha.A_2)} \qquad \frac{\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2}{\mathrm{split_i}(c\ A) \Rightarrow c\ A_1 \sqcap c\ A_2}$$

$$\frac{\mathrm{split_i}(B) \Rightarrow B_1 \sqcap B_2}{\mathrm{split_i}(A \to B) \Rightarrow (A \to B_1) \sqcap (A \to B_2)} \qquad \frac{\mathrm{split_i}(B) \not\Rightarrow \qquad \mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2}{\mathrm{split_i}(A \to B) \Rightarrow (A_1 \to B) \sqcap (A_2 \to B)}$$

$$\frac{\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2}{\mathrm{split_i}(A \sqcup B) \Rightarrow (A_1 \sqcup B) \sqcap (A_2 \sqcup B)} \qquad \frac{\mathrm{split_i}(A) \not\Rightarrow \qquad \mathrm{split_i}(B) \Rightarrow B_1 \sqcap B_2}{\mathrm{split_i}(A \sqcup B) \Rightarrow (A \sqcup B_1) \sqcap (A \sqcup B_2)}$$

$\boxed{A \leq_{\mathrm{alg}} B}$ *(algorithmic subtyping)*

$$\frac{}{A \leq_{\mathrm{alg}} A} \qquad \frac{A \leq_{\mathrm{alg}} B}{c\ A \leq_{\mathrm{alg}} c\ B} \qquad \frac{B_1 \leq_{\mathrm{alg}} A_1 \qquad A_2 \leq_{\mathrm{alg}} B_2}{A_1 \to A_2 \leq_{\mathrm{alg}} B_1 \to B_2} \qquad \frac{A \leq_{\mathrm{alg}} B}{\forall\,\alpha.A \leq_{\mathrm{alg}} \forall\,\alpha.B} \qquad \frac{}{\bot \leq_{\mathrm{alg}} A} \qquad \frac{}{A \leq_{\mathrm{alg}} \top}$$

$$\frac{\begin{array}{c}\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2 \\ A_1 \leq_{\mathrm{alg}} B\end{array}}{A \leq_{\mathrm{alg}} B} \qquad \frac{\begin{array}{c}\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2 \\ A_2 \leq_{\mathrm{alg}} B\end{array}}{A \leq_{\mathrm{alg}} B} \qquad \frac{\begin{array}{c}\mathrm{split_i}(B) \Rightarrow B_1 \sqcap B_2 \\ A \leq_{\mathrm{alg}} B_1 \qquad A \leq_{\mathrm{alg}} B_2\end{array}}{A \leq_{\mathrm{alg}} B} \qquad \frac{\begin{array}{c}\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2 \\ A_1 \leq_{\mathrm{alg}} B \qquad A_2 \leq_{\mathrm{alg}} B\end{array}}{A \leq_{\mathrm{alg}} B}$$

$$\frac{\mathrm{split_u}(B) \Rightarrow B_1 \sqcup B_2 \qquad A \leq_{\mathrm{alg}} B_1}{A \leq_{\mathrm{alg}} B} \qquad \frac{\mathrm{split_u}(B) \Rightarrow B_1 \sqcup B_2 \qquad A \leq_{\mathrm{alg}} B_2}{A \leq_{\mathrm{alg}} B}$$

$\boxed{\mathrm{disp}(A, \hat{B}) \Rightarrow C}$ *(type-level dispatch)*

$$\frac{\begin{array}{c}\mathrm{split_u}(B) \Rightarrow B_1 \sqcup B_2 \\ \mathrm{disp}(A, B_1) \Rightarrow C_1 \qquad \mathrm{disp}(A, B_2) \Rightarrow C_2\end{array}}{\mathrm{disp}(A, B) \Rightarrow C_1 \sqcup C_2} \qquad \frac{\mathrm{split_u}(\hat{B}) \not\Rightarrow}{\mathrm{disp}(\bot, \hat{B}) \Rightarrow \bot} \qquad \frac{\mathrm{split_u}(B) \not\Rightarrow \qquad B \leq A}{\mathrm{disp}(A \to A', B) \Rightarrow A'}$$

$$\frac{}{\mathrm{disp}(\forall\,\alpha.A, [B]) \Rightarrow A[B/\alpha]} \qquad \frac{\begin{array}{c}\mathrm{split_u}(\hat{B}) \not\Rightarrow \\ \mathrm{disp}(A_1, \hat{B}) \Rightarrow C_1 \qquad \mathrm{disp}(A_2, \hat{B}) \Rightarrow C_2\end{array}}{\mathrm{disp}(A_1 \sqcup A_2, \hat{B}) \Rightarrow C_1 \sqcup C_2} \qquad \frac{\begin{array}{c}\mathrm{split_u}(\hat{B}) \not\Rightarrow \\ \mathrm{disp}(A_1, \hat{B}) \Rightarrow C_1 \qquad \mathrm{disp}(A_2, \hat{B}) \not\Rightarrow\end{array}}{\mathrm{disp}(A_1 \sqcap A_2, \hat{B}) \Rightarrow C_1}$$

$$\frac{\begin{array}{c}\mathrm{split_u}(\hat{B}) \not\Rightarrow \\ \mathrm{disp}(A_1, \hat{B}) \not\Rightarrow \qquad \mathrm{disp}(A_2, \hat{B}) \Rightarrow C_2\end{array}}{\mathrm{disp}(A_1 \sqcap A_2, \hat{B}) \Rightarrow C_2} \qquad \frac{\begin{array}{c}\mathrm{split_u}(\hat{B}) \not\Rightarrow \\ \mathrm{disp}(A_1, \hat{B}) \Rightarrow C_1 \qquad \mathrm{disp}(A_2, \hat{B}) \Rightarrow C_2\end{array}}{\mathrm{disp}(A_1 \sqcap A_2, \hat{B}) \Rightarrow C_1 \sqcap C_2}$$

Fig. 8. Subtyping and Related Algorithms

derive $\mathrm{split_i}(C \to A \sqcap B) \Rightarrow (C \to A) \sqcap (C \to B)$. We use a negated arrow to express that a type is not splittable, e.g., $\mathrm{split_u}(\mathbf{Int}) \not\Rightarrow$.

These rules encode all of the distributivity in subtyping and therefore take the burden off the algorithmic subtyping rules (defined in the middle of Figure 8). Once we replace these type splitting judgments by equations, i.e., write $A = A_1 \sqcup A_2$ and $B = B_1 \sqcup B_2$ instead of $\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2$ and $\mathrm{split_u}(B) \Rightarrow B_1 \sqcup B_2$, the distributivity is fully eliminated, and we can see that the algorithmic subtyping rules follows a standard presentation.

$$\text{value types} \quad V \ ::= \ c\ V \mid A^{-}$$

$$\text{elim. frame types} \quad \hat{V} \ ::= \ V \mid [A]$$

$\boxed{\Gamma \vdash F \Rightarrow \hat{V}}$ *(elimination frame typing)*

$$\frac{\Gamma \vdash v \Rightarrow V}{\Gamma \vdash [\cdot]\, v \Rightarrow V} \qquad \frac{}{\Gamma \vdash [\cdot]\,[A] \Rightarrow [A]}$$

$\boxed{\Gamma; A \vdash F \Leftrightarrow B}$ *(elimination type synthesis/checking)*

$$\frac{\Gamma \vdash F \Rightarrow \hat{V} \qquad \mathrm{disp}(A, \hat{V}) \Rightarrow B}{\Gamma; A \vdash F \Rightarrow B} \qquad \frac{\Gamma; A \vdash F \Rightarrow B' \qquad B' \le B}{\Gamma; A \vdash F \Leftarrow B}$$

Fig. 9. Definitions Used in Soundness Proof

5.2 Typing Application

The meta function $\mathrm{disp}(A, B) \Rightarrow C$ is used to calculate the most precise output type for both applications and type applications. The function type A has a very general form as it describes overloaded functions or nested compositions. Metavariable $\hat{B}$ captures two cases: it is either the term's type in an ordinary application, which we use B to represent, or the type argument in a type application, denoted by $[B]$. The definitions ensure that type arguments are not splittable. In other words, $\mathrm{split_u}([B]) \not\Rightarrow$ holds trivially.

The base case for a type application is when a universal quantified type or a bottom type meets a type argument. Unlike in the function application rule, which checks subtyping, there is no side condition for type application, so we do not need to process the type argument. The dispatch function looks into all the universally quantified types in A, substitutes the type argument into their bodies and then re-composes them back into the original structure (all the bottom types are also kept in the process). For example, we have: $\mathrm{disp}((\bot \sqcup \forall\, \alpha.A), B) \Rightarrow \bot \sqcup A[B/\alpha]$.

For an intersection type to be applicable, the dispatch function requires that at least one part of it is applicable. Imagine an overloaded function with two implementations typed by $A_1 \to A_2$ and $B_1 \to B_2$, respectively. The argument only needs to satisfy either A_1 or B_1 for at least one implementation to be applicable in the runtime. For unions, it is mandatory that both parts are applicable, as unions only guarantee that one side is satisfied.

Dispatch splits unions in the argument type eagerly. To see why, consider the following situation, where an intersection of two function types is applied to a union. The two function types can each take one of the possible argument types, but when directly compared with the whole argument type, neither can have the subtyping condition satisfied.

$$\mathrm{disp}(((\mathbf{Int} \to A_1) \sqcap (\mathbf{Bool} \to A_2)), \mathbf{Int} \sqcup \mathbf{Bool}) \Rightarrow A_1 \sqcup A_2$$

After our dispatch function tears the argument type apart, both applications can proceed.

As the type of a term evolves during reduction, the dispatch function must be monotonic to ensure soundness. Specifically, the subtypes of two applicable types must also be applicable, and the return type computed by the dispatch function should be a subtype of the original return type.

Lemma 5.1 (Monotonicity of Type-Level Dispatch).

(1) *If* $\mathrm{disp}(A, \hat{B}) \Rightarrow C$ *and* $A' \le A$ *then there exists* $C' \le C$ *such that* $\mathrm{disp}(A', \hat{B}) \Rightarrow C'$.
(2) *If* $\mathrm{disp}(A, B) \Rightarrow C$ *and* $B' \le B$ *then there exists* $C' \le C$ *such that* $\mathrm{disp}(A, B') \Rightarrow C'$.

Proof. Follows from the type-level dispatch specification given by Lemma 4.1. □

6 RESULTS

We prove the soundness of $F_{\bowtie}$'s type system using a progress and preservation argument [48]. Given the presence of copatterns, our proof structure in some ways resembles that of Abel et al. [1]. It requires a standard substitution lemma as well as inversion and canonical forms lemmas.

Thanks to the presence of elimination pattern matching, we need to reason not only about the shape of well-typed values, but also well-typed elimination frames. Figure 9 defines typing judgments for elimination frames. The type $\hat{A}$ of an elimination frame F is given by the judgment $\Gamma \vdash F \Rightarrow \hat{V}$. An alternative method of typing elimination frames is by thinking of them as contexts which, when filled with an expression of type A, produce an expression of type B. In this case, we write $\Gamma; A \vdash F \Rightarrow B$. With these definitions, we state our canonical forms lemma. This lemma describes the shape of a value or elimination frame given its type.

Lemma 6.1 (Canonical Forms).

(1) *If* $\cdot \vdash v \Leftarrow c\ A$ *then there exists* v' *such that* $v = c\ v'$.
(2) *If* $\cdot \vdash v \Leftarrow A \to B$ *then there exists* M *such that* $v = \lambda M$.
(3) *If* $\cdot \vdash v \Leftarrow \forall \alpha.A$ *then there exists* M *such that* $v = \lambda M$.
(4) *There is no* v *such that* $\cdot \vdash v \Leftarrow \bot$.
(5) *If* $\cdot; A \to A' \vdash F \Rightarrow B$ *then there exists* v *such that* $F = [\cdot]\, v$.
(6) *If* $\cdot; \forall \alpha.A \vdash F \Rightarrow B$ *then there exists* C *such that* $F = [\cdot]\,[C]$
(7) *There are no* F *and* B *such that* $\cdot; \top \vdash F \Rightarrow B$.

We also have a number of additional inversion principles for the typing relation. Most notable are those for union and intersection types. Inversion principles for unrestricted union types have long been desired, but remain an active area of study [14]. Thanks to disjointness, we do, in fact, obtain:

Lemma 6.2. *If* $\Gamma \vdash v \Leftarrow A_1 \sqcup A_2$ *and* $A_1 \nLeftrightarrow A_2$ *then* $\Gamma \vdash v \Leftarrow A_1$ *or* $\Gamma \vdash v \Leftarrow A_2$.

The distinguishability premise is crucial to this lemma, as is the fact that v is a value. The lemma does not hold without these constraints. Dually, we also have a property about frames that eliminate intersections of mergeable types.

Lemma 6.3. *Suppose* $\cdot \vdash A_1 \sqcap A_2$. *If* $\cdot; A_1 \sqcap A_2 \vdash F \Rightarrow B$ *then there exists a* B' *such that either* $\cdot; A_1 \vdash F \Rightarrow B'$ *or* $\cdot; A_2 \vdash F \Rightarrow B'$.

6.1 Progress

To prove progress, we must show that every closed, well-typed term is either a value or may take a step. The key *coverage* lemma needed here is that if a value v and a pattern p have type A, then v should successfully match p. An analogous property for elimination frames and elimination patterns is also needed. There are a couple issues. First, A is not necessarily the principal type of v, so coverage is fundamentally a property relating subtyping to pattern matching. The subtyping relation is relatively complex, so proceeding directly by induction on it is tricky. Second, the type A does not uniquely determine the shape of the pattern p, which would require a direct proof to analyze many cases.

To deal with these issues, we define the value coverage relation $V/A \Rightarrow^+ \checkmark$, where V is the principal type of a value (also called a value type and described by Figure 9) and A is the type of a pattern. This relation holds when values of type V match patterns of type A; the defining rules of the relation closely correspond to the rules defining the pattern matching procedure so we leave it to the appendix for brevity. To abstract over elimination pattern matching, we similarly define an elimination coverage relation $\hat{V}/A^- \Rightarrow^- \checkmark$. Here, $\hat{V}$ is the type of an elimination frame.

The matching abstraction lemmas describe the intent behind both of these definitions. That is, if a value's (resp. elimination frame's) type matches a pattern's type, then the value (resp. elimination frame) matches the pattern.

Lemma 6.4 (Pattern Matching Abstraction).

(1) *Suppose* $\cdot \vdash v \Rightarrow V$ *and* $\cdot \vdash A$. *If* $V/A \Rightarrow^{+} \checkmark$ *then* v *matches all patterns* p *for which there exists some* Δ *such that* $\cdot \vdash p \Rightarrow A \dashv \Delta$.
(2) *Suppose* $\cdot \vdash F \Rightarrow \hat{V}$ *and* $\cdot \vdash \hat{p} \Rightarrow B^{-} \hookrightarrow C \dashv \Delta$. *If* $\hat{V}/B^{-} \Rightarrow^{-} \checkmark$ *then* F *matches* $\hat{p}$.

This lemma allows us to reason about pattern matching purely at the type level; we need not worry about the particular values or patterns involved. It is an approach with a clear connection to abstract interpretation [18].

It remains to connect subtyping and the type-level dispatch operator to the coverage relations.

Lemma 6.5 (Completeness of Coverage Relations).

(1) *If* $V \leq A$ *then* $V/A \Rightarrow^{+} \checkmark$.
(2) *If there exists* C *such that* $\mathrm{disp}(A^{-}, \hat{V}) \Rightarrow C$ *then* $\hat{V}/A^{-} \Rightarrow^{-} \checkmark$.

Proof. Both are proved via induction on the premise and use some auxiliary lemmas that construct the coverage relation judgments when one involved type is splittable. The proof of the second part makes use of the first when A^{-} is an arrow type. □

Coverage now follows easily from the previous two lemmas.

Lemma 6.6 (Coverage).

(1) *Suppose* $\cdot \vdash A$. *If* $\Gamma \vdash v \Leftarrow A$ *and* $\Gamma \vdash p \Rightarrow A \dashv \Delta$ *then* v *matches* p.
(2) *If* $\cdot \vdash F \Rightarrow \hat{V}$ *and* $\mathrm{disp}(A^{-}, \hat{V}) \Rightarrow B_1$ *and* $\cdot \vdash \hat{p} \Rightarrow A^{-} \hookrightarrow B_2 \dashv \Delta$ *then* F *matches* $\hat{p}$.

The desired progress lemma is a consequence of coverage.

Lemma 6.7 (Progress). *If* $\cdot \vdash e \Rightarrow A$ *then* e *is a value or there exists* e' *such that* $e \mapsto e'$.

6.2 Preservation

We face a rather subtle barrier to proving preservation. Consider the well-typed application of an overloaded function to an argument:

$$e = \lambda\{(x : \mathbf{Int}) \hookrightarrow \mathbf{Int} \mapsto e_1, (x : \mathbf{Bool}) \hookrightarrow \mathbf{Bool} \mapsto e_2\}\ v$$

Assuming $\cdot \vdash v \Rightarrow A$ and $x : \mathbf{Int} \vdash e_1 \Leftarrow \mathbf{Int}$ and $x : \mathbf{Bool} \vdash e_2 \Leftarrow \mathbf{Bool}$, we find that e has a type B such that $\mathrm{disp}((\mathbf{Int} \to \mathbf{Int}) \sqcap (\mathbf{Bool} \to \mathbf{Bool}), A) \Rightarrow B$. The expression e reduces to the expression e' below:

$$e' = \mathrm{disp}(\{\hat{p}_1 \mapsto e_1, \hat{p}_2 \mapsto e_2\}, [\cdot]\, v)$$

Establishing type preservation requires showing that the type of e' is a subtype of B. The problem arises in trying to determine what B is. The type-level dispatch operator first splits the type A. This makes things difficult since the only thing we know about A is that it is the type of an arbitrary value v. Intuitively, it seems that B should be the intersection of the output type of each overload that accepts B as an input. Unfortunately, that is not always true. Suppose $A = \mathbf{Int} \sqcup \mathbf{Bool}$. In this case, A is a subtype of neither $\mathbf{Int}$ nor $\mathbf{Bool}$. On the other hand, $B \equiv \mathbf{Int} \sqcup \mathbf{Bool}$. Thankfully, it turns out this case is impossible. There is no value whose principal type is $\mathbf{Int} \sqcup \mathbf{Bool}$; it is not a value type. To take advantage of this fact, we first need a couple properties relevant to the dispatch procedure.

Lemma 6.8 (Downward Closure of Distinguishability). *If* $A \diamond B$ *and* $B' \leq B$ *then* $A \diamond B'$.

Proof. Immediate from the distinguishability rules. □

Lemma 6.9 (Dispatch). *Suppose* $\mathrm{split_u}(B) \not\Rightarrow$ *and* $\mathrm{split_u}(B') \not\Rightarrow$. *If* $A_1 \bowtie A_2$ *and* $\mathrm{disp}(A_1, B) \Rightarrow C_1$ *and* $\mathrm{disp}(A_1, B') \not\Rightarrow$ *and* $\mathrm{disp}(A_2, B') \Rightarrow C_2'$ *then* $B \diamond B'$.

Proof. By induction on $A_1 \bowtie A_2$, using Lemma 6.8. □

From these we can prove the required properties of the disp operator.

Lemma 6.10 (Type-Level Dispatch on Value Types).
Suppose $\cdot \vdash A_1 \sqcap A_2$ *and* $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$.

(1) *If* $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$ *then* $B_1 \sqcap B_2 \leq B$.
(2) *If* $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ *and* $\mathrm{disp}(A_2, \hat{V}) \not\Rightarrow$ *then* $B_1 \leq B$.

Thanks to this lemma, we know that we can compute the output type of an application of an overloaded function to an unkown value from the output types of each individual overload. Lemma 5.1 allows the subtyping relationships in the conclusion of each of these cases to be strengthened to an equivalence.

With that difficulty taken care of, we next turn our attention to two key properties of pattern matching. The first ensures that patterns of distinguishable types are mutally exclusive while the second ensures that substitutions produced from pattern matching are well-typed.

Lemma 6.11 (Distinguishable Patterns). *Suppose* $\Gamma \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ *and* $\Gamma \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$ *where* $A_1 \diamond A_2$. *There does not exist a closed, well-typed value that matches both* p_1 *and* p_2.

Proof. By properties of the coverage relations and induction on $A_1 \diamond A_2$. □

Lemma 6.12 (Adequacy).

(1) *If* $\cdot \vdash p \Rightarrow A \dashv \Delta$ *and* $\Gamma \vdash v \Leftarrow A$ *and* $v/p \Rightarrow \sigma$ *then* $\Gamma \vdash \sigma \Leftarrow \Delta$
(2) *Suppose* $\cdot \vdash A$, *and* $\cdot \vdash \hat{p} \Rightarrow A \hookrightarrow B_1 \dashv \Delta$, *and* $\cdot; A \vdash F \Rightarrow B_2$, *and* $F/\hat{p} \Rightarrow \sigma \hookrightarrow B_3$. *Then* $\cdot; B_1 \vdash \sigma \Leftarrow \Delta \hookrightarrow B_3$ *and* $B_3 \leq B_2$.

Proof. By induction on the pattern matching rules using Lemma 6.11. In the case of value pattern matching against or-patterns, we again make use of Lemma 6.6. □

The judgement $\cdot; B_1 \vdash \sigma \Leftarrow \Delta \hookrightarrow B_3$ in the conclusion above means that each value in σ has a type given by Δ and that an elimination frame in subst has input and output types B_1 and B_3 respectively. We refer the reader to the appendix for its formal definition. With adequacy established, we can now prove the type preservation lemma.

Lemma 6.13 (Preservation). *If* $\cdot \vdash e \Leftarrow A$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' \Leftarrow A$.

Proof. By induction on $e \mapsto e'$, making use of Lemma 5.1, Lemma 6.10, and Lemma 6.12. □

6.3 Soundness and Determinism

With progress and preservation proven, our main result is now within reach.

Theorem 6.14 (Type Soundness). *If* $\cdot \vdash e \Leftarrow A$ *and* $e \mapsto^* e'$ *then either* e' *is a value or* e' *can take another step.*

This establishes that well-typed programs do not "go wrong" [33]. Furthermore, the type system also ensures the determinism of reduction.

Lemma 6.15 (Determinism of Evaluation).

(1) *If* $\Gamma \vdash p \Rightarrow A \dashv \Delta$ *and* $\Gamma \vdash v \Leftarrow A$ *and* $v/p \Rightarrow \sigma_1$ *and* $v/p \Rightarrow \sigma_2$ *then* $\sigma_1 = \sigma_2$.

(2) *If* $\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta$ *and* $\Gamma \vdash F \Rightarrow \hat{V}$ *and* $F/\hat{p} \Rightarrow \sigma_1 \hookrightarrow B_1$ *and* $F/\hat{p} \Rightarrow \sigma_2 \hookrightarrow B_2$ *then* $\sigma_1 = \sigma_2$ *and* $B_1 = B_2$*.*

(3) *Suppose* $\cdot \vdash e \Rightarrow A$*. If* $e \mapsto e_1$ *and* $e \mapsto e_2$ *then* $e_1 = e_2$*.*

Proof. The only source of nondeterminism in the definition of pattern matching (and reduction) is when p has the form $p_1|p_2$ where v matches both p_1 and p_2. By inversion on the typing derivation for $p_1|p_2$, the type of p_1 must be distinguishable from that of p_2. This contradicts Lemma 6.11. □

7 RELATED WORK

The Merge Operator and Disjoint Intersection Types. Oliveira et al. [36] proposed the λ_i calculus, which only allows intersections of disjoint types. The goal of the disjointness restriction was to address the ambiguity in Dunfield's calculus and prove the coherence of the elaboration. Various extensions to λ_i, as well as relaxations to the type system were proposed afterwards. Bi et al. [6] relaxed the disjointness restriction, requiring it only on merges and allowing the use of unrestricted intersections. To enable nested composition, they added a more powerful subtyping relation based on the well-known BCD subtyping [4] relation, which supports distributivity rules for intersections over other type constructs.

Huang et al. [29] proposed a new approach to model the type-directed semantics of calculi with a merge operator, allowing for a direct proof of determinism of the operational semantics. Runtime implicit (up)casting replaces the coercive subtyping. For example, being annotated by an intersection type $A \sqcap B$ means a value will be cast by A and B respectively and merged together. $F_{\bowtie}$ also uses a type-directed semantics, but it does not employ a casting relation. Instead, our dynamic semantics is based on η-expansion. At runtime, merges are rewritten without duplication and other annotations are discarded. In addition, unlike all previous calculi with disjoint intersection types, $F_{\bowtie}$ supports overloading and union types by using two, more refined, disjointness relations (distinguishability and mergeability). The mergeability relation in $F_{\bowtie}$ is closely related to the disjointness relation in the previous calculi. At the moment, $F_{\bowtie}$ does not yet support unrestricted intersections/unions and disjoint polymorphism, but we plan to study these extensions in the future.

Refinement Types. Intersection and union types are used in refinement type systems to increase the type-level expressiveness [19, 25]. Such systems either do not support overloading, as there is no merge, or use it only as a mechanism to annotate the term differently [20]. These type systems face a similar problem: they must calculate the return type for application, where the function can have an intersection or union type. Due to the lack of overloading, the subtyping relation can be stronger, as we discussed in §4.1, but reduction is type irrelevant and often erases annotations.

Overloading, Semantic Subtyping, and CDuce. Castagna et al. [13] studied a restricted form of the merge operator for overloading. In their calculus $\lambda\&$ only merges of functions are allowed and only one function is selected for β-reduction, so nested composition is not supported. Instead of disjointness, the criteria in $\lambda\&$ for types of merged functions is: for any argument they can take, a best-matching type always exists. This restriction forbids merging a function of $\mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$ and a function of $\mathbf{Int} \to \mathbf{Bool} \to \mathbf{Bool}$, but it allows overlapping among overloaded functions, which violates disjointness. Nonetheless adding more implementations to an overloaded function may lead to a different implementation being choosen with a best-match semantics, which could be unexpected. With disjointness this cannot happen, since adding a new overlapping implementation to an existing merge would result in a type error.

Semantic subtyping [26] is an approach where types are given a set-theoretic interpretation. Calculi with semantic subtyping are equipped with very expressive subtyping relations, containing unions, intersections, negation types and various distributivity rules. $F_{\bowtie}$ employs syntactic subtyping. While its subtyping relation is quite expressive, it does not support negation types. Calculi with semantic subtyping support expressive forms of overloading, which is typically resolved using a *best-match* semantics, as in Castagna et al.'s work. However, existing calculi with semantic subtyping do not support a merge operator or nested composition, key features for compositional programming [51]. In contrast, $F_{\bowtie}$ is designed to support compositional programming features, and nested composition in particular. It integrates features of two previously separate types of calculi: those with disjoint intersection types and those with overloading.

CDuce uses semantic subtyping. Its core calculus [49] has a type-case expression which has two branches and takes an expression, whose type, after evaluating to a value, determines which branch will be executed. Due to its special typing rule, the type case can encode overloaded functions typed by an intersection type, making it very similar to disjoint merges. But it has no disjointness constraint and only compares the value's type against a given type then behaves like a if-then-else expression. The cost is: functions in CoreCDuce are explicitly annotated. In contrast, our distinguishability relation intentionally avoids the need for comparing a function value to a function type in the runtime.

Elimination Constructs for Union Types. Unlike the sum type (or tagged unions), untagged unions are not always equipped by an explicit elimination construct. When first introduced by MacQueen et al. [32], the typing rule for union allows it to be eliminated under any context that can handle both possibilities of the union. Unfortunately, this rule breaks type preservation unless the β-reduction is performed in parallel [3]. A sound calculus with this typing rule must avoid evaluating the term of a union type multiple times after it substitutes the same variable. Two kinds of restrictions are employed in the literature: van Bakel et al. [45] only type values by unions, while Dunfield and Pfenning [23] requires the variable of union types occurs only once in the context.

For the above calculi, the same piece of code is executed no matter what runtime type the term has. But practically, many languages supporting union types choose to have a type-dependent elimination construct: Igarashi and Nagira [30] proposed a case analysis expression for Featherweight Java. XDuce has a pattern matching expression [27]. Generally speaking, the elimination constructs in those languages offer a first-match semantics, where cases can overlap and reordering the cases may change the semantics of the program. In $F_{\bowtie}$ union types are eliminated directly with applications, where the function part can, but does not have to be overloaded. Overloading functions, compared to type-case expression (typed by unions), provide more precise typing results as it can discriminate the argument type. For example, applying $(\mathbf{Int} \to \mathbf{Int}) \sqcap (\mathbf{Bool} \to \mathbf{Bool})$ to $\mathbf{Int}$ gives us $\mathbf{Int}$ while the corresponding type-case, which has type $(\mathbf{Int} \sqcup \mathbf{Bool}) \to (\mathbf{Int} \sqcup \mathbf{Bool})$, can only gives us $\mathbf{Int} \sqcup \mathbf{Bool}$. Note that the intersection type is a (proper) subtype of the function type with distributivity laws in subtyping. Based on matching styles, Rehman et al. [40]'s work is the closest to ours. They proposed a union elimination construct (a switch-case expression) based on disjointness. Their construct is inspired by the Ceylon language [31]. Their disjointness relation is closely related to our notion of distinguishability. Although their system supports intersection types, it does not have a merge operator and all values have a principal type that is not an intersection or union, which eliminates many of the technical issues at the cost of significantly less expressive power.

REFERENCES

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38.

[2] Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *Journal of logic and computation* 2, 3 (1992), 297–347.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (June 1995), 202–230.

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

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

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

[7] Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *European Symposium on Programming (ESOP)*.

[8] Richard Bird and Oege de Moor. 1996. *The Algebra of Programming*. Prentice-Hall. http://www.cs.ox.ac.uk/publications/books/algebra/

[9] Luca Cardelli and John C. Mitchell. 1990. Operations on Records. In *Proceedings of the Fifth International Conference on Mathematical Foundations of Programming Semantics* (New Orleans, Louisiana, USA). Springer-Verlag, Berlin, Heidelberg, 22–52.

[10] J. Carette, O. Kiselyov, and C. Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509–543.

[11] Giuseppe Castagna. 1997. Unifying Overloading and λ-Abstraction: $\Lambda_{\{\}}$. *Theor. Comput. Sci.* 176, 1–2 (apr 1997), 337–345.

[12] Giuseppe Castagna. 2022. Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science* Volume 16, Issue 1 (Feb. 2022).

[13] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (feb 1995), 115–135.

[14] Giuseppe Castagna, Mickaël Laurent, Kim Nguyễn, and Matthew Lutze. 2022. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (Jan 2022), 31 pages.

[15] Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science (MSCS)* 6, 5 (1996), 469–501.

[16] William R. Cook, Walter Hill, and Peter S. Canning. 1989. Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. Association for Computing Machinery, 125–135.

[17] Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Archiv. Math. Logik* 19 (Jan 1978), 139–156.

[18] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252.

[19] Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208.

[20] Jana Dunfield. 2012. Annotations for Intersection Typechecking. In *Proceedings of the Sixth Workshop on Intersection Types and Related Systems (EPTCS, Vol. 121)*, Stéphane Graham-Lengrand and Luca Paolini (Eds.). 35–47.

[21] Jana Dunfield. 2014. Elaborating Intersection and Union Types. *J. Functional Programming* 24, 2–3 (2014), 133–165.

[22] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (May 2021), 38 pages.

[23] Jana Dunfield and Frank Pfenning. 2003. Type assignment for intersections and unions in call-by-value languages. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 250–266.

[24] Erik Ernst. 2001. Family Polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*.

[25] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

[26] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sep 2008), 64 pages.

[27] Haruo Hosoya and Benjamin C Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)* 3, 2 (2003), 117–148.

[28] Xuejing Huang and Bruno C. d. S. Oliveira. 2021. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–24.

[29] Xuejing Huang, Jinxu Zhao, and Bruno C. D. S. Oliveira. 2021. Taming the Merge Operator. *Journal of Functional Programming* 31 (2021).

[30] Atsushi Igarashi and Hideshi Nagira. 2006. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*. 1435–1441.

[31] Gavin King. 2013. The Ceylon language specification, version 1.0. https://ceylon-lang.org/documentation/1.0/spec/

[32] David MacQueen, Gordon Plotkin, and Ravi Sethi. 1984. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 165–174.

[33] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (Aug. 1978), 348–375.

[34] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report. EPFL.

[35] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming (ECOOP)*.

[36] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2016-09-04) *(ICFP 2016)*. Association for Computing Machinery, 364–377.

[37] Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Proceedings of the 10th International Conference on Functional and Logic Programming* (Sendai, Japan) *(FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 56–71.

[38] Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism*. Ph. D. Dissertation. CMU-CS-91-205, Carnegie Mellon University.

[39] Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. Academic Press.

[40] Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. 2022. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.).

[41] John C. Reynolds. 1988. *Preliminary design of the programming language Forsythe*. Technical Report. CMU-CS-88-159, Carnegie Mellon University.

[42] John C. Reynolds. 1997. *Design of the Programming Language FORSYTHE*. Birkhauser Boston Inc., USA, 173–233.

[43] Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, 624–641.

[44] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop (Scheme)*. ACM, 81–92.

[45] Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohama. 2000. *The minimal relevant logic and the call-by-value lambda calculus*. Technical Report. TR-ARP-05-2000, The Australian National University.

[46] Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998). https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[47] Mitchell Wand. 1989. Type Inference for Record Concatenation and Multiple Inheritance. In *Symposium on Logic in Computer Science (LICS)*.

[48] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov 1994), 38–94.

[49] Zhiwu Xu. 2013. *Parametric Polymorphism for XML Processing Languages*. Ph. D. Dissertation. Université Paris-Diderot-Paris VII.

[50] Noam Zeilberger. 2008. Focusing and Higher-Order Abstract Syntax. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 359–369.

[51] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems* (April 2021).

A PRELIMINARY DEFINITIONS

$\boxed{\Gamma \vdash A}$ *(type well-formedness)*

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash c\, A} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma, \alpha \vdash B}{\Gamma \vdash \forall \alpha . B} \qquad \frac{}{\Gamma \vdash \top} \qquad \frac{}{\Gamma \vdash \bot}$$

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad A_1 \bowtie A_2}{\Gamma \vdash A_1 \sqcap A_2} \qquad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad A_1 \mathrel{\diamond} A_2}{\Gamma \vdash A_1 \sqcup A_2}$$

$\boxed{\Gamma \vdash M \Rightarrow A}$ *(λ body typing)*

$$\frac{\begin{array}{c}\Gamma; \hat{p}_1 \vdash e_1 \Rightarrow A_1, \ \ldots, \Gamma; \hat{p}_n \vdash e_n \Rightarrow A_n \\ A_1 \bowtie \ \ldots \ \bowtie A_n\end{array}}{\Gamma \vdash \{\hat{p}_1 \mapsto e_1, \ \ldots, \hat{p}_n \mapsto e_n\} \Rightarrow A_1 \sqcap \ldots \sqcap A_n}$$

$\boxed{\Gamma \vdash \sigma \Leftarrow \Delta}$ *(substitution type checking)*

$$\frac{\Gamma; A \vdash \sigma \Leftarrow \Delta \hookrightarrow \sigma(A)}{\Gamma \vdash \sigma \Leftarrow \Delta}$$

$\boxed{\Gamma; A \vdash \sigma \Leftarrow \Delta \hookrightarrow B}$ *(substitution type checking)*

$$\frac{}{\Gamma; A \vdash \cdot \Leftarrow \cdot \hookrightarrow A} \qquad \frac{\begin{array}{c}\Gamma; A \vdash \sigma \Leftarrow \Delta \hookrightarrow B \\ \Gamma \vdash v \ \Leftarrow \ \sigma(C)\end{array}}{\Gamma; A \vdash \sigma[x \mapsto v] \Leftarrow \Delta, x : C \hookrightarrow B} \qquad \frac{\begin{array}{c}\Gamma; A \vdash \sigma \Leftarrow \Delta \hookrightarrow B \\ \Gamma \vdash C \qquad B[C/\alpha] \leq B'\end{array}}{\Gamma; A \vdash \sigma[\alpha \mapsto C] \Leftarrow \Delta, \alpha \hookrightarrow B'}$$

$$\frac{\begin{array}{c}\Gamma; A \vdash \sigma \Leftarrow \Delta \hookrightarrow B \\ \Gamma; B \vdash F \ \Leftarrow \ C\end{array}}{\Gamma; A \vdash \sigma[\, * \ \mapsto F] \Leftarrow \Delta \hookrightarrow C}$$

$\boxed{\Delta_1 \sqcup \Delta_2 \text{ and } \Delta_1 \sqcap \Delta_2}$

$$x : A \in (\Delta_1 \sqcup \Delta_2) \quad \text{iff} \quad x : A_1 \in \Delta_1 \text{ and } x : A_2 \in \Delta_2 \text{ and } A = A_1 \sqcup A_2$$

$$\begin{array}{rcl} x : A_1 \in (\Delta_1 \sqcap \Delta_2) & \text{if} & x : A_1 \in \Delta_1 \text{ and } x \notin \mathrm{dom}\,(\Delta_2) \\ x : A_2 \in (\Delta_1 \sqcap \Delta_2) & \text{if} & x : A_2 \in \Delta_2 \text{ and } x \notin \mathrm{dom}\,(\Delta_1) \\ x : A_1 \sqcap A_2 \in (\Delta_1 \sqcap \Delta_2) & \text{if} & x : A_1 \in \Delta_1 \text{ and } x : A_2 \in \Delta_2 \end{array}$$

$\boxed{A \mathrel{\diamond}_{\mathrm{alg}} B}$ *(algorithmic distinguishability)*

$$\frac{A \mathrel{\diamond}_{\mathrm{ax}} B}{A \mathrel{\diamond}_{\mathrm{alg}} B} \qquad \frac{A \mathrel{\diamond}_{\mathrm{alg}} B}{c\, A \mathrel{\diamond}_{\mathrm{alg}} c\, B} \qquad \frac{\begin{array}{c}\mathrm{split}_{\mathrm{u}}(A) \Rightarrow A_1 \sqcup A_2 \\ A_1 \mathrel{\diamond}_{\mathrm{alg}} B \qquad A_2 \mathrel{\diamond}_{\mathrm{alg}} B\end{array}}{A \mathrel{\diamond}_{\mathrm{alg}} B} \qquad \frac{\begin{array}{c}\mathrm{split}_{\mathrm{u}}(A) \Rightarrow A_1 \sqcup A_2 \\ B \mathrel{\diamond}_{\mathrm{alg}} A_1 \qquad B \mathrel{\diamond}_{\mathrm{alg}} A_2\end{array}}{B \mathrel{\diamond}_{\mathrm{alg}} A}$$

$$\frac{A \mathrel{\diamond}_{\mathrm{alg}} B}{A \sqcap A' \mathrel{\diamond}_{\mathrm{alg}} B} \qquad \frac{A' \mathrel{\diamond}_{\mathrm{alg}} B}{A \sqcap A' \mathrel{\diamond}_{\mathrm{alg}} B} \qquad \frac{B \mathrel{\diamond}_{\mathrm{alg}} A}{B \mathrel{\diamond}_{\mathrm{alg}} A \sqcap A'} \qquad \frac{B \mathrel{\diamond}_{\mathrm{alg}} A'}{B \mathrel{\diamond}_{\mathrm{alg}} A \sqcap A'}$$

$$\boxed{V\,/\,A \Rightarrow^{+} \checkmark} \qquad \textit{(value coverage)}$$

$$\frac{V\,/\,A \Rightarrow^{+} \checkmark}{c\ V\,/\,c\ A \Rightarrow^{+} \checkmark} \qquad \frac{V\,/\,A \Rightarrow^{+} \checkmark}{V\,/\,A \sqcup B \Rightarrow^{+} \checkmark} \qquad \frac{V\,/\,B \Rightarrow^{+} \checkmark}{V\,/\,A \sqcup B \Rightarrow^{+} \checkmark} \qquad \frac{V\,/\,A \Rightarrow^{+} \checkmark \qquad V\,/\,B \Rightarrow^{+} \checkmark}{V\,/\,A \sqcap B \Rightarrow^{+} \checkmark}$$

$$\frac{}{V\,/\,A^{-} \Rightarrow^{+} \checkmark}$$

$$\boxed{\hat{V}\,/\,A^{-} \Rightarrow^{-} \checkmark} \qquad \textit{(elimination coverage)}$$

$$\frac{V\,/\,A \Rightarrow^{+} \checkmark}{V\,/\,A \to B \Rightarrow^{-} \checkmark} \qquad \frac{}{[B]\,/\,\forall\,\alpha.A \Rightarrow^{-} \checkmark} \qquad \frac{\begin{array}{c}\hat{V}\,/\,A_1^{-} \Rightarrow^{-} \checkmark \\ \hat{V}\,/\,A_2^{-} \Rightarrow^{-} \checkmark\end{array}}{\hat{V}\,/\,A_1^{-} \sqcup A_2^{-} \Rightarrow^{-} \checkmark} \qquad \frac{\hat{V}\,/\,A_1^{-} \Rightarrow^{-} \checkmark}{\hat{V}\,/\,A_1^{-} \sqcap A_2^{-} \Rightarrow^{-} \checkmark}$$

$$\frac{\hat{V}\,/\,A_2^{-} \Rightarrow^{-} \checkmark}{\hat{V}\,/\,A_1^{-} \sqcap A_2^{-} \Rightarrow^{-} \checkmark}$$

B TYPING ALGORITHMS

Lemma B.1. *If* $A \mathrel{\diamond}_{\text{alg}} B$ *then* $A \mathrel{\diamond} B$.

Proof. Coq proof in `Distinguishability.v` (distinguishability_sound). □

Lemma B.2. *If* $A \mathrel{\diamond} B$ *then* $A \mathrel{\diamond}_{\text{alg}} B$.

Proof. Coq proof in `Distinguishability.v` (distinguishability_complete). □

Lemma B.3. *If* $A \leq B$ *then* $A \leq_{\text{alg}} B$.

Proof. Coq proof in `DistSubtyping.v` (dsub2asub). □

Lemma B.4. *If* $A \leq_{\text{alg}} B$ *then* $A \leq B$.

Proof. Coq proof in `DistSubtyping.v` (dsub2asub). □

C TYPE SOUNDNESS

C.1 Properties of Subtyping and Related Operators

Lemma C.1 (Completeness of Coverage Relations).

(1) *If* $V \leq A$ *then* $V\,/\,A \Rightarrow^{+} \checkmark$.
(2) *If there exists* C *such that* $\mathrm{disp}(A^{-}, \hat{V}) \Rightarrow C$ *then* $\hat{V}\,/\,A^{-} \Rightarrow^{-} \checkmark$.

Proof. Coq proof in `SimpleSub.v` (`[1] sub2psub; [2] apply2nsub`). Proven with the help of the following two lemmas.
If $\mathrm{split}_{\mathrm{u}}(V) \Rightarrow V_1 \sqcup V_2$ and $V_1\,/\,A \Rightarrow^{+} \checkmark$ (or $V_2\,/\,A \Rightarrow^{+} \checkmark$) then $V\,/\,A \Rightarrow^{+} \checkmark$. (Coq proof in `SimpleSub.v` (`psub_unionL and psub_unionR`).)
If $\mathrm{split}_{\mathrm{i}}(A) \Rightarrow A_1 \sqcap A_2$ and $V\,/\,A_1 \Rightarrow^{+} \checkmark$ and $V\,/\,A_2 \Rightarrow^{+} \checkmark$ then $V\,/\,A \Rightarrow^{+} \checkmark$. (Coq proof in `SimpleSub.v` (psub_merge_intersection).)
If $\mathrm{split}_{\mathrm{u}}(A) \Rightarrow A_1 \sqcup A_2$ and $V\,/\,A_1 \Rightarrow^{+} \checkmark$ (or $V\,/\,A_2 \Rightarrow^{+} \checkmark$) then $V\,/\,A \Rightarrow^{+} \checkmark$. (Coq proof in `SimpleSub.v` (`psub_splu_left and psub_splu_right`).)
If $\mathrm{split}_{\mathrm{u}}(A^{-}) \Rightarrow A_1^{-} \sqcup A_2^{-}$ and $\hat{V}\,/\,A_1^{-} \Rightarrow^{-} \checkmark$ (or $\hat{V}\,/\,A_2^{-} \Rightarrow^{-} \checkmark$) then $\hat{V}\,/\,A^{-} \Rightarrow^{-} \checkmark$. (Coq proof in `SimpleSub.v` (`nsub_unionL and nsub_unionR`).) □

Lemma C.2. *If* $A \leq \bot$ *then* $V\,/\,A \not\Rightarrow^{+}$.

Proof. Coq proof in `SimpleSub.v` (`psub_sub_bot_inv`). Proven in Coq with the help of the following two inversion lemmas.
If $\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2$ and $V/A \Rightarrow^+ \checkmark$ then $V/A_1 \Rightarrow^+ \checkmark$ or $V/A_2 \Rightarrow^+ \checkmark$.
If $\mathrm{split_i}(A) \Rightarrow A_1 \sqcap A_2$ and $V/A \Rightarrow^+ \checkmark$ then $V/A_1 \Rightarrow^+ \checkmark$ and $V/A_2 \Rightarrow^+ \checkmark$. □

$\boxed{A \sim B}$ *(Similar Types)*

$$\frac{}{A^- \sim B^-} \qquad \frac{V_1 \sim V_2}{l\ V_1 \sim l\ V_2}$$

Fig. 10. Similarity Relation on Types

Lemma C.3 (Properties of the Union-Split Results of a Value Type (Similar Types)).

(1) *If* $\mathrm{split_u}(V) \Rightarrow A \sqcup B$ *then* $A \sim B$.
(2) *If* $A \sim B$ *then not* $A \diamond B$.
(3) *If* $A \sim B$ *and* $A \leq A'$ *and* $B \leq B'$ *then not* $A' \diamond B'$.
(4) *If* $\cdot \vdash A_1 \sqcap A_2$ *and* $B_1 \sim B_2$*, it is impossible that* $\mathrm{disp}(A_1, B_2) \not\Rightarrow$ *and* $\mathrm{disp}(A_2, B_1) \not\Rightarrow$ *knowing* $\mathrm{disp}(A_1, B_1) \Rightarrow C$ *and* $\mathrm{disp}(A_2, B_2) \Rightarrow C'$.

Proof. (1) Coq proof in `SimpleSub.v` (`sim2similar`). The lemma can be proved by induction on the value type V. Note that the splitting results of a negative type are always two negative types.
(2) Coq proof in `Distinguishability.v` (`sim_no_distinguishability`). Induction on the similarity judgment.
(3) Coq proof in `Dispatch.v` (`sub_sim_distinguishability_sub_inv`). Proved via the last lemma and the downward closure of distinguishability (Lemma C.11).
(4) Coq proof in `Dispatch.v` (`applyty_and_sim_inv`). Proved via the second last lemma and Lemma C.12. □

Lemma C.4 (Inversion of Subtyping on (Co-)Value types).

(1) *If* $V \leq l\ A$ *then there exists* B *such that* $V = l\ B$.
(2) *If* $V \leq A \rightarrow B$ *then* V *is negative.*
(3) *If* $V \leq \forall\, \alpha.A$ *then* V *is negative.*
(4) *There is no* V *such that* $V \leq \bot$.
(5) *If* $\mathrm{disp}(A \rightarrow A', \hat{V}) \Rightarrow B$ *then* $\hat{V}$ *is a value type.*
(6) *If* $\mathrm{disp}(\forall\, \alpha.A, \hat{V}) \Rightarrow B$ *then* $\hat{V} = [C]$ *for some* C.
(7) $\mathrm{disp}(\top, \hat{V}) \not\Rightarrow$.
(8) $\mathrm{disp}(l\ V, \hat{V}) \not\Rightarrow$.

Proof.

(1) By Lemma C.1, $V/l\ A \Rightarrow^+ \checkmark$. Applying an inversion lemma, we have B such that $V = l\ B$. Proven in Coq.

(1-4) Coq proof in `SimpleSub.v` (`[1] valtyp_sub_rcd_inv; [2] valtyp_sub_arrow_inv; [3] valtyp_sub_forall_inv; [4] valtyp_bot_inv`).
(5-8) Coq proof in `ApplyTy.v` (`[5] applyty_arrow; [6] applyty_forall; [7] apply_top_false_1. apply_top_false_2; [8] apply_box_false_1, apply_box_false_2`). □

Lemma C.5 (Inversion of Subtyping).

(1) *If* $\{l : A\} \leq \{l : B\}$ *then* $A \leq B$.
(2) *If* $A \leq B_1 \sqcap B_2$ *then* $A \leq B_1$ *and* $A \leq B_2$.

Proof. Coq proof in `DistSubtyping.v` (`[1] algo_sub_rcd_inv; [2] algo_sub_and_inv`). ☐

Lemma C.6 (Inversion of Disjoint Union Subtyping). *If* $V \leq A_1 \sqcup A_2$ *and* $A_1 \diamond A_2$ *then* $V \leq A_1$ *or* $V \leq A_2$.

Proof. Coq proof in `Dispatch.v` (`sub_inv_distinguishable_union`). It is proved by induction on the sum of sizes of V and $A_1 \sqcup A_2$. When V is union-ordinary, it can be directly proved via an inversion lemma on subtyping. When $\mathrm{split_u}(V) \Rightarrow B_1 \sqcup B_2$, we know $B_1 \leq A_i$ and $B_2 \leq A_j$ via the inductive hypothesis. When i equals to j, the goal can be proved easily via subtyping properties.

Now we show it is impossible for i and j to be different. Firstly we use the inductively defined relation (Figure 10) to describe the similarity of B_1 and B_2: either they are both negative types, or they are record types with the same label and their inner types still share the similarity. From Lemma C.3 we know that B_1 and B_2 cannot have two supertypes respectively that are distinguishable, like A_1 and A_2. ☐

Lemma C.7 (Inversion of Type-Level Dispatch).

(1) *If* $\mathrm{disp}(\bot, \hat{A}) \Rightarrow B$ *then* $B \equiv \bot$.
(2) *If* $\mathrm{disp}(A \to B, \hat{A}) \Rightarrow B'$ *then there exists* $A' = \hat{A}$ *such that* $A' \leq A$ *and* $B' \equiv B$.
(3) *If* $\mathrm{disp}(\forall \alpha.B, \hat{A}) \Rightarrow B'$ *then there exists* $A = \hat{A}$ *such that* $B' \equiv B[A/\alpha]$.
(4) *If* $\mathrm{disp}(A, \hat{A}) \Rightarrow B$ *and* $\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2$ *then* $\mathrm{disp}(A_1, \hat{A}) \Rightarrow B_1$ *for some* B_1 *and* $\mathrm{disp}(A_2, \hat{A}) \Rightarrow B_2$ *for some* B_2 *such that* $B \equiv B_1 \sqcup B_2$.
(5) *If* $\mathrm{disp}(A_1 \sqcap A_2, \hat{A}) \not\Rightarrow$ *then* $\mathrm{disp}(A_1, \hat{A}) \not\Rightarrow$ *and* $\mathrm{disp}(A_2, \hat{A}) \not\Rightarrow$.
(6) $\mathrm{disp}(\top, \hat{A}) \not\Rightarrow$
(7) *If* $\mathrm{disp}(A, \hat{A}) \not\Rightarrow$ *and* $\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2$ *then* $\mathrm{disp}(A_1, \hat{A}) \not\Rightarrow$ *or* $\mathrm{disp}(A_2, \hat{A}) \not\Rightarrow$.

Proof. Coq proof in `ApplyTy.v` (`[1] applyty_bot; [2] applyty_arrow_sound_2; [3] applyty_forall_sound_1; [4] applyty_splitu_inv; [5] napplyty_spliti_inv; [6] apply_top_false; [7] napplyty_splitu_inv`). ☐

Lemma C.8 (Soundness of Type-Level Dispatch).

(1) *If* $\mathrm{disp}(A, B) \Rightarrow C$ *then* $A \leq B \to C$.
(2) *If* $\mathrm{disp}(A, [B]) \Rightarrow C$ *then there exists* A' *such that* $A \leq \forall \alpha.A'$ *and* $C \leq A'[B/\alpha]$.

Proof. Coq proof in `ApplyTy.v` (`[1] applyty_soundness_1; [2] applyty_soundness_2`). ☐

Lemma C.9.

(1) *If* $\mathrm{split_u}(A) \Rightarrow A_1 \sqcup A_2$ *and* $\mathrm{disp}(A_1, \hat{A}) \not\Rightarrow$ *or* $\mathrm{disp}(A_2, \hat{A}) \not\Rightarrow$ *then* $\mathrm{disp}(A, \hat{A}) \not\Rightarrow$.

Proof. Coq proof in `ApplyTy.v` (`napplyty_splitu_fun`). ☐

Lemma C.10 (Monotonicity of Type-Level Dispatch).

(1) *If* $\mathrm{disp}(A, \hat{B}) \Rightarrow C$ *and* $A' \leq A$ *then there exists* $C' \leq C$ *such that* $\mathrm{disp}(A', \hat{B}) \Rightarrow C'$.
(2) *If* $\mathrm{disp}(A, B) \Rightarrow C$ *and* $B' \leq B$ *then there exists* $C' \leq C$ *such that* $\mathrm{disp}(A, B') \Rightarrow C'$.

Proof. Coq proof in `ApplyTy.v` (`[1] monotonicity_applyty_1; [2] monotonicity_applyty_2_1`). Induction on the sum of sizes of A, A' and $\hat{B}$, or on the sum of sizes of A, B, and B'. Then we analyze the form of types to decide how to construct the type-level dispatch in the goal. For each cases, we use the soundness (Lemma C.8) and completeness (Lemma C.32) property to convert the premise to the goal. For the polymorphic instantiation case, Lemma C.30 is also important. □

C.2 Dispatch

Lemma C.11 (Downward Closure of Distinguishability). *If $A \nLeftrightarrow B$ and $B' \leq B$ then $A \nLeftrightarrow B'$.*

Proof. Coq proof in `Distinguishability.v` (`distinguishability_downward`). Induction on the sum of the size of the three types. The basic cases is when all of them are ordinary, as a splittable type has a bigger size than any of its splitting components. For the inductive cases, we want to always be able to invert the premises. We consider three cases: both A and B are union-ordinary; A is union-ordinary while B is union-splittable; A is union-splittable. The ordinary conditions helps us invert the distinguishability premises. In the first case, we analyse every constructors of $A \nLeftrightarrow B$. □

Lemma C.12 (Dispatch). *Suppose* $\mathrm{split_u}(B) \nRightarrow$ *and* $\mathrm{split_u}(B') \nRightarrow$. *If $A_1 \bowtie A_2$ and* $\mathrm{disp}(A_1, B) \Rightarrow C_1$ *and* $\mathrm{disp}(A_1, B') \nRightarrow$ *and* $\mathrm{disp}(A_2, B') \Rightarrow C_2'$ *then $B \nLeftrightarrow B'$.*

Proof. Coq proof in `Dispatch.v` (`dispatch`). By induction on $A_1 \bowtie A_2$. Note we only discuss function application but not polymorphic instantiation here. So no universal type are involved. In the first two cases, we know A_1 and A_2 are both function types. From Lemma C.7(2) we can obtain the subtyping relation between B and B' and the argument type of A_1 and A_2 respectively. For the first case where the two argument types are distinguishable, we can prove the goal via Lemma C.11. For the second case, the two argument types are the same, which leads to contradiction as we know B' is a subtype of the argument type from $\mathrm{disp}(A_2, B') \Rightarrow C_2'$ but we have $\mathrm{disp}(A_1, B') \nRightarrow$ (Coq proof in `ApplyTy.v` (`napplyty_sub_inv`).). The third case about universal types is impossible here. The axiomatic cases are also impossible because the top type cannot be dispatched (Lemma C.7(6)). For other cases, we use the inversion lemmas (Lemma C.7(4)(5)(7)) to destruct the premises and then apply the inductive hypothesis. □

Lemma C.13. *If $V \nLeftrightarrow V'$ then $V'/V \nRightarrow^+$.*

Proof. Coq proof in `Distinguishability.v` (`distinguishability_valtyp_not_psub`). □

Lemma C.14. *Consider a closed and well-formed type A. If* $\mathrm{disp}(A, V) \Rightarrow C$ *and* $\mathrm{disp}(A, V') \Rightarrow C'$ *where $V'/V \Rightarrow^+ \checkmark$ then $C \leq C'$.*

Proof. Coq proof in `Dispatch.v` (`applyty_valtyp_psub`). Induction on the sum of sizes of A, V and V'. First consider the case where V and V' are both union-ordinary. We destruct the type A by inversion on the type-level dispatch judgment and the well-formedness of A. In some of the cases, we need to combine Lemma C.12 and Lemma C.13 to show the contradiction. When either of V and V' is union-splittable,

□

Lemma C.15 (Dispatch on Value Types).
(1) *If $\cdot \vdash A_1 \sqcap A_2$ and* $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$ *and* $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ *and* $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$ *then $B_1 \sqcap B_2 \leq B$.*
(2) *If $\cdot \vdash A_1 \sqcap A_2$ and* $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$ *and* $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ *and* $\mathrm{disp}(A_2, \hat{V}) \nRightarrow$ *then $B_1 \leq B$.*
(3) *If $\cdot \vdash A_1 \sqcap A_2$ and* $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$ *and* $\mathrm{disp}(A_1, \hat{V}) \nRightarrow$ *and* $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$ *then $B_2 \leq B$.*

(4) *If* $\mathrm{disp}(A_1 \sqcup A_2, \hat{V}) \Rightarrow B$ *and* $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ *and* $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$ *then* $B \leq B_1 \sqcup B_2$.

Proof. Coq proof in `Dispatch.v` (`[1] applyty_andl_sub_1; [2] applyty_andl_sub_2; [3] applyty_andl_sub_3; [4] applyty_orl_sub`). The proof of all these properties relies on the determinism of type-level dispatch and the inversion lemma Lemma C.7(4). Lemma (4) is a direct corollary of them. The other three lemmas requires analyzing whether $\hat{V}$ is union-ordinary or union-splittable. For the splittable cases, the proof makes use of the similarity relation Lemma C.3. □

Lemma C.16 (Inversion of Dispatch on Value Types). *If* $\cdot \vdash A_1 \sqcap A_2$ *and* $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$ *then there exists a* B' *such that either* $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B'$ *or* $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B'$.

Proof. Coq proof in `Dispatch.v` (`applyty_valtyp_inter_inv and applyty_inter_inv`). When the dispatching is an instantiation, the goal can be directly prove. When it is a function application, we do induction on the size of $\hat{V}$. The key case is when it is an union-splittable, like the proof of Figure C.6, we rule out some problematic possibility via the property of splittable value types (Lemma C.3(4)). □

Lemma C.17.

(1) *If* $\cdot \vdash A_1 \sqcap A_2$ *and* $\cdot; A_1 \sqcap A_2 \vdash F \Rightarrow B$ *and* $\cdot; A_1 \vdash F \Rightarrow B_1$ *and* $\cdot; A_2 \vdash F \Rightarrow B_2$ *then* $B \equiv B_1 \sqcap B_2$.

(2) *If* $\cdot \vdash A_1 \sqcap A_2$ *and* $\cdot; A_1 \sqcap A_2 \vdash F \Rightarrow B$ *then either* $\cdot; A_1 \vdash F \Rightarrow B_1$ *and* $B_1 \equiv B$ *or* $\cdot; A_2 \vdash F \Rightarrow B_2$ *and* $B_2 \equiv B$.

(3) *If* $\cdot; A_1 \sqcup A_2 \vdash F \Rightarrow B$ *and* $\cdot; A_1 \vdash F \Rightarrow B_1$ *and* $\cdot; A_2 \vdash F \Rightarrow B_2$ *then* $B \equiv B_1 \sqcup B_2$.

Proof.

(1) From our assumptions we have:
- $\cdot \vdash F \Rightarrow \hat{V}$
- $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$
- $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$
- $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$

$B_1 \sqcap B_2 \leq B$ is a direct consequence of Lemma C.15. Applying the monotonicity of the dispatch operator (Lemma C.10), we have $B \leq B_1$ and $B \leq B_2$. It follows $B \leq B_1 \sqcap B_2$. Thus, $B \equiv B_1 \sqcap B_2$.

(2) Without loss of generality, from our assumptions we have:
- $\cdot \vdash F \Rightarrow \hat{V}$
- $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$

If $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ and $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$ both hold then the previous case applies. At least one must hold since $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$. Thus, without loss of generality we assume $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ and $\mathrm{disp}(A_2, \hat{V}) \not\Rightarrow$.

$B_1 \leq B$ is a direct consequence of Lemma C.15. Applying the monotonicity of the dispatch operator (Lemma C.10), we have $B \leq B_1$. Thus, $B \equiv B_1$.

(3) From our assumptions we have:
- $\cdot \vdash F \Rightarrow \hat{V}$
- $\mathrm{disp}(A_1 \sqcup A_2, \hat{V}) \Rightarrow B$
- $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$
- $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$

$B \leq B_1 \sqcup B_2$ is a direct consequence of Lemma C.15. Applying the monotonicity of the dispatch operator (Lemma C.10), we have $B_1 \leq B$ and $B_2 \leq B$. It follows $B_1 \sqcup B_2 \leq B$. Thus, $B \equiv B_1 \sqcup B_2$.

□

C.3 Properties of the Typing Relation

LEMMA C.18 (REGULARITY). *Suppose for each $x : A$ in Γ, we have $\Gamma \vdash A$.*

1. *If $\Gamma \vdash M \Rightarrow A$ then $\Gamma \vdash A$.*
2. *If $\Gamma; \hat{p} \vdash e \Rightarrow A$ then $\Gamma \vdash A$.*
3. *If $\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta$ then $\Gamma \vdash A$.*
4. *If $\Gamma \vdash p \Rightarrow A \dashv \Delta$ then $\Gamma \vdash A$.*

PROOF. Induction on the typing derivation in each case. Note that the type system requires each type annotation to be well-formed and every union and intersection to be disjoint. □

LEMMA C.19 (CANONICAL FORMS).

1. *If $\cdot \vdash v \Leftarrow l\ A$ then there exists v' such that $v = l\ v'$.*
2. *If $\cdot \vdash v \Leftarrow A \rightarrow B$ then there exists M such that $v = \lambda M$.*
3. *If $\cdot \vdash v \Leftarrow \forall \alpha.A$ then there exists M such that $v = \lambda M$.*
4. *There is no v such that $\cdot \vdash v \Leftarrow \bot$.*
5. *If $\cdot; A \rightarrow A' \vdash F \Rightarrow B$ then there exists v such that $F = [\cdot]\, v$.*
6. *If $\cdot; \forall \alpha.A \vdash F \Rightarrow B$ then there exists C such that $F = [\cdot][C]$*
7. *There are no F and B such that $\cdot; \top \vdash F \Rightarrow B$.*
8. *If $\cdot \vdash v \Rightarrow V$ and $\mathrm{disp}(V, \hat{V}) \Rightarrow B$ then $v = \lambda M$.*

PROOF.

1. We have some V such that $\Gamma \vdash v \Rightarrow V$ and $V \leq l\ A$. By inversion of the subtyping relation Lemma C.4, $V = l\ V'$. By case analysis on the typing relation, there must exist v' such that $v = l\ v'$.
2. We have some V such that $\Gamma \vdash v \Rightarrow V$ and $V \leq A \rightarrow B$. By inversion of the subtyping relation Lemma C.4, V is negative. By case analysis on the typing relation, there must exist M such that $v = \lambda M$.
3. We have some V such that $\Gamma \vdash v \Rightarrow V$ and $V \leq \forall \alpha.A$. By inversion of the subtyping relation Lemma C.4, V is negative. By case analysis on the typing relation, there must exist M such that $v = \lambda M$.
4. We must have some V such that $\Gamma \vdash v \Rightarrow V$ and $V \leq \bot$, but according to Lemma C.4 there is no such V, so we have a contradiction.
5. We have some $\hat{V}$ such that $\Gamma \vdash F \Rightarrow \hat{V}$ and $\mathrm{disp}(A \rightarrow A', \hat{V}) \Rightarrow B$. By Lemma C.4, $\hat{V}$ is equal to a value type V. It follows $F = [\cdot]\, v$ for some v.
6. We have some $\hat{V}$ such that $\Gamma \vdash F \Rightarrow \hat{V}$ and $\mathrm{disp}(\forall \alpha.A, \hat{V}) \Rightarrow B$. By Lemma C.4, $\hat{V}$ is equal to $[C]$ for some C. It follows $F = [\cdot][C]$.
7. Suppose for contradiction $\Gamma; \top \vdash F \Rightarrow B$. We then have some $\hat{V}$ such that $\Gamma \vdash F \Rightarrow \hat{V}$ and $\mathrm{disp}(\top, \hat{V}) \Rightarrow B$. This latter fact contradicts Lemma C.4.
8. Proceed by case analysis on $\Gamma \vdash v \Rightarrow V$.
   In the first case, we have $\Gamma \vdash l\ v' \Rightarrow l\ V'$. From Lemma C.4 we have $\mathrm{disp}(l\ V', \hat{V}) \not\Rightarrow$. This contradicts our assumption.
   In the second case, we have $\Gamma \vdash \lambda M \Rightarrow A^-$. Immediately, $v = \lambda M$.

□

LEMMA C.20 (INVERSION OF TYPING).

1. *If $\Gamma \vdash l\ v \Leftarrow l\ A$ then $\Gamma \vdash v \Leftarrow A$.*
2. *If $\Gamma \vdash v \Leftarrow A \sqcap B$ then $\Gamma \vdash v \Leftarrow A$ and $\Gamma \vdash v \Leftarrow B$.*

(3) *If* $\cdot; A \to B \vdash [\cdot]v \Rightarrow B'$ *then* $\cdot \vdash v \Leftarrow A$ *and* $B \equiv B'$.
(4) *If* $\cdot; \forall\alpha.B \vdash [\cdot][A] \Rightarrow B'$ *then* $B' \equiv B[A/\alpha]$.
(5) *If* $\Gamma; A' \sqcup A'' \vdash F \Rightarrow B$ *then* $\Gamma; A' \vdash F \Leftarrow B$ *and* $\Gamma; A'' \vdash F \Leftarrow B$.
(6) *If* $\Gamma; \bot \vdash F \Rightarrow B$ *then* $B \equiv \bot$.
(7) *If* $\cdot \vdash F[v] \Leftarrow A$ *then there exists* V *such that* $\cdot \vdash v \Rightarrow V$ *and* $\cdot; V \vdash F \Leftarrow A$.

Proof.

(1) Inverting $\Gamma \vdash l\ v \Leftarrow l\ A$, we have $\Gamma \vdash l\ v \Rightarrow B$ such that $B \leq l\ A$. By further inversion, $\Gamma \vdash v \Rightarrow A'$ where $B = l\ A'$. Applying the inversion of subtyping lemma (Lemma C.5), $l\ A' \leq l\ A$ implies $A' \leq A$. It follows $\Gamma \vdash v \Leftarrow A$.
(2) Inverting $\Gamma \vdash v \Leftarrow A \sqcap B$, we have $\Gamma \vdash v \Rightarrow C$ such that $C \leq A \sqcap B$. By inversion of subtyping, $C \leq A$ and $C \leq B$. It follows $\Gamma \vdash v \Leftarrow A$ and $\Gamma \vdash v \Leftarrow B$.
(3) Inverting $\cdot; A \to B \vdash [\cdot]v \Rightarrow B'$ we have $\cdot \vdash [\cdot]v \Rightarrow V$ where $\cdot \vdash v \Rightarrow V$ and $\mathrm{disp}(A \to B, V) \Rightarrow B'$. By Lemma C.7, $V \leq A$ and $B \equiv B'$. Thus, $\cdot \vdash v \Leftarrow A$.
(4) Inverting $\cdot; \forall\alpha.B \vdash [\cdot][A] \Rightarrow B'$ we have $\cdot \vdash [\cdot][A] \Rightarrow [A]$ where $\cdot \vdash A$ and $\mathrm{disp}(\forall\alpha.B, [A]) \Rightarrow B'$. By Lemma C.7, $B' = B[A/\alpha]$.
(5) Inverting $\Gamma; A' \sqcup A'' \vdash F \Rightarrow B$ we have $\Gamma \vdash F \Rightarrow \hat{V}$ where $\mathrm{disp}(A' \sqcup A'', \hat{V}) \Rightarrow B$. By Lemma C.7, $\mathrm{disp}(A', \hat{V}) \Rightarrow B'$ for some $B' \leq B$ and $\mathrm{disp}(A'', \hat{V}) \Rightarrow B''$ for some $B'' \leq B$. Thus, $\Gamma; A' \vdash F \Leftarrow B$ and $\Gamma; A'' \vdash F \Leftarrow B''$.
(6) Inverting $\Gamma; \bot \vdash F \Rightarrow B$ we have $\Gamma \vdash F \Rightarrow \hat{V}$ where $\mathrm{disp}(\bot, \hat{V}) \Rightarrow B$. By Lemma C.7, $B \equiv \bot$.
(7) Proceed by case analysis on $\cdot \vdash F[v] \Leftarrow A$. The first case is $\cdot \vdash v\ v' \Leftarrow A$ where
   - $\cdot \vdash v \Rightarrow V$,
   - $\cdot \vdash v' \Rightarrow V'$,
   - $\mathrm{disp}(V, V') \Rightarrow A'$, and
   - $A' \leq A$.

   It follows that $\cdot; V \vdash [\cdot]v' \Leftarrow A$.
   The second case is $\cdot \vdash v\ [A'] \Leftarrow A$ where
   - $\cdot \vdash v \Rightarrow V$,
   - $\cdot \vdash A'$,
   - $\mathrm{disp}(V, [A']) \Rightarrow A''$, and
   - $A'' \leq A$.

   It follows that $\cdot; V \vdash [\cdot][A'] \Leftarrow A$.

□

Lemma C.21. *If* $\Gamma \vdash v \Leftarrow A_1 \sqcup A_2$ *and* $A_1 \diamond A_2$ *then* $\Gamma \vdash v \Leftarrow A_1$ *or* $\Gamma \vdash v \Leftarrow A_2$.

Proof. Follows from Lemma C.6. □

Lemma C.22. *Suppose* $\cdot \vdash A_1 \sqcap A_2$. *If* $\cdot; A_1 \sqcap A_2 \vdash F \Rightarrow B$ *then there exists a* B' *such that either* $\cdot; A_1 \vdash F \Rightarrow B'$ *or* $\cdot; A_2 \vdash F \Rightarrow B'$.

Proof. Follows from Lemma C.16. □

Lemma C.23 (Checking).

(1) *If* $\Gamma \vdash e \Leftarrow A$ *then* $\Gamma \vdash l\ e \Leftarrow l\ A$.
(2) *If* $\Gamma \vdash e \Leftarrow A_1$ *and* $\Gamma \vdash e' \Rightarrow A_2$ *and* $\mathrm{disp}(A_1, A_2) \Rightarrow B$, *then* $\Gamma \vdash e\ e' \Leftarrow B$.
(3) *If* $\Gamma \vdash e \Rightarrow A_1$ *and* $\Gamma \vdash e' \Leftarrow A_2$ *and* $\mathrm{disp}(A_1, A_2) \Rightarrow B$, *then* $\Gamma \vdash e\ e' \Leftarrow B$.
(4) *If* $\Gamma \vdash e \Leftarrow A_1$ *and* $\Gamma \vdash A_2$ *and* $\mathrm{disp}(A_1, [A_2]) \Rightarrow B$ *then* $\Gamma \vdash e\ [A_2] \Leftarrow B$.
(5) *If* $\Gamma \vdash e \Leftarrow A$ *and* $\Gamma \vdash e \Leftarrow B$ *then* $\Gamma \vdash e \Leftarrow A \sqcap B$.
(6) *If* $\cdot; A_1 \vdash F \Leftarrow B_1$ *and* $\cdot; A_2 \vdash F \Leftarrow B_2$ *then* $\cdot; A_1 \sqcap A_2 \vdash F \Leftarrow B_1 \sqcap B_2$.

(7) *If* $\cdot; A_1 \vdash F \Leftarrow B_1$ *and* $\cdot; A_2 \vdash F \Leftarrow B_2$ *then* $\cdot; A_1 \sqcup A_2 \vdash F \Leftarrow B_1 \sqcup B_2$.

Proof.

(1) By inversion on $\Gamma \vdash e \Leftarrow A$, there exists $B \leq A$ such that $\Gamma \vdash e \Rightarrow B$. By the typing rules, $\Gamma \vdash l\ e \Rightarrow l\ B$. It follows $\Gamma \vdash l\ e \Leftarrow l\ A$.
(2) By inversion on $\Gamma \vdash e \Leftarrow A_1$, there exists $A_1' \leq A_1$ such that $\Gamma \vdash e \Rightarrow A_1'$. By Lemma C.10, there exists $B' \leq B$ such that $\mathrm{disp}(A_1', A_2) \Rightarrow B'$. From the typing rules, $\Gamma \vdash e\ e' \Rightarrow B'$. It follows $\Gamma \vdash e\ e' \Leftarrow B$.
(3) By inversion on $\Gamma \vdash e' \Leftarrow A_2$, there exists $A_2' \leq A_2$ such that $\Gamma \vdash e' \Rightarrow A_2'$. By Lemma C.10, there exists $B' \leq B$ such that $\mathrm{disp}(A_1, A_2') \Rightarrow B'$. From the typing rules, $\Gamma \vdash e\ e' \Rightarrow B'$. It follows $\Gamma \vdash e\ e' \Leftarrow B$.
(4) By inversion on $\Gamma \vdash e \Leftarrow A_1$, there exists $A_1' \leq A_1$ such that $\Gamma \vdash e \Rightarrow A_1'$. By Lemma C.10, there exists $B' \leq B$ such that $\mathrm{disp}(A_1', [A_2]) \Rightarrow B'$. It follows $\Gamma \vdash e\ [A_2] \Leftarrow B$.
(5) Since $\Gamma \vdash e \Leftarrow A$ and $\Gamma \vdash e \Leftarrow B$, there must exist C such that $\Gamma \vdash e \Leftarrow C$ where $C \leq A$ and $C \leq B$. It follows $C \leq A \sqcap B$, so we have $\Gamma \vdash e \Leftarrow A \sqcap B$.
(6) From our hypotheses, $\cdot \vdash F \Rightarrow \hat{V}$ where $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ and $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$. By Lemma C.25, there exists B such that $\mathrm{disp}(A_1 \sqcap A_2, \hat{V}) \Rightarrow B$. Thus, $\cdot; A_1 \sqcap A_2 \vdash F \Rightarrow B$. By monotonicity (Lemma C.10), $B \leq B_1 \sqcap B_2$. It follows $\cdot; A_1 \sqcap A_2 \vdash F \Leftarrow B_1 \sqcap B_2$.
(7) From our hypotheses, $\cdot \vdash F \Rightarrow \hat{V}$ where $\mathrm{disp}(A_1, \hat{V}) \Rightarrow B_1$ and $\mathrm{disp}(A_2, \hat{V}) \Rightarrow B_2$. By Lemma C.25, there exists B such that $\mathrm{disp}(A_1 \sqcup A_2, \hat{V}) \Rightarrow B$. By Lemma C.17, $B \leq B_1 \sqcup B_2$. It follows $\cdot; A_1 \sqcup A_2 \vdash F \Leftarrow B_1 \sqcup B_2$.

□

Lemma C.24. *Suppose* $\cdot; A_1 \sqcap \ldots \sqcap A_n \vdash F \Rightarrow B$ *and for all* $i \in I$, $\cdot; A_i \vdash F \Rightarrow B_i$. *Then* $B \leq \bigsqcap_{i \in I} B_i$.

Proof. Follows from Lemma C.7. □

Lemma C.25 (Type-Level Dispatch).

(1) *If* $V \leq A$ *then there exists* C *such that* $\mathrm{disp}(A \to B, V) \Rightarrow C$.
(2) *There exists* C *such that* $\mathrm{disp}(\forall \alpha.B, [A]) \Rightarrow C$.
(3) *If* $\mathrm{disp}(A_1, V) \Rightarrow B_1$ *or* $\mathrm{disp}(A_2, V) \Rightarrow B_2$ *then there exists* B *such that* $\mathrm{disp}(A_1 \sqcap A_2, V) \Rightarrow B$.
(4) *If* $\mathrm{disp}(A_1, V) \Rightarrow B_1$ *and* $\mathrm{disp}(A_2, V) \Rightarrow B_2$ *then there exists* B *such that* $\mathrm{disp}(A_1 \sqcup A_2, V) \Rightarrow B$.

Proof. Coq proof in `ApplyTy.v` (`[1] applyty_arrow_complete; [2] applyty_forall_complete; [3] applyty_inter; [4] applyty_union`). □

Lemma C.26 (Evaluation Context Decomposition). *If* $\cdot \vdash E[e] \Rightarrow A$ *then there exists* B *such that* $\cdot \vdash e \Rightarrow B$.

Proof. Routine induction on the shape of E. The base case $E = [\cdot]$ is immediate. In each inductive case, we invert the derivation of $\cdot \vdash E[e] \Rightarrow A$ to find that $E'[e]$ is well typed for some E' smaller than E. Then by the induction hypothesis, e is well-typed. □

Lemma C.27 (Evaluation Context Replacement). *Suppose* $\cdot \vdash e_1 \Rightarrow A$ *and* $\cdot \vdash e_2 \Leftarrow A$. *If* $\cdot \vdash E[e_1] \Rightarrow B$ *then* $\cdot \vdash E[e_2] \Leftarrow B$.

Proof. Induction on the shape of E.

Case: $[\cdot]$

Since $\cdot \vdash e_1 \Rightarrow B$, we have $A = B$. It follows $\cdot \vdash e_2 \Leftarrow B$.

Case: $l\,E'$

By inversion on $\cdot \vdash l\,E'[e_1] \;\Rightarrow\; B$, there exists B' such that $\cdot \vdash E'[e_1] \;\Rightarrow\; B'$ and $B = l\,B'$. By the induction hypothesis, $\cdot \vdash E'[e_2] \;\Leftarrow\; B'$. By Lemma C.23, $\cdot \vdash l\,E'[e_2] \;\Leftarrow\; l\,B'$.

Case: $E'\,e'$

By inversion on $\cdot \vdash E'[e_1]\;e' \;\Rightarrow\; B$, there exists A_1 and A_2 such that:

- $\cdot \vdash E'[e_1] \;\Rightarrow\; A_1$,
- $\cdot \vdash e' \;\Rightarrow\; A_2$, and
- $\mathrm{disp}(A_1, A_2) \Rightarrow B$.

By the induction hypothesis, $\cdot \vdash E'[e_2] \;\Leftarrow\; A_1$. By Lemma C.23, $\cdot \vdash E'[e_2]\;e' \;\Leftarrow\; B$.

Case: $v\,E'$

By inversion on $\cdot \vdash v\,E'[e_1] \;\Rightarrow\; B$, there exists A_1 and A_2 such that:

- $\cdot \vdash v \;\Rightarrow\; A_1$,
- $\cdot \vdash E'[e_1] \;\Rightarrow\; A_2$, and
- $\mathrm{disp}(A_1, A_2) \Rightarrow B$.

By the induction hypothesis, $\cdot \vdash E'[e_2] \;\Leftarrow\; A_2$. By Lemma C.23, $\cdot \vdash v\,E'[e_2] \;\Leftarrow\; B$.

Case: $E'[A']$

By inversion on $\cdot \vdash E'[e_1]\;[A'] \;\Rightarrow\; B$, we have $\cdot \vdash A'$ and there exists A_1 such that $\cdot \vdash E'[e_1] \;\Rightarrow\; A_1$ and $\mathrm{disp}(A_1, [A']) \;\Rightarrow\; B$. By the induction hypothesis, $\cdot \vdash E'[e_2] \;\Leftarrow\; A_1$. By Lemma C.23, $\cdot \vdash E'[e_2]\;[A'] \;\Leftarrow\; B$.

Case: $(E' : B_1^-) \shortparallel (e' : B_2^-)$

By inversion on $\cdot \vdash (E'[e_1] : B_1^-) \shortparallel (e' : B_2^-) \;\Rightarrow\; B$ we have $B_1^- \bowtie B_2^-$ and $B = B_1^- \sqcap B_2^-$ where

- $\cdot \vdash E'[e_1] \;\Rightarrow\; B_1'$,
- $B_1' \leq B_1^-$, and
- $\cdot \vdash e' \;\Leftarrow\; B_2^-$.

By the induction hypothesis and transitivity of subtyping, $\cdot \vdash E'[e_2] \;\Leftarrow\; B_1^-$. By the typing rules, $\cdot \vdash (E'[e_2] : B_1^-) \shortparallel (e' : B_2^-) \;\Rightarrow\; B_1^- \sqcap B_2^-$.

Case: $(v : B_1^-) \shortparallel (E' : B_2^-)$

By inversion on $\cdot \vdash (v : B_1^-) \shortparallel (E'[e_1] : B_2^-) \;\Rightarrow\; B$ we have $B_1^- \bowtie B_2^-$ and $B = B_1^- \sqcap B_2^-$ where

- $\cdot \vdash v \;\Leftarrow\; B_1^-$,
- $\cdot \vdash E'[e_1] \;\Rightarrow\; B_2'$, and
- $B_2' \leq B_2^-$.

By the induction hypothesis and transitivity of subtyping, $\cdot \vdash E'[e_2] \;\Leftarrow\; B_2^-$. By the typing rules, $\cdot \vdash (v : B_1^-) \shortparallel (E'[e_2] : B_2^-) \;\Rightarrow\; B_1^- \sqcap B_2^-$. □

C.4 Properties of Substitution

Lemma C.28 (Substitution Typing).

1. *If* $\Gamma \vdash \sigma_1 \Leftarrow \Delta_1$ *and* $\sigma = \sigma_1 \sqcup \sigma_2$ *and* $\Delta = \Delta_1 \sqcup \Delta_2$ *then* $\Gamma \vdash \sigma \Leftarrow \Delta$.
2. *If* $\Gamma \vdash \sigma_1 \Leftarrow \Delta_1$ *and* $\Gamma \vdash \sigma_2 \Leftarrow \Delta_2$ *and* $\sigma = \sigma_1 \sqcap \sigma_2$ *and* $\Delta = \Delta_1 \sqcap \Delta_2$ *then* $\Gamma \vdash \sigma \Leftarrow \Delta$.
3. *If* $\cdot; A_1 \vdash \sigma_1 \Leftarrow \cdot \hookrightarrow B_1$ *and* $\cdot; A_2 \vdash \sigma_2 \Leftarrow \cdot \hookrightarrow B_2$ *and* $\sigma = \sigma_1 \sqcup \sigma_2$ *then* $\cdot; A_1 \sqcap A_2 \vdash \sigma \Leftarrow \cdot \hookrightarrow B_1 \sqcap B_2$.
4. *If* $\cdot; A_1 \vdash \sigma_1 \Leftarrow \cdot \hookrightarrow B_1$ *and* $\cdot; A_2 \vdash \sigma_2 \Leftarrow \cdot \hookrightarrow B_2$ *and* $\sigma = \sigma_1 \sqcap \sigma_2$ *then* $\cdot; A_1 \sqcup A_2 \vdash \sigma \Leftarrow \cdot \hookrightarrow B_1 \sqcup B_2$.

Proof.

(1) By definition, each $[x \mapsto v]$ in σ occurs in both σ_1 and σ_2. The typing of σ_1 implies $x : A \in \Delta_1$ and $\Gamma \vdash v \Leftarrow A$. The definition of Δ ensures that if $x : B \in \Delta$ for some B then $A \leq B$. Transitivity of subtyping yields $\Gamma \vdash v \Leftarrow B$.
Similarly, each $[\alpha \mapsto A]$ in σ occurs in both σ_1 and σ_2. The typing of σ_1 implies $\alpha \in \Delta_1$ and $\Gamma \vdash A$. The definition of Δ ensures each α in Δ is also in Δ_1.
It follows $\Gamma \vdash \sigma \Leftarrow \Delta$.
(2) By definition, each $[x \mapsto v]$ in σ occurs in either σ_1 or σ_2. Without loss of generality, consider the following two cases.
   - $[x \mapsto v]$ occurs in σ_1 but not σ_2. The typing of σ_1 implies $x : A_1 \in \Delta_1$ and $\Gamma \vdash v \Leftarrow A_1$. The definition of Δ ensures $x : A_1 \in \Delta$.
   - $[x \mapsto v]$ occurs in σ_1 and σ_2. The typing of σ_1 and σ_2 imply $x : A_i \in \Delta_i$ and $\Gamma \vdash v \Leftarrow A_i$ for all $i \in 1, 2$. By Lemma C.23, it follows that $\Gamma \vdash v \Leftarrow A_1 \sqcap A_2$ The definition of Δ ensures $x : A_1 \sqcap A_2 \in \Delta$.

   Similarly, each $[\alpha \mapsto A]$ in σ occurs in either σ_1 or σ_2. Without loss of generality, assume the former. The typing of σ_1 ensures $\alpha \in \Delta_1$ and $\Gamma \vdash A$. The definition of Δ ensures each α in Δ is also in Δ_1 or Δ_2.
From both of these cases it follows $\Gamma \vdash \sigma \Leftarrow \Delta$.
(3) The typing of σ_1 and σ_2 implies that they each contain a mapping of the form $[* \mapsto F_i]$. The fact that σ is well-defined means that the elimination frames in each are the same. That is, both σ_1 and σ_2 contain $[* \mapsto F]$. The typing derivations give us the following facts:
   - $\cdot; A_1 \vdash F \Leftarrow B_1$
   - $\cdot; A_2 \vdash F \Leftarrow B_2$

   From Lemma C.23, $\cdot; A_1 \sqcap A_2 \vdash F \Leftarrow B_1 \sqcap B_2$. It follows $\cdot; A_1 \sqcap A_2 \vdash \sigma \Leftarrow \cdot \hookrightarrow B_1 \sqcap B_2$.
(4) The typing of σ_1 and σ_2 implies that they each contain a mapping of the form $[* \mapsto F_i]$. The fact that σ is well-defined means that the elimination frames in each are the same. That is, both σ_1 and σ_2 contain $[* \mapsto F]$. The typing derivations give us the following facts:
   - $\cdot; A_1 \vdash F \Leftarrow B_1$
   - $\cdot; A_2 \vdash F \Leftarrow B_2$

   From Lemma C.23, $\cdot; A_1 \sqcup A_2 \vdash F \Leftarrow B_1 \sqcup B_2$. It follows $\cdot; A_1 \sqcup A_2 \vdash \sigma \Leftarrow \cdot \hookrightarrow B_1 \sqcup B_2$.

□

Lemma C.29 (Type Substitution). *If* $\Gamma_1, \alpha, \Gamma_2 \vdash A$ *and* $\Gamma_1 \vdash B$ *then* $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A[B/\alpha]$.

Proof. Proceed by induction on $\Gamma_1, \alpha, \Gamma_2 \vdash A$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash \alpha$
We have $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash B$ by weakening on our assumption.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash \beta$ where $\alpha \neq \beta$
Either $\beta \in \Gamma_1$ or $\beta \in \Gamma_2$. It follows $\beta \in \Gamma_1$ or $\Gamma_2[B/\alpha]$ so $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash \beta$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash l\, A'$ where $\Gamma_1, \alpha, \Gamma_2 \vdash A'$
By the induction hypothesis, $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A'[B/\alpha]$. It follows $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash l\,(A'[B/\alpha])$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash A_1 \sqcap A_2$ where $\Gamma_1, \alpha, \Gamma_2 \vdash A_1$ and $\Gamma_1, \alpha, \Gamma_2 \vdash A_2$ and $A_1 \bowtie A_2$
By the induction hypothesis, $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1[B/\alpha]$ and $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_2[B/\alpha]$. Since mergeability is closed under substitution (Lemma C.35), $A_1[B/\alpha] \bowtie A_2[B/\alpha]$. It follows $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1 \sqcap A_2$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash A_1 \sqcup A_2$ where $\Gamma_1, \alpha, \Gamma_2 \vdash A_1$ and $\Gamma_1, \alpha, \Gamma_2 \vdash A_2$
By the induction hypothesis, $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1[B/\alpha]$ and $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_2[B/\alpha]$. It follows $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1 \sqcup A_2$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash A_1 \to A_2$ where $\Gamma_1, \alpha, \Gamma_2 \vdash A_1$ and $\Gamma_1, \alpha, \Gamma_2 \vdash A_2$
By the induction hypothesis, $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1[B/\alpha]$ and $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_2[B/\alpha]$. It follows $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash A_1 \to A_2$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash \forall \beta.A'$ where $\Gamma_1, \alpha, \Gamma_2, \beta \vdash A'$
By the induction hypothesis, $\Gamma_1, (\Gamma_2[B/\alpha]), \beta \vdash A'[B/\alpha]$. It follows $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash \forall \beta.A'[B/\alpha]$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash \top$
It is immediate that $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash \top$.

Case: $\Gamma_1, \alpha, \Gamma_2 \vdash \bot$
It is immediate that $\Gamma_1, (\Gamma_2[B/\alpha]) \vdash \bot$.

□

Lemma C.30 (Monotonicity of Type Substitution). *If $A \leq B$ then $A[C/\alpha] \leq B[C/\alpha]$.*

Proof. Coq proof in `DistSubtyping.v` (`typsubst_typ_algo_sub`). Note that the subtyping algorithm depends on type splitting which is sensitive to type substitution (e.g. $\text{split}_\text{u}(\alpha \sqcap (A_1 \sqcup A_2)) \Rightarrow \alpha \sqcap A_1 \sqcup \alpha \sqcap A_2$ while $\text{split}_\text{u}((B_1 \sqcup B_2) \sqcap (A_1 \sqcup A_2)) \Rightarrow B_1 \sqcap (A_1 \sqcup A_2) \sqcup B_2 \sqcap (A_1 \sqcup A_2)$). Thus we define another non-deterministic type splitting relation that is stable under type substitution and prove that using it to replace the type splitting we have in subtyping does not change its expressiveness. For this new version of subtyping, the goal can be directly proved. □

Lemma C.31 (Type Substitution Over Type-Level Dispatch). *If $\text{disp}(A_1, A_2) \Rightarrow C$ then there exists C' such that $C' \leq C[B/\alpha]$ and $\text{disp}(A_1[B/\alpha], A_2[B/\alpha]) \Rightarrow C'$.*

Proof. Coq proof in `ApplyTy.v` (`typsubst_applyty`).ApplyTy.v Via the soundness property of the dispatch operator (Lemma C.8), we derive $A_1 \leq A_2 \to C$ from the premise. Then we perform type substitution on the subtyping judgment (Lemma C.30). After it we use the completeness of type-level dispatch (Lemma C.32) to obtain the goal. □

Lemma C.32 (Completeness of Type-Level Dispatch).

(1) *If $A \leq B \to C$ then there exists C' such that $\text{disp}(A, B) \Rightarrow C'$ and $B \to C' \leq B \to C$.*
(2) *If $A \leq \forall \alpha.B$ then there exists C such that $\text{disp}(A, [\alpha]) \Rightarrow C$ and $\forall \alpha.C \leq \forall \alpha.B$.*

Proof. Coq proof in `ApplyTy.v` (`applyty_completeness_1_all and applyty_completeness_2`). □

Lemma C.33 (Substitution). *Suppose $\cdot \vdash \Gamma_1$ and $\Gamma_1 \vdash \Delta$ and $\Gamma_1, \Delta \vdash \Gamma_2$.*

(1) *If $\Gamma_1, \Delta, \Gamma_2 \vdash e \Leftarrow A$ and $\Gamma_1 \vdash \sigma \Leftarrow \Delta$ then $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e) \Leftarrow \sigma(A)$.*
(2) *If $\Gamma_1, \Delta, \Gamma_2 \vdash M \Leftarrow A$ and $\Gamma_1 \vdash \sigma \Leftarrow \Delta$ then $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(M) \Leftarrow \sigma(A)$.*
(3) *If $\Gamma_1, \Delta, \Gamma_2; \hat{p} \vdash e \Rightarrow A$ and $\Gamma_1 \vdash \sigma \Leftarrow \Delta$ then $\Gamma_1, \sigma(\Gamma_2); \sigma(\hat{p}) \vdash \sigma(e) \Rightarrow \sigma(A)$.*
(4) *If $\Gamma_1, \Delta, \Gamma_2 \vdash \hat{p} \Rightarrow A^- \hookrightarrow B \dashv \Delta'$ and $\Gamma_1 \vdash \sigma \Leftarrow \Delta$ then $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(\hat{p}) \Rightarrow \sigma(A) \hookrightarrow \sigma(B) \dashv \sigma(\Delta')$.*
(5) *If $\Gamma_1, \Delta, \Gamma_2 \vdash p \Rightarrow A \dashv \Delta'$ and $\Gamma_1 \vdash \sigma \Leftarrow \Delta$ then $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p) \Rightarrow \sigma(A) \dashv \sigma(\Delta')$.*
(6) *If $\Gamma_1, \Delta, \Gamma_2 \vdash e \Leftarrow A$ and $\Gamma_1; A \vdash \sigma \Leftarrow \Delta \hookrightarrow B$ then $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e) \Leftarrow \sigma(B)$.*

Proof. We proceed by mutual induction on the typing rules.

For part (1), we invert $\Gamma_1, \Delta, \Gamma_2 \vdash e \Leftarrow A$ to find $\Gamma_1, \Delta, \Gamma_2 \vdash e \Rightarrow A'$ and $A' \leq A$. We proceed by induction on the derivation of the former fact.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash x \Rightarrow A'$ where $x : A' \in \Gamma_1$
Note that $\cdot \vdash \Gamma_1$ and Γ_1 is disjoint with Δ_1. It follows $\sigma(A') = A'$ and $\sigma(A') = A'$. Therefore, from the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash x \Rightarrow A'$.

By monotonicity of type substitution (Lemma C.30), $A' = \sigma(A') \le \sigma(A)$. Therefore, $\Gamma_1, \sigma(\Gamma_2) \vdash x \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash x \Rightarrow A'$ where $x : A' \in \Delta$
From $\Gamma_1 \vdash \sigma \Leftarrow \Delta$, we have $\sigma(A') = A'$ and there exists v such that $\sigma(x) = v$ and $\Gamma_1 \vdash v \Leftarrow A'$. By weakening, $\Gamma_1, \sigma(\Gamma_2) \vdash v \Leftarrow A'$.

By monotonicity of type substitution (Lemma C.30), $A' = \sigma(A') \le \sigma(A)$. Therefore, $\Gamma_1, \Gamma_2 \vdash v \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash x \Rightarrow A'$ where $x : A' \in \Gamma_2$
Note that $\Gamma_1, \Delta \vdash \Gamma_2$ and Γ_2 is disjoint with Δ_1. It follows $\sigma(x) = x$. Therefore, from the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash x \Rightarrow \sigma(A')$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash l\, e' \Rightarrow l\, A''$ where $A' = l\, A''$ and $\Gamma_1, \Delta, \Gamma_2 \vdash e' \Rightarrow A''$
By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e') \Leftarrow \sigma(A'')$. Therefore by the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash l\, \sigma(e') \Leftarrow l\, \sigma(A'')$.

By monotonicity of type substitution (Lemma C.30), $\sigma(l\, A'') \le \sigma(A)$. It follows $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(l\, e') \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash e_1\ e_2 \Rightarrow A'$ where $\Gamma_1, \Delta, \Gamma_2 \vdash e_1 \Rightarrow A'_1$ and $\Gamma_1, \Delta, \Gamma_2 \vdash e_2 \Rightarrow A'_2$ and $\mathrm{disp}(A'_1, A'_2) \Rightarrow A'$
By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_1) \Leftarrow \sigma(A'_1)$ and $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_2) \Leftarrow \sigma(A'_2)$. Therefore, there exists $A''_1 \le \sigma(A'_1)$ and $A''_2 \le \sigma(A'_2)$ such that $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_1) \Rightarrow A''_1$ and $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_2) \Rightarrow A''_2$.

Applying substitution over the dispatch operator (Lemma C.31) to $\mathrm{disp}(A'_1, A'_2) \Rightarrow A'$, we have that there exists $A'' \le \sigma(A')$ such that $\mathrm{disp}(\sigma(A'_1), \sigma(A'_2)) \Rightarrow A''$. Applying monotonicity (Lemma C.10) and transitivity of subtyping, we have that there exists an $A''' \le \sigma(A')$ such that $\mathrm{disp}(A''_1, A''_2) \Rightarrow A'''$. Therefore by the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_1)\, \sigma(e_2) \Rightarrow A'''$.

Note $\sigma(A') \le \sigma(A)$ by monotonicity of type substitution (Lemma C.30). It follows $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e_1\ e_2) \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash e'\,[A'_2] \Rightarrow A'$ where $\Gamma_1, \Delta, \Gamma_2 \vdash e' \Rightarrow A'_1$ and $\Gamma_1, \Delta, \Gamma_2 \vdash A'_2$ and $\mathrm{disp}(A'_1, [A'_2]) \Rightarrow A'$
By the induction hypothesis, there exists $A''_1 \le \sigma(A'_1)$ such that $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e') \Rightarrow A''_1$. By the type substitution lemma (Lemma C.29), $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(A'_2)$.

Applying monotonicity (Lemma C.10) and type substitution over the dispatch operator (Lemma C.31) to $\mathrm{disp}(A'_1, [A'_2]) \Rightarrow A'$, we have that there exists $A'' \le \sigma(A')$ such that $\mathrm{disp}(A''_1, [\sigma(A'_2)]) \Rightarrow A''$. Therefore, by the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e')\ [\sigma(A'_2)] \Rightarrow A''$.

Note $\sigma(A') \le \sigma(A)$ by monotonicity of type substitution (Lemma C.30). It follows $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(e'\ [A'_2]) \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash (e_1 : A'_1) \shortparallel (e_2 : A'_2) \Rightarrow A'_1 \sqcap A'_2$ where $A' = A'_1 \sqcap A'_2$ and A'_1 is negative and A'_2 is negative and $\Gamma_1, \Delta, \Gamma_2 \vdash e_1 \Leftarrow A'_1$ and $\Gamma_1, \Delta, \Gamma_2 \vdash e_2 \Leftarrow A'_2$ and $A'_1 \bowtie A'_2$
By the induction hypothesis, $\Gamma_1 \vdash \sigma(e_1) \Leftarrow \sigma(A'_1)$ and $\Gamma_1 \vdash \sigma(e_2) \Leftarrow \sigma(A'_2)$.

Note $\sigma(A'_1)$ and $\sigma(A'_2)$ are negative. By the substitution over disjoint types lemma (Lemma C.35), $\sigma(A'_1) \bowtie \sigma(A'_2)$. Therefore, by the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash (\sigma(e_1) : \sigma(A'_1)) \shortparallel (\sigma(e_2) : \sigma(A'_2)) \Rightarrow \sigma(A'_1) \sqcap \sigma(A'_2)$.

Note $\sigma(A'_1 \sqcap A'_2) \le \sigma(A)$ by monotonicity of type substitution (Lemma C.30). It follows $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma((e_1 : A'_1) \shortparallel (e_2 : A'_2)) \Leftarrow \sigma(A)$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash \lambda M \Rightarrow A'$ where $\Gamma_1, \Delta, \Gamma_2 \vdash M \Rightarrow A'$
By the induction hypothesis, there exists $A'' \le \sigma(A')$ such that $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(M) \Rightarrow A''$. It follows immediately from the typing rules that $\Gamma_1, \sigma(\Gamma_2) \vdash \lambda\sigma(M) \Rightarrow A''$.

Note $\sigma(A') \le \sigma(A'')$ by the monotonicity of type substitution (Lemma C.30). It follows $\Gamma_1, \sigma(\Gamma) \vdash \sigma(\lambda M) \Leftarrow \sigma(A'')$.

For part (2), we invert $\Gamma_1, \Delta, \Gamma_2 \vdash M \Leftarrow A$ to find $M = \{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\}$ where $\Gamma_1, \Delta, \Gamma_2; \hat{p}_i \vdash e_i \Rightarrow A_i$ for $1 \leq i \leq n$ and $A_1 \bowtie \ldots \bowtie A_n$ and $A_1 \sqcap \ldots \sqcap A_n \leq A$.

By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2); \sigma(\hat{p}_i) \vdash \sigma(e_i) \Rightarrow \sigma(A_i)$ for $1 \leq i \leq n$.

By substitution over mergeable types (Lemma C.35), $\sigma(A_1) \bowtie \ldots \bowtie \sigma(A_n)$. It follows that $\Gamma_1, \sigma(\Gamma_2) \vdash \{\sigma(\hat{p}_1) \mapsto \sigma(e_1), \ldots, \sigma(\hat{p}_n) \mapsto \sigma(e_n)\} \Leftarrow \sigma(A_1) \sqcap \ldots \sqcap \sigma(A_n)$.

By the monotonicity of type substitution (Lemma C.30), $\sigma(A_1 \sqcap \ldots \sqcap A_n) \leq \sigma(A)$. Thus, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(M) \Leftarrow \sigma(A)$.

For part (3), we invert $\Gamma_1, \Delta, \Gamma_2; \hat{p}_i \vdash e \Rightarrow A$ to find $\Gamma_1, \Delta, \Gamma_2 \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta'$ and $\Gamma_1, \Delta, \Gamma_2, \Delta' \vdash e \Leftarrow B$.

Note Δ' is disjoint from Γ_1 and Δ and Γ_2. By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(\hat{p}) \Rightarrow \sigma(A) \hookrightarrow \sigma(B) \dashv \sigma(\Delta')$ and $\Gamma, \sigma(\Gamma_2), \sigma(\Delta') \vdash \sigma(e) \Leftarrow \sigma(B)$.

It follows from the typing rules that $\Gamma_1, \sigma(\Gamma_2); \sigma(\hat{p}) \vdash \sigma(e) \Rightarrow \sigma(A)$.

For part (4), we proceed by induction on $\Gamma_1, \Delta, \Gamma_2 \vdash \hat{p} \Rightarrow A^- \hookrightarrow B \dashv \Delta'$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash (* \hookrightarrow A^-) \Rightarrow A^- \hookrightarrow A^- \dashv \cdot$ where $\Gamma_1, \Delta, \Gamma_2 \vdash A^-$ and $A^- = B$
By the type substitution lemma (Lemma C.29), $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(A^-)$. Therefore, by the typing rules, $\Gamma_1, \sigma(\Gamma_2) \vdash (* \hookrightarrow \sigma(A^-)) \Rightarrow \sigma(A^-) \hookrightarrow \sigma(A^-) \dashv \cdot$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash (p \hookrightarrow B) \Rightarrow (A' \to B) \hookrightarrow B \dashv \Delta'$ where $\Gamma_1, \Delta, \Gamma_2 \vdash B$ and $\Gamma_1, \Delta, \Gamma_2 \vdash p \Rightarrow A' \dashv \Delta'$ and $A^- = A' \to B$
By the type substitution lemma (Lemma C.29), $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(B)$. By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p) \Rightarrow \sigma(A') \dashv \sigma(\Delta')$.

Now applying the typing rules we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash (\sigma(p) \hookrightarrow \sigma(B)) \Rightarrow (\sigma(A') \to \sigma(B)) \hookrightarrow \sigma(B) \dashv \sigma(\Delta')$$

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash (\alpha \hookrightarrow B) \Rightarrow (\forall \alpha.B) \hookrightarrow B \dashv \alpha$ where $\Gamma_1, \Delta, \Gamma_2, \alpha \vdash B$ and $A^- = \forall \alpha.B$
Note $\sigma(\alpha) = \alpha$. By the type substitution lemma (Lemma C.29), $\Gamma, \sigma(\Gamma_2), \alpha \vdash \sigma(B)$.

Now applying the typing rules we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash (\alpha \hookrightarrow \sigma(B)) \Rightarrow (\forall \alpha.\sigma(B)) \hookrightarrow \sigma(B) \dashv \alpha$$

For part (5), we proceed by induction on $\Gamma_1, \Delta, \Gamma_2 \vdash p \Rightarrow A \dashv \Delta'$.

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash x : A \Rightarrow A \dashv x : A$ where $\Gamma_1, \Delta, \Gamma_2 \vdash A$ and $\Delta' = x : A$
By the type substitution lemma (Lemma C.29), $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(A)$.

Now applying the typing rules we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash x : \sigma(A) \Rightarrow \sigma(A) \dashv x : \sigma(A)$$

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash l\,p' \Rightarrow l\,A' \dashv \Delta'$ where $\Gamma_1, \Delta, \Gamma_2 \vdash p' \Rightarrow A' \dashv \Delta'$ and $A = l\,A'$
By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p') \Rightarrow \sigma(A') \dashv \sigma(\Delta')$.

Now applying the typing rules we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash l\,\sigma(p') \Rightarrow l\,\sigma(A') \dashv \sigma(\Delta')$$

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash p_1 | p_2 \Rightarrow A_1 \sqcup A_2 \dashv \Delta'_1 \sqcup \Delta'_2$ where $\Gamma_1, \Delta, \Gamma_2 \vdash p_1 \Rightarrow A_1 \dashv \Delta'_1$ and $\Gamma_1, \Delta, \Gamma_2 \vdash p_2 \Rightarrow A_2 \dashv \Delta'_2$ and $p_1 \sim p_2$ and $A = A_1 \sqcup A_2$ and $\Delta' = \Delta'_1 \sqcup \Delta'_2$
By substitution over consistent patterns (Lemma C.36), $\sigma(p_1) \sim \sigma(p_2)$. By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p_1) \Rightarrow \sigma(A_1) \dashv \sigma(\Delta'_1)$ and $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p_2) \Rightarrow \sigma(A_2) \dashv \sigma(\Delta'_2)$.

Now applying the typing rules we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p_1) | \sigma(p_2) \Rightarrow \sigma(A_1) \sqcup \sigma(A_2) \dashv \sigma(\Delta'_1) \sqcup \sigma(\Delta'_2)$$

Case: $\Gamma_1, \Delta, \Gamma_2 \vdash p_1 \& p_2 \Rightarrow A_1 \sqcap A_2 \dashv \Delta'_1 \sqcap \Delta'_2$ where $\Gamma_1, \Delta, \Gamma_2 \vdash p_1 \Rightarrow A_1 \dashv \Delta'_1$ and $\Gamma_1, \Delta, \Gamma_2 \vdash p_2 \Rightarrow A_2 \dashv \Delta'_2$ and $A_1 \bowtie A_2$ and $p_1 \sim p_2$ and $A = A_1 \sqcap A_2$ and $\Delta' = \Delta'_1 \sqcap \Delta'_2$
By the induction hypothesis, $\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(p_1) \Rightarrow \sigma(A_1) \dashv \sigma(\Delta'_1)$ and $\Gamma_2, \sigma(\Gamma_2) \vdash \sigma(p_2) \Rightarrow \sigma(A_2) \dashv \sigma(\Delta'_2)$.

By substitution over mergeable types (Lemma C.35), $\sigma(A_1) \bowtie \sigma(A_2)$ By substitution over consistent patterns (Lemma C.36), $\sigma(p_1) \sim \sigma(p_2)$.

Now applying the typing rules we have

$$\Gamma, \sigma(\Gamma_2) \vdash \sigma(p_1) \& \sigma(p_2) \Rightarrow \sigma(A_1) \sqcap \sigma(A_2) \dashv \sigma(\Delta'_1) \sqcap \sigma(\Delta'_2)$$

□

Lemma C.34 (Substitution Over Distinguishable Types). *If $A_1 \diamond A_2$ then $A_1[B/\alpha] \diamond A_2[B/\alpha]$.*

Proof. Proceed by induction on $A_1 \diamond A_2$.

Case: $l\ A'_1 \diamond l\ A'_2$ where $A'_1 \diamond A'_2$
By the induction hypothesis, $A'_1[B/\alpha] \diamond A'_2[B/\alpha]$. It follows $l\ A'_1[B/\alpha] \diamond l\ A'_2[B/\alpha]$.

Case: $A'_1 \sqcup A''_1 \diamond A_2$ where $A'_1 \diamond A_2$ and $A''_1 \diamond A_2$
By the induction hypothesis, $A'_1[B/\alpha] \diamond A_2[B/\alpha]$ and $A''_1[B/\alpha] \diamond A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcup A''_1[B/\alpha] \diamond A_2[B/\alpha]$.

Case: $A'_1 \sqcap A''_1 \diamond A_2$ where $A'_1 \diamond A_2$
By the induction hypothesis, $A'_1[B/\alpha] \diamond A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcap A''_1[B/\alpha] \diamond A_2[B/\alpha]$.

Case: $A'_1 \sqcap A''_1 \diamond A_2$ where $A''_1 \diamond A_2$
By the induction hypothesis, $A''_1[B/\alpha] \diamond A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcap A''_1[B/\alpha] \diamond A_2[B/\alpha]$.

Case: $\bot \diamond A_2$
We have immediately that $\bot \diamond A_2[B/\alpha]$.

Case: $A'_1 \sqcap (A''_1 \sqcup A'''_1) \diamond A_2$ where $A'_1 \sqcap A''_1 \diamond A_2$ and $A'_1 \sqcap A'''_1 \diamond A_2$
By the induction hypothesis, $A'_1[B/\alpha] \sqcap A''_1[B/\alpha] \diamond A_2[B/\alpha]$ and $A'_1[B/\alpha] \sqcap A'''_1[B/\alpha] \diamond A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcap (A''_1[B/\alpha] \sqcup A'''_1[B/\alpha]) \diamond A_2[B/\alpha]$.

Case: $l_1\ A'_1 \diamond l_2\ A'_2$ where $l_1 \neq l_2$
We have immediately that $l_1\ A'_1[B/\alpha] \diamond l_2\ A'_2[B/\alpha]$. □

Lemma C.35 (Substitution Over Mergeable Types). *If $A_1 \bowtie A_2$ then $A_1[B/\alpha] \bowtie A_2[B/\alpha]$.*

Proof. Proceed by induction on $A_1 \bowtie A_2$.

Case: $l\ A'_1 \bowtie l\ A'_2$ where $A'_1 \bowtie A'_2$
By the induction hypothesis, $A'_1[B/\alpha] \bowtie A'_2[B/\alpha]$. It follows $l\ A'_1[B/\alpha] \bowtie l\ A'_2[B/\alpha]$.

Case: $A'_1 \to A''_1 \bowtie A'_2 \to A''_2$ where $A'_1 \diamond A'_2$
By Lemma C.34, $A'_1[B/\alpha] \diamond A'_2[B/\alpha]$. It follows $A'_1[B/\alpha] \to A''_1[B/\alpha] \bowtie A'_2[B/\alpha] \to A''_2[B/\alpha]$.

Case: $A' \to A''_1 \bowtie A' \to A''_2$ where $A''_1 \bowtie A''_2$
By the induction hypothesis, $A''_1[B/\alpha] \bowtie A''_2[B/\alpha]$. It follows $A'[B/\alpha] \to A''_1[B/\alpha] \bowtie A'[B/\alpha] \to A''_2[B/\alpha]$.

Case: $\forall \beta.A'_1 \bowtie \forall \beta.A'_2$ where $A'_1 \bowtie A'_2$
By the induction hypothesis, $A'_1[B/\alpha] \bowtie A'_2[B/\alpha]$. It follows $\forall \beta.A'_1[B/\alpha] \bowtie \forall \beta.A'_2[B/\alpha]$.

Case: $A'_1 \sqcup A''_1 \bowtie A_2$ where $A'_1 \bowtie A_2$ and $A''_1 \bowtie A_2$
By the induction hypothesis, $A'_1[B/\alpha] \bowtie A_2[B/\alpha]$ and $A''_1[B/\alpha] \bowtie A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcup A''_1[B/\alpha] \bowtie A_2[B/\alpha]$

Case: $A'_1 \sqcap A''_1 \bowtie A_2$ where $A'_1 \bowtie A_2$ and $A''_1 \bowtie A_2$
By the induction hypothesis, $A'_1[B/\alpha] \bowtie A_2[B/\alpha]$ and $A''_1[B/\alpha] \bowtie A_2[B/\alpha]$. It follows $A'_1[B/\alpha] \sqcap A''_1[B/\alpha] \bowtie A_2[B/\alpha]$

Case: $\top \bowtie A_2$
We have immediately that $\top \bowtie A_2[B/\alpha]$.
Case: $A'_1 \to A''_1 \bowtie \forall \beta.A'_2$
We have immediately that $A'_1[B/\alpha] \to A''_1[B/\alpha] \bowtie \forall \beta.A'_2[B/\alpha]$. □

Lemma C.36 (Substitution Over Consistent Patterns). *If $p_1 \sim p_2$ then $\sigma(p_1) \sim \sigma(p_2)$.*

Proof. Proceed by induction on $p_1 \sim p_2$.
Case: $(x : A) \sim (y : B)$
It is immediate that $(x : \sigma(A)) \sim (y : \sigma(B))$.
Case: $l\ p'_1 \sim l\ p'_2$ where $p'_1 \sim p'_2$
By the induction hypothesis, $\sigma(p'_1) \sim \sigma(p'_2)$. It follows $l\ \sigma(p'_1) \sim l\ \sigma(p'_2)$.
Case: $p'_1|p''_1 \sim p_2$ where $p'_1 \sim p_2$ and $p''_1 \sim p_2$
By the induction hypothesis, $\sigma(p'_1) \sim \sigma(p_2)$ and $\sigma(p''_1) \sim \sigma(p_2)$. It follows $\sigma(p'_1)|\sigma(p''_1) \sim \sigma(p_2)$.
Case: $p'_1 \& p''_1 \sim p_2$ where $p'_1 \sim p_2$ and $p''_1 \sim p_2$
By the induction hypothesis, $\sigma(p'_1) \sim \sigma(p_2)$ and $\sigma(p''_1) \sim \sigma(p_2)$. It follows $\sigma(p'_1) \& \sigma(p''_1) \sim \sigma(p_2)$. □

Lemma C.37. *Suppose $\cdot; A_1 \sqcap \ldots \sqcap A_n \vdash F \implies A'$ where $n \geq 1$ and $\cdot \vdash A_1 \sqcap \ldots \sqcap A_n$. Let J be the subset of $\{1, \ldots, n\}$ where there exists A'_j such that $\cdot; A_j \vdash F \implies A'_j$ for each $j \in J$. Then $\bigsqcap_{j \in J} A'_j \leq A'$.*

Proof. First, note that J is non-empty by Lemma C.17. Proceed by induction on n.
Case: $n = 1$
In this case, $J = \{1\}$. It follows $A'_1 = A'$, so we need only trivially show $A'_1 \leq A'_1$.
Case: $n > 1$
Note $A_1 \sqcap \ldots \sqcap A_n = A_1 \sqcap (A_2 \sqcap \ldots \sqcap A_n)$. There are three subcases:

(1) $1 \in J$ and $\exists j \in \{2, \ldots, n\} \cap J$
(2) $1 \notin J$ and $\exists j \in \{2, \ldots, n\} \cap J$
(3) $1 \in J$ and $\nexists j \in \{2, \ldots, n\} \cap J$

In the first subcase, we have $\cdot; A_1 \vdash F \implies A'_1$ and $\cdot; A_2 \sqcap \ldots \sqcap A_n \vdash F \implies B$. By the induction hypothesis, $\bigsqcap_{j \in J \setminus \{1\}} A'_j \leq B$. By Lemma C.15, $A'_1 \sqcap B \leq A'$. It follows $\bigsqcap_{j \in J} A'_j \leq A'$.

In the second subcase, we have $\cdot; A_1 \vdash F \not\Rightarrow$ and $\cdot; A_2 \sqcap \ldots \sqcap A_n \vdash F \implies B$. By the induction hypothesis, $\bigsqcap_{j \in J} A'_j \leq B$. By Lemma C.15, $B \leq A'$. It follows $\bigsqcap_{j \in J} A'_j \leq A'$.

In the third subcase, we have $\cdot; A_1 \vdash F \implies A'_1$ and $\cdot; A_2 \sqcap \ldots \sqcap A_n \vdash F \not\Rightarrow$. By Lemma C.15, $A'_1 \leq A'$. □

C.5 Properties of Distinguishability

Lemma C.38 (Distinguishability). *If $A \nLeftrightarrow B$ and $V/B \Rightarrow^+ \checkmark$ then $V/A \not\Rightarrow^+$.*

Proof. By induction on $A \nLeftrightarrow B$.
Case: $l\ A' \nLeftrightarrow l\ B'$ where $A' \nLeftrightarrow B'$
From inversion on $V/l\ B' \Rightarrow^+ \checkmark$, we have $V = l\ V'$ where $V'/B' \Rightarrow^+ \checkmark$. By the induction hypothesis, $V'/A' \not\Rightarrow^+$. It follows $l\ V'/l\ A' \not\Rightarrow^+$.
Case: $A_1 \sqcup A_2 \nLeftrightarrow B$ where $A_1 \nLeftrightarrow B$ and $A_2 \nLeftrightarrow B$
By the induction hypotheses, $V/A_1 \not\Rightarrow^+$ and $V/A_2 \not\Rightarrow^+$. Suppose for contradiction $V/A_1 \sqcup A_2 \Rightarrow^+ \checkmark$. By inversion on this fact, either $V/A_1 \Rightarrow^+ \checkmark$ or $V/A_2 \Rightarrow^+ \checkmark$. These contradict our induction hypotheses.
Case: $A \nLeftrightarrow B_1 \sqcup B_2$ where $A \nLeftrightarrow B_1$ and $A \nLeftrightarrow B_2$
We have $V/(B_1 \sqcup B_2) \Rightarrow^+ \checkmark$. By inversion, either $V/B_1 \Rightarrow^+ \checkmark$ or $V/B_2 \Rightarrow^+ \checkmark$. By the induction hypotheses, if $V/B_i \Rightarrow^+ \checkmark$ for some $i \in 1, 2$ then $V/A \Rightarrow^+ \checkmark$

Case: $A_1 \sqcap A_2 \diamond B$ where $A_1 \diamond B$
By the induction hypothesis, $V/A_1 \not\Rightarrow^+$. Suppose for contradiction that $V/(A_1 \sqcap A_2) \Rightarrow^+ \checkmark$. By inversion, $V/A_1 \not\Rightarrow^+$ (and $V/A_2 \not\Rightarrow^+$). This contradicts our induction hypothesis.

Case: $A \diamond B_1 \sqcap B_2$ where $A \diamond B_1$
By inversion on $V/(B_1 \sqcap B_2) \Rightarrow^+ \checkmark$, we have $V/B_1 \Rightarrow^+ \checkmark$ and $V/B_2 \Rightarrow^+ \checkmark$. By the induction hypothesis, $V/A \not\Rightarrow^+$.

Case: $\bot \diamond B$
$V/\bot \not\Rightarrow^+$ follows from Lemma C.2.

Case: $A \diamond \bot$
$V/\bot \Rightarrow^+ \checkmark$ is a contradiction by Lemma C.2.

Case: $A \diamond B_1 \sqcap (B_2 \sqcup B_3)$ where $A \diamond B_1 \sqcap B_2$ and $A \diamond B_1 \sqcap B_3$
We have $V/B_1 \sqcap (B_2 \sqcup B_3) \Rightarrow^+ \checkmark$. By inversion, $V/B_1 \Rightarrow^+ \checkmark$ and either $V/B_2 \Rightarrow^+ \checkmark$ or $V/B_3 \Rightarrow^+ \checkmark$. It follows either $V/B_1 \sqcap B_2 \Rightarrow^+ \checkmark$ or $V/B_1 \sqcap B_3 \Rightarrow^+ \checkmark$. By the induction hypothesis, $V/A \not\Rightarrow^+$.

Case: $A_1 \sqcap (A_2 \sqcup A_3) \diamond B$ where $A_1 \sqcap A_2 \diamond B$ and $A_1 \sqcap A_3 \diamond B$
By the induction hypotheses, $V/A_1 \sqcap A_2 \not\Rightarrow^+$ and $V/A_1 \sqcap A_3 \not\Rightarrow^+$. Suppose for contradiction $V/A_1 \sqcap (A_2 \sqcup A_3) \Rightarrow^+ \checkmark$ By inversion on this fact, $V/A_1 \Rightarrow^+ \checkmark$ and either $V/A_1 \Rightarrow^+ \checkmark$ or $V/A_2 \Rightarrow^+ \checkmark$. It follows either $V/A_1 \sqcap A_2 \Rightarrow^+ \checkmark$ or $V/A_1 \sqcap A_3 \Rightarrow^+ \checkmark$. Both of these contradict our induction hypotheses.

Case: $l\ A' \diamond k\ B'$ where $l \neq k$
By inversion on $V/k\ B' \Rightarrow^+ \checkmark$, we have $V = k\ V'$ such that $V'/B' \Rightarrow^+ \checkmark$. By straightforward case analysis, $k\ V'/l\ A' \not\Rightarrow^+$. ☐

Lemma C.39 (Distinguishable Patterns). *Suppose* $\Gamma \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ *and* $\Gamma \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$ *where* $A_1 \diamond A_2$. *There does not exist a closed, well-typed value that matches both* p_1 *and* p_2.

Proof. For contradiction, suppose we have v such that $\cdot \vdash v \Rightarrow V$ and $v/p_1 \Rightarrow \sigma_1$ and $v/p_2 \Rightarrow \sigma_2$.

By Lemma C.50 $V/A_1 \Rightarrow^+ \checkmark$ and $V/A_2 \Rightarrow^+ \checkmark$. But this contradicts Lemma C.38, which states that one of these two facts cannot be true. ☐

C.6 Properties of Mergeability

Lemma C.40 (Mergeability). *If* $\hat{V}/A_2^- \Rightarrow^- \checkmark$ *and* $\mathrm{disp}(A_1^-, \hat{V}) \Rightarrow B_1$ *and* $\cdot \vdash A_1^- \sqcap A_2^-$, *then there exists* B_2 *such that* $\mathrm{disp}(A_2^-, \hat{V}) \Rightarrow B_2$.

Proof. By inversion on $\cdot \vdash A_1^- \sqcap A_2^-$ we have $\cdot \vdash A_1^-$, $\cdot \vdash A_2^-$, and $A_1^- \bowtie A_2^-$. We proceed by induction on $A_1^- \bowtie A_2^-$.

Case: $A_1' \to A_1'' \bowtie A_2' \to A_2''$ where $A_1' \diamond A_2'$
By inversion on $\hat{V}/A_2' \to A_2'' \Rightarrow^- \checkmark$, there exists $V = \hat{V}$ such that $V/A_2' \Rightarrow^+ \checkmark$.

By inversion (Lemma C.7) on $\mathrm{disp}(A_1' \to A_1'', V) \Rightarrow B_1$, we have $V \leq A_1'$. From the completeness of the coverage relation (Lemma C.1) it follows $V/A_1' \Rightarrow^+ \checkmark$.

But by Lemma C.38, V cannot match both of the distinguishable types A_1' and A_2', so we have a contradiction.

Case: $A \to A_1' \bowtie A \to A_2'$ where $A_1' \bowtie A_2'$
By inversion (Lemma C.7) on $\mathrm{disp}(A \to A_1', \hat{V}) \Rightarrow B_1$, there exists $V = \hat{V}$ such that $V \leq A$. It follows from Lemma C.25 that there exists A_2'' such that $\mathrm{disp}(A \to A_2', V) \Rightarrow A_2''$.

Case: $\forall \alpha.A_1' \bowtie \forall \alpha.A_2'$ where $A_1' \bowtie A_2'$
By inversion (Lemma C.7) on $\mathrm{disp}(\forall \alpha.A_1', \hat{V}) \Rightarrow B_1$, there exists A such that $\hat{V} = [A]$. It follows from Lemma C.25 that there exists a B_2 such that $\mathrm{disp}(\forall \alpha.A_2', [A]) \Rightarrow B_2$

Case: $A_1' \sqcap A_1'' \bowtie A_2^-$ where $A_1' \bowtie A_2^-$ and $A_1'' \bowtie A_2^-$ and both A_1' and A_1'' are negative
By inversion (Lemma C.16) on our assumption $\text{disp}(A_1' \sqcap A_1'', \hat{V}) \Rightarrow B_1$, there exists C such that either:

- $\text{disp}(A_1', \hat{V}) \Rightarrow C$ or
- $\text{disp}(A_1'', \hat{V}) \Rightarrow C$

In both cases, we have a B_2 from the induction hypothesis such that $\text{disp}(A_2^-, \hat{V}) \Rightarrow B_2$, which is what we needed.

Case: $A_1^- \bowtie A_2' \sqcap A_2''$ where $A_1^- \bowtie A_2'$ and $A_1^- \bowtie A_2''$ and both A_2' and A_2'' are negative

By inversion on $\hat{V}/A_2' \sqcap A_2'' \Rightarrow^- \checkmark$, either $\hat{V}/A_2' \Rightarrow^- \checkmark$ or $\hat{V}/A_2'' \Rightarrow^- \checkmark$. In each of these cases, from the induction hypothesis we have B_2' or B_2'' respectively where $\text{disp}(A_2', \hat{V}) \Rightarrow B_2'$ or $\text{disp}(A_2'', \hat{V}) \Rightarrow B_2''$

From Lemma C.25 we find that there exists B_2 such that $\text{disp}(A_2' \sqcap A_2'', \hat{V}) \Rightarrow B_2$.

Case: $A_1' \sqcup A_1'' \bowtie A_2^-$ where $A_1' \bowtie A_2^-$ and $A_1'' \bowtie A_2^-$ and both A_1' and A_1'' are negative

By inversion on $\cdot \vdash A_1' \sqcup A_1''$, we have $\cdot \vdash A_1'$ and $\cdot \vdash A_1''$.

By Lemma C.7 on $\text{disp}(A_1' \sqcup A_1'', \hat{V}) \Rightarrow B_1$, there exists B_1' and B_1'' such that

- $\text{disp}(A_1', \hat{V}) \Rightarrow B_1'$ and
- $\text{disp}(A_1'', \hat{V}) \Rightarrow B_1''$.

From the induction hypothesis, there exists B_2 such that $\text{disp}(A_2^-, \hat{V}) \Rightarrow B_2$.

Case: $A_1^- \bowtie A_2' \sqcup A_2''$ where $A_1^- \bowtie A_2'$ and $A_1^- \bowtie A_2''$ and both A_2' and A_2'' are negative

By inversion on $\cdot \vdash A_2' \sqcup A_2''$, we have $\cdot \vdash A_2'$ and $\cdot \vdash A_2''$.

By inversion on $\hat{V}/A_2' \sqcup A_2'' \Rightarrow^- \checkmark$ we have $\hat{V}/A_2' \Rightarrow^- \checkmark$ and $\hat{V}/A_2'' \Rightarrow^- \checkmark$.

From the induction hypothesis, we have B_2' and B_2'' where $\text{disp}(A_2', \hat{V}) \Rightarrow B_2'$ and $\text{disp}(A_2'', \hat{V}) \Rightarrow B_2''$ Applying Lemma C.25, there exists B_2 such that $\text{disp}(A_2' \sqcup A_2'', \hat{V}) \Rightarrow B_2$.

Case: $\top \bowtie A_2^-$
By Lemma C.7, $\text{disp}(\top, \hat{V}) \Rightarrow B_1$ is a contradiction.

Case: $A_1^- \bowtie \top$
By inversion, $\hat{V}/\top \Rightarrow^- \checkmark$ is a contradiction.

Case: $A_1' \to A_1'' \bowtie \forall\alpha.A_2'$
By inversion on $\hat{V}/\forall\alpha.A_2' \Rightarrow^- \checkmark$, we have A_2'' such that $\hat{V} = [A_2'']$. But by Lemma C.7 on $\text{disp}(A_1' \to A_1'', \hat{V}) \Rightarrow B_1$, we have V such that $\hat{V} = V$. This is a contradiction.

Case: $\forall\alpha.A_1' \bowtie A_2' \to A_2''$
By inversion on $\hat{V}/A_2' \to A_2'' \Rightarrow^- \checkmark$, we have V such that $\hat{V} = V$. But by Lemma C.7 on $\text{disp}(\forall\alpha.A_1', \hat{V}) \Rightarrow B_1$, we have A_1'' such that $\hat{V} = [A_1'']$. This is a contradiction. □

Lemma C.41. *If* $\cdot \vdash \hat{p} \Rightarrow A_2^- \hookrightarrow B_2 \dashv \Delta$ *and* $F/\hat{p} \Rightarrow \sigma_2 \hookrightarrow B_2$ *and* $\cdot; A_1^- \vdash F \Rightarrow B_1'$ *and* $A_1^- \bowtie A_2^-$ *and* $\cdot \vdash A_1^-$ *and* $\cdot \vdash A_2^-$, *then there exists* B_2' *such that* $\cdot; A_2^- \vdash F \Rightarrow B_2'$.

Proof. Inverting $\cdot; A_1^- \vdash F \Rightarrow B_1'$, we have that $\cdot \vdash F \Rightarrow \hat{V}$ and $\text{disp}(A_1^-, \hat{V}) \Rightarrow B_1'$.

From Lemma C.51 together with our assumptions that $\cdot \vdash \hat{p} \Rightarrow A_2^- \hookrightarrow B_2 \dashv \Delta$ and $F/\hat{p} \Rightarrow \sigma_2 \hookrightarrow B_2$, we have $\hat{V}/A_2^- \Rightarrow^- \checkmark$ Finally, Lemma C.40 gives us B_2' such that $\cdot; A_2^- \vdash F \Rightarrow B_2'$. □

Lemma C.42. *Suppose* $\cdot; A_1^- \sqcap \ldots \sqcap A_n^- \vdash F \Rightarrow B$ *and* $\cdot \vdash A_1^- \sqcap \ldots \sqcap A_n^-$.

Consider any i *such that* $1 \le i \le n$. *If* $\cdot \vdash \hat{p}_i \Rightarrow A_i^- \hookrightarrow B_i \dashv \Delta_i$ *and* $F/\hat{p}_i \Rightarrow \sigma_i \hookrightarrow B_i$ *then there exists* B_i' *such that* $\cdot; A_i^- \vdash F \Rightarrow B_i'$.

Proof. Applying Lemma C.17, we have at least one B'_j such that $\cdot; A_j^- \vdash F \Rightarrow B'_j$. If $i = j$ then this is precisely what we need to show.

For any $i \neq j$ such that $\cdot \vdash \hat{p}_i \Rightarrow A_i^- \hookrightarrow B_i \dashv \Delta_i$ and $F/\hat{p}_i \Rightarrow \sigma_i \hookrightarrow B_i$, we apply Lemma C.41 to obtain B'_i such that $\cdot; A_i^- \vdash F \Rightarrow B'_i$. □

C.7 Properties of Pattern Matching

Lemma C.43.

(1) *If $v/(x : A) \Rightarrow \sigma$ then $\sigma = [x \mapsto v]$.*
(2) *If $F/(* \hookrightarrow A^-) \Rightarrow \sigma \hookrightarrow B$ then $\sigma = [* \mapsto F]$.*

Proof. The proof of part (1) proceeds by induction on the type A.

Case: A is negative
We have immediately that $\sigma = [x \mapsto v]$.

Case: $A = c\,A'$ where $v/c\,(x : A') \Rightarrow [x \mapsto v']$ and $\sigma = [x \mapsto c\,v']$
By inversion on $v/c\,(x : A') \Rightarrow [x \mapsto v']$, we have $v = c\,v'$. Thus, $\sigma = [x \mapsto v]$.

Case: $A = A_1 \sqcup A_2$ where $v/((x : A_1)|(x : A_2)) \Rightarrow \sigma$
By inversion on $v/((x : A_1)|(x : A_2)) \Rightarrow \sigma$, we have either

- $v/(x : A_1) \Rightarrow \sigma$ or
- $v/(x : A_2) \Rightarrow \sigma$.

In both cases, we have from the induction hypothesis that $\sigma = [x \mapsto v]$.

Case: $A = A_1 \sqcap A_2$ where $v/((x : A_1)\&(x : A_2)) \Rightarrow \sigma$
By inversion on $v/((x : A_1)\&(x : A_2)) \Rightarrow \sigma$, we have

- $v/(x : A_1) \Rightarrow \sigma_1$,
- $v/(x : A_2) \Rightarrow \sigma_2$, and
- $\sigma = \sigma_1 \sqcap \sigma_2$

Applying the induction hypotheses for A_1 and A_2, it follows $\sigma = \sigma_1 = \sigma_2 = [x \mapsto v]$.

The proof of part (2) proceeds by induction on the type A.

Case: $A^- = A_1 \to A_2$ where $F/((x : A_1) \hookrightarrow A_2) \Rightarrow [x \mapsto v] \hookrightarrow B$ and $\sigma = [* \mapsto [\cdot]\,v]$
By inversion on $F/((x : A_1) \hookrightarrow A_2) \Rightarrow [x \mapsto v] \hookrightarrow B$, we have $F = [\cdot]\,v$. Thus, $\sigma = [* \mapsto F]$.

Case: $A^- = \forall\,\alpha.B'$ where $F/(\alpha \hookrightarrow B) \Rightarrow [\alpha \mapsto A'] \hookrightarrow B$ and $\sigma = [* \mapsto [\cdot][A']]$
By inversion on $F/(\alpha \hookrightarrow B') \Rightarrow [\alpha \mapsto A'] \hookrightarrow B$, we have $F = [\cdot][A']$. Thus, $\sigma = [* \mapsto F]$.

Case: $A^- = A_1^- \sqcup A_2^-$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_1$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_2$ and $\sigma = \sigma_1 \sqcap \sigma_2$.
By the induction hypotheses for A_1^- and A_2^-, we have $\sigma_1 = \sigma_2 = [* \mapsto F]$. It follows $\sigma = [* \mapsto F]$.

Case: $A^- = A_1^- \sqcap A_2^-$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma \hookrightarrow B$ and $F/(* \hookrightarrow A_2^-) \not\Rightarrow$
By the induction hypothesis, we have $\sigma = [* \mapsto F]$.

Case: $A^- = A_1^- \sqcap A_2^-$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_1$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_2$ and $\sigma = \sigma_1 \sqcup \sigma_2$
By the induction hypothesis, we have $\sigma_1 = \sigma_2 = [* \mapsto F]$. It follows $\sigma = [* \mapsto F]$. □

Lemma C.44 (Consistency). *If $p_1 \sim p_2$ and $v/p_1 \Rightarrow \sigma_1$ and $v/p_2 \Rightarrow \sigma_2$ then $\sigma_1 \sqcap \sigma_2$ is defined.*

Proof. Proceed by induction on $p_1 \sim p_2$.

Case: $(x : A) \sim (x : B)$
By Lemma C.43, $\sigma_1 = \sigma_2 = [x \mapsto v]$. Thus, $\sigma_1 \sqcap \sigma_2 = [x \mapsto v]$.

Case: $(x : A) \sim (y : B)$ where $x \neq y$
By Lemma C.43, $\sigma_1 = [x \mapsto v]$ and $\sigma_2 = [y \mapsto v]$. It follows $\sigma_1 \sqcap \sigma_2 = [x \mapsto v][y \mapsto v]$.

Case: $c\ p_1' \sim c\ p_2'$ where $p_1' \sim p_2'$
By inversion of the value matching definition, there exists v' such that $v = c\ v'$ and $v'/p_1' \Rightarrow \sigma_1$ and $v'/p_2' \Rightarrow \sigma_2$. By the induction hypothesis, $\sigma_1 \sqcap \sigma_2$ is defined.
Case: $p_1'|p_1'' \sim p_2$ where $p_1' \sim p_2$ and $p_1'' \sim p_2$
By inversion of the value matching definition, either $v/p_1' \Rightarrow \sigma_1$ or $v/p_1'' \Rightarrow \sigma_1$. In both cases, the induction hypothesis gives us that $\sigma_1 \sqcap \sigma_2$ is defined.
Case: $p_1'\&p_1'' \sim p_2$ where $p_1' \sim p_2$ and $p_1'' \sim p_2$
By inversion of the value matching definition, we have

- $v/p_1' \Rightarrow \sigma_1'$,
- $v/p_1'' \Rightarrow \sigma_1''$, and
- $\sigma_1 = \sigma_1' \sqcap \sigma_1''$.

The induction hypotheses give us that $\sigma_1' \sqcap \sigma_2$ and $\sigma_1'' \sqcap \sigma_2$ are defined. It is a consequence of the definition of $\sqcap$ that $(\sigma_1' \sqcap \sigma_1'') \sqcap \sigma_2$ is then well-defined. □

Lemma C.45. *If $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_1$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_2$ then $\sigma_1 \sqcup \sigma_2$ is defined.*

Proof. This is a direct consequence of Lemma C.43. □

Lemma C.46 (Inversion of Value Pattern Typing). *Suppose $\cdot \vdash p \Rightarrow A \dashv \Delta$.*

(1) *If $A = l\ A'$ and v' matches all p' for which there exists some Δ' such that $\cdot \vdash p' \Rightarrow A' \dashv \Delta'$ then $l\ v'$ matches p.*
(2) *Assume $A = A_1' \sqcup A_2'$ and either:*
 - *v matches all p_1' such that $\cdot \vdash p_1' \Rightarrow A_1' \dashv \Delta'$, or*
 - *v matches all p_2' such that $\cdot \vdash p_2' \Rightarrow A_2' \dashv \Delta'$.*

 Then v matches p.
(3) *If $A = A_1' \sqcap A_2'$ and v matches all p_1' and p_2' such that $\cdot \vdash p_1' \Rightarrow A_1' \dashv \Delta_1'$ and $\cdot \vdash p_2' \Rightarrow A_2' \dashv \Delta_2'$ then v matches p.*
(4) *If A is negative then every v matches p.*

Proof.

(1) By case analysis on the derivation of p, either $p = l\ p'$ where $\cdot \vdash p' \Rightarrow A' \dashv \Delta$ or $p = (x : l\ A')$.
 In the former subcase, our premise tells us that v' matches p'. It follows $l\ v'$ matches $l\ p'$.
 The latter subcase reduces to the former, since it suffices to show v matches $l\ (x : A')$.
(2) By case analysis on the derivation of p, either $p = p_1'|p_2'$ or $p = (x : A_1' \sqcup A_2')$.
 In the former subcase, we have Δ_1' and Δ_2' such that $\cdot \vdash p_1' \Rightarrow A_1' \dashv \Delta_1'$ and $\cdot \vdash p_2' \Rightarrow A_2' \dashv \Delta_2'$ where $\Delta = \Delta_1' \sqcup \Delta_2'$. From our premise, v matches either p_1' or p_2'. It follows that v matches $p_1'|p_2'$.
 The latter subcase reduces to the former, since it suffices to show v matches $(x : A_1')|(x : A_2')$.
(3) By case analysis on the derivation of p, either $p = p_1'\&p_2'$ or $p = (x : A_1' \sqcap A_2')$.
 In the former subcase, we have Δ_1' and Δ_2' such that $\cdot \vdash p_1' \Rightarrow A_1' \dashv \Delta_1'$ and $\cdot \vdash p_2' \Rightarrow A_2' \dashv \Delta_2'$ and $p_1' \sim p_2'$. From our premise, there exists σ_1 and σ_2 such that $v/p_1' \Rightarrow \sigma_1$ and $v/p_2' \Rightarrow \sigma_2$. To show v matches $p_1'\&p_2'$, it remains to be proven that $\sigma_1 \sqcap \sigma_2$ is defined. This follows from Lemma C.44.
 The latter subcase reduces to the former, since it suffices to show v matches $(x : A_1')\&(x : A_2')$.
(4) By case analysis on the derivation of p, we have $p = (x : A)$. Since A is negative, it follows that v matches $(x : A)$.

□

Lemma C.47 (Inversion of Elimination Pattern Typing). *Suppose* $\cdot \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta$.

(1) *If* $A = A_1 \rightarrow A_2$ *and* v *matches all* p *for which there exists* Δ' *such that* $\cdot \vdash p \Rightarrow A_1 \dashv \Delta'$ *then* $[\cdot]\, v$ *matches* $\hat{p}$.
(2) *If* $A = \forall\, \alpha.A'$ *then* $[\cdot]\,[A'']$ *matches* $\hat{p}$.
(3) *If* $A = A_1 \sqcup A_2$ *and* F *matches all* $\hat{p}_1$ *and* $\hat{p}_2$ *such that* $\cdot \vdash \hat{p}_1 \Rightarrow A_1 \hookrightarrow B_1 \dashv \cdot$ *and* $\cdot \vdash \hat{p}_2 \Rightarrow A_2 \hookrightarrow B_2 \dashv \cdot$ *then* F *matches* $\hat{p}$.
(4) *If* $A = A_1 \sqcap A_2$ *and* F *matches some* $\hat{p}_i$ *such that* $i \in 1, 2$ *and* $\cdot \vdash \hat{p}_1 \Rightarrow A_i \hookrightarrow B_i \dashv \cdot$ *and then* F *matches* $\hat{p}$.

Proof.

(1) By case analysis on the derivation of $\hat{p}$, either $\hat{p} = p \hookrightarrow A_2$ where $\cdot \vdash p \Rightarrow A_1 \dashv \Delta$ or $\hat{p} = (* \hookrightarrow (A_1 \rightarrow A_2))$.
In the former subcase, our premise tells us v matches p. It follows $[\cdot]\, v$ matches $p \hookrightarrow A_2$.
The latter subcase reduces to the former, since it suffices to show $[\cdot]\, v$ matches $(x : A_1) \hookrightarrow A_2$.
(2) By case analysis on the derivation of $\hat{p}$, either $\hat{p} = \alpha \hookrightarrow A'$ where $\alpha \vdash A'$ or $\hat{p} = (* \hookrightarrow \forall\, \alpha.A')$.
In the former subcase, $[\cdot]\,[A'']$ immediately matches $\alpha \hookrightarrow A'$.
The latter subcase reduces to the former, since it suffices to show $[\cdot]\,[A'']$ matches $\alpha \hookrightarrow A'$.
(3) By case analysis on the derivation of $\hat{p}$, we have $\hat{p} = (* \hookrightarrow A_1 \sqcup A_2)$. From our premise, F matches $* \hookrightarrow A_1$ and $* \hookrightarrow A_2$. An application of Lemma C.43 is needed to ensure that the meet of the results of these two matches exists. It follows that F matches $* \hookrightarrow A_1 \sqcup A_2$.
(4) By case analysis on the derivation of $\hat{p}$, we have $\hat{p} = (* \hookrightarrow A_1 \sqcap A_2)$. From our premise, F matches $* \hookrightarrow A_i$ for some $i \in 1, 2$. An application of Lemma C.43 is needed to ensure that the join of the results of these two matches exists. It follows that F matches $* \hookrightarrow A_1 \sqcap A_2$.

□

Lemma C.48 (Value Pattern Matching Abstraction). *Suppose* $\cdot \vdash v \Rightarrow V$. *If* $V/A \Rightarrow^+ \checkmark$ *then* v *matches all* p *for which there exists some* Δ *such that* $\cdot \vdash p \Rightarrow A \dashv \Delta$.

Proof. Proceed by induction on $V/A \Rightarrow^+ \checkmark$.

Case: $l\; V'/l\; A' \Rightarrow^+ \checkmark$ where $V'/A' \Rightarrow^+ \checkmark$

By canonical forms (Lemma C.19) and inversion on the derivation of v, we have $v = l\; v'$ where $\cdot \vdash v \Rightarrow V'$. By Lemma C.46, to prove that $l\; v'$ matches p, it suffices to show v' matches an arbitrary p' where $\cdot \vdash p' \Rightarrow A' \dashv \Delta'$.

Applying the induction hypothesis to $V'/A' \Rightarrow^+ \checkmark$, we obtain that v' does indeed match p'.

Case: $V/A'_1 \sqcup A'_2 \Rightarrow^+ \checkmark$ where $V/A'_1 \Rightarrow^+ \checkmark$

By Lemma C.46, it suffices to show v matches an arbitrary p'_1 such that $\cdot \vdash p'_1 \Rightarrow A'_1 \dashv \Delta'$.

By the induction hypothesis, v matches p'_1.

Case: $V/A'_1 \sqcap A'_2 \Rightarrow^+ \checkmark$ where $V/A'_1 \Rightarrow^+ \checkmark$ and $V/A'_2 \Rightarrow^+ \checkmark$

By Lemma C.46, it suffices to show v matches both p'_1 and p'_2 such that $\cdot \vdash p'_1 \Rightarrow A'_1 \dashv \Delta'_1$ and $\cdot \vdash p'_2 \Rightarrow A'_2 \dashv \Delta'_2$. Both of these follow directly from the induction hypotheses.

Case: $V/A^- \Rightarrow^+ \checkmark$

Immediate from Lemma C.46. □

Lemma C.49 (Elimination Pattern Matching Abstraction). *Suppose* $\cdot \vdash F \Rightarrow \hat{V}$ *and* $\cdot \vdash \hat{p} \Rightarrow B^- \hookrightarrow C \dashv \Delta$. *If* $\hat{V}/B^- \Rightarrow^- \checkmark$ *then* F *matches* $\hat{p}$.

Proof. Proceed by induction on $\hat{V}/B^- \Rightarrow^- \checkmark$.

Case: $V/B_1 \to B_2 \Rightarrow^- \checkmark$ where $V/B_1 \Rightarrow^+ \checkmark$

By canonical forms and inversion Lemma C.20, $F = [\cdot]\,v$ where $\cdot \vdash v \Rightarrow V$.

By Lemma C.47 to prove that $[\cdot]\,v$ matches $\hat{p}$ it suffices to show v matches an arbitrary p such that $\cdot \vdash p \Rightarrow B_1 \dashv \Delta'$. This follows immediately from Lemma C.48.

Case: $[A_1]/\forall\alpha.B_1 \Rightarrow^- \checkmark$

By canonical forms, $F = [\cdot][A_1]$. By Lemma C.47, $[\cdot][A_1]$ matches $\hat{p}$.

Case: $\hat{V}/B_1^- \sqcup B_2^- \Rightarrow^- \checkmark$ where $\hat{V}/B_1^- \Rightarrow^- \checkmark$ and $\hat{V}/B_2^- \Rightarrow^- \checkmark$

By Lemma C.47, to prove F matches $\hat{p}$ it suffices to show F matches all $\hat{p}_1$ and $\hat{p}_2$ where $\cdot \vdash \hat{p}_1 \Rightarrow B_1^- \hookrightarrow C_1 \dashv \Delta_1$ and $\cdot \vdash \hat{p}_2 \Rightarrow B_2^- \hookrightarrow C_2 \dashv \Delta_2$ and $\Delta = \Delta_1 \sqcup \Delta_2$. The fact F matches $\hat{p}_1$ and $\hat{p}_2$ follows immediately from the induction hypotheses.

Case: $\hat{V}/B_1^- \sqcap B_2^- \Rightarrow^- \checkmark$ where $\hat{V}/B_i^- \Rightarrow^- \checkmark$ for some $i \in 1,2$

By Lemma C.47, to prove F matches $\hat{p}$ it suffices to show F matches all $\hat{p}_i$ where $\cdot \vdash \hat{p}_i \Rightarrow B_i^- \hookrightarrow C_i \dashv \Delta_i$. This follows immediately from the induction hypothesis. □

Lemma C.50. *Suppose* $\cdot \vdash v \Rightarrow V$ *and* $\cdot \vdash p \Rightarrow A \dashv \Delta$. *If* v *matches* p *then* $V/A \Rightarrow^+ \checkmark$.

Proof. Proceed by induction on $v/p \Rightarrow \sigma$.

Case: $l\,v'/l\,p' \Rightarrow \sigma$ where $v'/p' \Rightarrow \sigma$

By inversion of the typing relation, $\cdot \vdash v' \Rightarrow V'$ where $V = l\,V'$ and $\cdot \vdash p' \Rightarrow A' \dashv \Delta$ where $A = l\,A'$. By the induction hypothesis, $V'/A' \Rightarrow^+ \checkmark$. It follows $l\,V'/l\,A' \Rightarrow^+ \checkmark$.

Case: $v/(p_1|p_2) \Rightarrow \sigma$ where $v/p_1 \Rightarrow \sigma$

By inversion of the typing relation, $\cdot \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ and $\cdot \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$ where $A = A_1 \sqcup A_2$. By the induction hypothesis, $V/A_1 \Rightarrow^+ \checkmark$. It follows that $V/A_1 \sqcup A_2 \Rightarrow^+ \checkmark$.

Case: $v/(p_1\&p_2) \Rightarrow \sigma$ where $v/p_1 \Rightarrow \sigma_1$ and $v/p_2 \Rightarrow \sigma_2$

By inversion of the typing relation, $\cdot \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ and $\cdot \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$ where $A = A_1 \sqcap A_2$. By the induction hypothesis, $V/A_1 \Rightarrow^+ \checkmark$ and $V/A_2 \Rightarrow^+ \checkmark$. It follows that $V/A_1 \sqcap A_2 \Rightarrow^+ \checkmark$.

Case: $v/(x : A^-) \Rightarrow \sigma$

By inversion of the typing relation, $A = A^-$. It is immediate that $V/A^- \Rightarrow^+ \checkmark$.

Case: $v/(x : l\,A') \Rightarrow \sigma$ where $v/l\,(x : A') \Rightarrow \sigma'$

By inversion of the typing relation, $A = l\,A'$. According to the typing rules, $\cdot \vdash l\,(x : A') \Rightarrow l\,A' \dashv x : A'$. By the induction hypothesis, $V/l\,A' \Rightarrow^+ \checkmark$.

Case: $v/(x : A_1 \sqcup A_2) \Rightarrow \sigma$ where $v/(x : A_1)|(x : A_2) \Rightarrow \sigma'$

By inversion of the typing relation, $A = A_1 \sqcup A_2$. According to the typing rules, $\cdot \vdash (x : A_1)|(x : A_2) \Rightarrow A_1 \sqcup A_2 \dashv x : A_1 \sqcup A_2$. By the induction hypothesis, $V/A_1 \sqcup A_2 \Rightarrow^+ \checkmark$.

Case: $v/(x : A_1 \sqcap A_2) \Rightarrow \sigma$ where $v/(x : A_1)\&(x : A_2) \Rightarrow \sigma'$

By inversion of the typing relation, $A = A_1 \sqcap A_2$. According to the typing rules, $\cdot \vdash (x : A_1)\&(x : A_2) \Rightarrow A_1 \sqcap A_2 \dashv x : A_1 \sqcap A_2$. By the induction hypothesis, $V/A_1 \sqcap A_2 \Rightarrow^+ \checkmark$. □

Lemma C.51. *Suppose* $\cdot \vdash F \Rightarrow \hat{V}$ *and* $\cdot \vdash \hat{p} \Rightarrow A^- \hookrightarrow B \dashv \Delta$. *If* $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$ *then* $\hat{V}/A^- \Rightarrow^- \checkmark$.

Proof. Proceed by induction on $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$.

Case: $[\cdot]v/(p \hookrightarrow B') \Rightarrow \sigma \hookrightarrow B'$ where $v/p \Rightarrow \sigma$

By inversion on the typing relation, we have $\cdot \vdash v \Rightarrow V$ where $V = \hat{V}$ and $\cdot \vdash p \Rightarrow A_0 \dashv \Delta$ where $A^- = A_0 \to B$.

By Lemma C.50, $V/A_0 \Rightarrow^+ \checkmark$. It follows $V/A_0 \to B \Rightarrow^- \checkmark$.

Case: $[\cdot][A_0]/(\alpha \hookrightarrow B'') \Rightarrow \sigma \hookrightarrow B''[A_0/\alpha]$ where $B' = B''[A_0/\alpha]$

By inversion on the typing relation, we have $[A_0] = \hat{V}$ and $A^- = \forall\alpha.B''$. It is immediate that $[A_0]/\forall\alpha.B'' \Rightarrow^- \checkmark$.

Case: $F/(* \hookrightarrow (A_0 \to B')) \Rightarrow \sigma \hookrightarrow B'$ where $F/((x : A_0) \hookrightarrow B') \Rightarrow \sigma' \hookrightarrow B'$
By inversion of the typing relation, $A^- = A_0 \to B'$. According to the typing rules, $\cdot \vdash ((x : A_0) \hookrightarrow B') \Rightarrow A_0 \to B' \hookrightarrow B' \dashv (x : A_0)$. By the induction hypothesis, $\hat{V}/A_0 \to B' \Rightarrow^- \checkmark$.

Case: $F/(* \hookrightarrow (\forall \alpha.B'')) \Rightarrow \sigma \hookrightarrow B'$ where $F/(\alpha \hookrightarrow B'') \Rightarrow \sigma' \hookrightarrow B'$
By inversion of the typing relation, $A^- = \forall \alpha.B''$. According to the typing rules, $\cdot \vdash (\alpha \hookrightarrow B'') \Rightarrow \forall \alpha.B'' \hookrightarrow B'' \dashv \alpha$. By the induction hypothesis, $\hat{V}/\forall \alpha.B'' \Rightarrow^- \checkmark$.

Case: $F/(* \hookrightarrow A_1^- \sqcup A_2^-) \Rightarrow \sigma \hookrightarrow A_1^- \sqcup A_2^-$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow A_1^-$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow A_2^-$
By inversion of the typing relation, $A^- = A_1^- \sqcup A_2^-$. According to the typing rules, $\cdot \vdash (* \hookrightarrow A_1^-) \Rightarrow A_1^- \hookrightarrow A_1^- \dashv \cdot$ and $\cdot \vdash (* \hookrightarrow A_2^-) \Rightarrow A_2^- \hookrightarrow A_2^- \dashv \cdot$. By the induction hypothesis, $\hat{V}/A_1^- \Rightarrow^- \checkmark$ and $\hat{V}/A_2^- \Rightarrow^- \checkmark$. It follows $\hat{V}/A_1^- \sqcup A_2^- \Rightarrow^- \checkmark$.

Case: $F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow A_1^- \sqcap A_2^-$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow A_1^-$
By inversion of the typing relation, $A^- = A_1^- \sqcap A_2^-$. According to the typing rules, $\cdot \vdash (* \hookrightarrow A_1^-) \Rightarrow A_1^- \hookrightarrow A_1^- \dashv \cdot$. By the induction hypothesis, $\hat{V}/A_1^- \Rightarrow^- \checkmark$. It follows $\hat{V}/A_1^- \sqcap A_2^- \Rightarrow^- \checkmark$. ◻

Lemma C.52 (Inversion of Elimination Matching).

(1) *If $\Gamma \vdash \hat{p} \Rightarrow A \to A' \hookrightarrow B \dashv \Delta$ and $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$ then $A' = B'$ and there exists p together with a value v that matches it such that $F = [\cdot]\, v$ and $\Gamma \vdash p \Rightarrow A \dashv \Delta'$.*
(2) *if $\Gamma \vdash \hat{p} \Rightarrow \forall \alpha.A \hookrightarrow B \dashv \Delta$ and $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$ then there exists A' such that $A[A'/\alpha] = B'$ and $F = [\cdot][A']$.*
(3) *If $\Gamma \vdash \hat{p} \Rightarrow A_1 \sqcap A_2 \hookrightarrow B \dashv \Delta$ and $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$ then $\hat{p} = * \hookrightarrow A_1 \sqcap A_2$ and either*
   - *$F/* \hookrightarrow A_1 \Rightarrow \sigma \hookrightarrow B'$ or*
   - *$F/* \hookrightarrow A_2 \Rightarrow \sigma \hookrightarrow B'$ or*
   - *$F/* \hookrightarrow A_1 \Rightarrow \sigma_1 \hookrightarrow B_1'$ and $F/* \hookrightarrow A_2 \Rightarrow \sigma_2 \hookrightarrow B_2'$ and $B' = B_1' \sqcap B_2'$*
(4) *If $\Gamma \vdash \hat{p} \Rightarrow A_1 \sqcup A_2 \hookrightarrow B \dashv \Delta$ and $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$ then there exists B_1', B_2', σ_1, and σ such that $\hat{p} = * \hookrightarrow A_1 \sqcup A_2$ and $F/* \hookrightarrow A_1 \Rightarrow \sigma_1 \hookrightarrow B_1'$ and $F/* \hookrightarrow A_2 \Rightarrow \sigma_2 \hookrightarrow B_2'$ and $B' = B_1' \sqcap B_2'$.*
(5) *If $\Gamma \vdash \hat{p} \Rightarrow \top \hookrightarrow B \dashv \Delta$ then there are no F, σ, and B' such that $F/\hat{p} \Rightarrow \sigma \hookrightarrow B'$.*

Proof.

(1) By case analysis on $\Gamma \vdash \hat{p} \Rightarrow A \to A' \hookrightarrow B \dashv \Delta$, we see that $\hat{p}$ either has the shape $p \hookrightarrow A'$ where $\Gamma \vdash p \Rightarrow A \dashv \Delta$ or the shape $* \hookrightarrow A \to A'$. The latter case reduces to the former by inverting the definition of elimination pattern matching.
   By inversion of $F/p \hookrightarrow A' \Rightarrow \sigma \hookrightarrow B'$ we find $A' = B'$ and $F = [\cdot]\, v$ such that v matches p.
(2) By case analysis on $\Gamma \vdash \hat{p} \Rightarrow \forall \alpha.A \hookrightarrow B \dashv \Delta$, we see that $\hat{p}$ either has the shape $\alpha \hookrightarrow A$ or the shape $* \hookrightarrow \forall \alpha.A$. The latter case reduces to the former by inverting the definition of elimination pattern matching.
   By inversion of $F/\alpha \hookrightarrow A \Rightarrow \sigma \hookrightarrow B'$, we find $F = [\cdot][A']$ and $B' = A[A'/\alpha]$.
(3) By case analysis on $\Gamma \vdash \hat{p} \Rightarrow A_1 \sqcap A_2 \hookrightarrow B \dashv \Delta$, we see that $\hat{p}$ has the shape $* \hookrightarrow A_1 \sqcap A_2$.
   By inversion of $F/(* \hookrightarrow A_1 \sqcap A_2) \Rightarrow \sigma \hookrightarrow B'$, either:
   - $F/(* \hookrightarrow A_1) \Rightarrow \sigma \hookrightarrow B'$ and $F/(* \hookrightarrow A_2) \not\Rightarrow$,
   - $F/(* \hookrightarrow A_1) \not\Rightarrow$ and $F/(* \hookrightarrow A_2) \Rightarrow \sigma \hookrightarrow B'$, or
   - $F/(* \hookrightarrow A_1) \Rightarrow \sigma_1 \hookrightarrow B_1'$ and $F/(* \hookrightarrow A_2) \Rightarrow \sigma_2 \hookrightarrow B_2'$ where $\sigma = \sigma_1 \sqcup \sigma_2$ and $B' = B_1' \sqcap B_2'$.
(4) By case analysis on $\Gamma \vdash \hat{p} \Rightarrow A_1 \sqcup A_2 \hookrightarrow B \dashv \Delta$, we see that $\hat{p}$ has the shape $* \hookrightarrow A_1 \sqcup A_2$.
   By inversion of $F/(* \hookrightarrow A_1 \sqcup A_2) \Rightarrow \sigma \hookrightarrow B'$, we find $F/(* \hookrightarrow A_1) \Rightarrow \sigma_1 \hookrightarrow B_1'$ and $F/(* \hookrightarrow A_2) \Rightarrow \sigma_2 \hookrightarrow B_2'$ and where $\sigma = \sigma_1 \sqcap \sigma_2$ and $B' = B_1' \sqcup B_2'$.

(5) By case analysis on $\Gamma \vdash \hat{p} \Rightarrow \top \hookrightarrow B \dashv \Delta$, we see that $\hat{p}$ has the shape $* \hookrightarrow \top$. Case analysis on the definition of elimination pattern matching reveals that no F can match this pattern.

□

C.8 Progress

Lemma C.53 (Value Pattern Coverage). *If* $\Gamma \vdash v \Leftarrow A$ *and* $\Gamma \vdash p \Rightarrow A \dashv \Delta$ *then* v *matches* p.

Proof. By inversion on $\Gamma \vdash v \Leftarrow A$, we have $\Gamma \vdash v \Rightarrow V$ where $V \leq A$. Since value coverage is complete with respect to subtyping (Lemma C.1), we have $V/A \Rightarrow^{+} \checkmark$. Applying Lemma C.48 to this, we have that v matches p. □

Lemma C.54 (Elimination Pattern Coverage). *If* $\cdot; A^{-} \vdash F \Rightarrow B_1$ *and* $\cdot \vdash \hat{p} \Rightarrow A^{-} \hookrightarrow B_2 \dashv \Delta$ *then* F *matches* $\hat{p}$.

Proof. By inversion on $\cdot; A^{-} \vdash F \Rightarrow B_1$ we have $\cdot \vdash F \Rightarrow \hat{V}$ and $\mathrm{disp}(A^{-}, \hat{V}) \Rightarrow B_1$. Since elimination coverage is complete with respect to type-level dispatch (Lemma C.1), we have $\hat{V}/A^{-} \Rightarrow^{-} \checkmark$. Applying Lemma C.49, we have that F matches $\hat{p}$. □

Lemma C.55 (Progress of Application). *If* $\cdot \vdash M \Rightarrow A$ *and* $\cdot; A \vdash F \Rightarrow B$ *then there exists* e *such that* $\mathrm{disp}(M, F) = e$.

Proof. Let $M = \{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\}$. From the typing derivation for M, we have $A = A_1 \sqcap \ldots \sqcap A_n$. The proof proceeds by induction on n.

Case: $n = 0$
In this case, $A = \top$. By Lemma C.19, $\cdot; \top \vdash F \Leftarrow B$ is a contradiction.

Case: $n \geq 1$
From the regularity of typing (Lemma C.18), $\cdot \vdash A$. By the inversion lemma (Lemma C.17), either

- $\cdot; A_1 \vdash F \Rightarrow B_1$ or
- $\cdot; A_2 \sqcap \ldots \sqcap A_n \vdash F \Rightarrow B_2$.

In the former subcase, from the typing derivation for M, we have B_1' and Δ_1 such that $\cdot \vdash \hat{p}_1 \Rightarrow A_1 \hookrightarrow B_1' \dashv \Delta_1$ The coverage lemma (Lemma C.54) then tells us that F matches $\hat{p}_1$. This means $\mathrm{disp}(M, F)$ is defined.

In the latter subcase, from the typing derivation for M, we have $\cdot \vdash \{\hat{p}_2 \mapsto e_2, \ldots, \hat{p}_n \mapsto e_n\} \Rightarrow A_2 \sqcap \ldots \sqcap A_n$. By the induction hypothesis, $\mathrm{disp}(\{\hat{p}_2 \mapsto e_2, \ldots, \hat{p}_n \mapsto e_n\}, F) = e$. It follows that $\mathrm{disp}(M, F)$ is also defined. □

Lemma C.56 (Progress). *If* $\cdot \vdash e \Rightarrow A$ *then* e *is a value or there exists* e' *such that* $e \mapsto e'$.

Proof. By induction on the typing derivation for e.

Case: $\cdot \vdash l\, e_1 \Rightarrow l\, A_1$ where $\cdot \vdash e_1 \Rightarrow A_1$
By the induction hypothesis, either e_1 is a value – in which case $l\, e_1$ is a value – or $e_1 \mapsto e_1'$ – in which case $l\, e_1 \mapsto l\, e_1'$.

Case: $\cdot \vdash e_1\; e_2 \Rightarrow B$ where $\cdot \vdash e_1 \Rightarrow A_1$ and $\cdot \vdash e_2 \Rightarrow A_2$ and $\mathrm{disp}(A_1, A_2) \Rightarrow B$
By the induction hypothesis, either $e_1 \mapsto e_1'$ or e_1 is a value. In the former case, $e_1\; e_2 \mapsto e_1'\; e_2$. In the latter case, let $v_1 = e_1$.

Again applying the induction hypothesis, either $e_2 \mapsto e_2'$ or e_2 is a value. In the former case, $v_1\; e_2 \mapsto v_1\; e_2'$. Otherwise, let $v_2 = e_2$. In this case, we need to show $v_1\; v_2$ takes a step.

Note that A_1 and A_2 are value types and $\cdot; A_1 \vdash [\cdot]\, v_2 \Rightarrow B$. By the canonical forms lemma (Lemma C.19), $v_1 = \lambda M$. By inversion, $\cdot \vdash M \Rightarrow A_1$. From the progress of application lemma (Lemma C.55), it follows there exists $e' = \mathrm{disp}(M, [\cdot]\, v_2)$.

Case: $\cdot \vdash e_1\ [A_2] \ \Rightarrow\ B$ where $\cdot \vdash e_1 \ \Rightarrow\ A_1$ and $\mathrm{disp}(A_1, [A_2]) \Rightarrow B$
By the induction hypothesis, either $e_1 \mapsto e_1'$ or e_1 is a value. In the former case, $e_1\ [A_2] \mapsto e_1'\ [A_2]$. Otherwise, let $v_1 = e_1$. In this case, we need to show $v_1\ [A_2]$ takes a step.

Note that A_1 is a value type and $\cdot; A_1 \vdash [\cdot][A_2] \ \Rightarrow\ B$. By the canonical forms lemma (Lemma C.19), $v_1 = \lambda M$. By inversion, $\cdot \vdash M \ \Rightarrow\ A_1$. From the progress of application lemma (Lemma C.55), it follows there exists $e' = \mathrm{disp}(M, [\cdot][A_2])$.

Case: $\cdot \vdash (e_1 : A_1^-) \shortparallel (e_2 : A_2^-) \ \Rightarrow\ A_1^- \sqcap A_2^-$ where $\cdot \vdash e_1 \ \Leftarrow\ A_1^-$ and $\cdot \vdash e_2 \ \Leftarrow\ A_2^-$ and $A_1^- \bowtie A_2^-$
By the induction hypothesis, either $e_1 \mapsto e_1'$ or e_1 is a value. In the former case,

$$(e_1 : A_1^-) \shortparallel (e_2 : A_2^-) \mapsto (e_1' : A_1^-) \shortparallel (e_2 : A_2^-)$$

In the latter, let $v_1 = e_1$.

By the induction hypothesis, either $e_2 \mapsto e_2'$ or e_2 is a value. In the former case,

$$(v_1 : A_1^-) \shortparallel (e_2 : A_2^-) \mapsto (v_1 : A_1^-) \shortparallel (e_2' : A_2^-)$$

Otherwise, let $v_2 = e_2$. In this case, we need to show $(v_1 : A_1^-) \shortparallel (v_2 : A_2^-)$ takes a step.

This follows from the operational semantics:

$$(v_1 : A_1^-) \shortparallel (v_2 : A_2^-) \mapsto \lambda\{* \hookrightarrow A_1^- \mapsto \lambda M_1, * \hookrightarrow A_2^- \mapsto \lambda M_2\}$$

.

Case: $\cdot \vdash \lambda M \ \Rightarrow\ A$
By definition, λM is a value.

Case: $\cdot \vdash (e : A) \ \Rightarrow\ A$ where $\cdot \vdash e \ \Rightarrow\ B$ and $B \leq A$
By the induction hypothesis, either $e \mapsto e'$ or e is a value. In the former case, $(e : A) \mapsto (e' : A)$. In the latter case, $(e : A) \mapsto e$. □

C.9 Preservation

Lemma C.57 (Preservation of Mergeability). *Suppose* $\Gamma \vdash \hat{p}_1 \ \Rightarrow\ A_1^- \hookrightarrow B_1 \dashv \Delta_1$ *and* $\Gamma \vdash \hat{p}_2 \Rightarrow A_2^- \hookrightarrow B_2 \dashv \Delta_2$ *where* $A_1^- \bowtie A_2^-$.
If $\cdot \vdash F \Rightarrow \hat{V}$ *and* $F/\hat{p}_1 \Rightarrow \sigma_1 \hookrightarrow B_1'$ *and* $F/\hat{p}_2 \Rightarrow \sigma_2 \hookrightarrow B_2'$ *then* $B_1' \bowtie B_2'$

Proof. Proceed by induction on $A_1^- \bowtie A_2^-$.

Case: $A_1' \to A_1'' \bowtie A_2' \to A_2''$ where $A_1' \diamond A_2'$
From inversion on the elimination pattern matching judgments (Lemma C.52), we have $F = [\cdot]v$ where $\Gamma \vdash p_1 \ \Rightarrow\ A_1' \dashv \Delta_1'$ and $\Gamma \vdash p_2 \ \Rightarrow\ A_2' \dashv \Delta_2'$ and v matches both p_1 and p_2. Also by inversion, we have $\hat{V} = V$ where $\cdot \vdash v \ \Rightarrow\ V$.

This is a contradiction, since by Lemma C.39 two patterns for distinguishable types cannot match the same value.

Case: $A_1' \to A_1'' \bowtie A_2' \to A_2''$ where $A_1'' \bowtie A_2''$
From inversion on the elimination pattern matching judgments (Lemma C.52), we have $B_1' = A_1''$ and $B_2' = A_2''$. We assumed $A_1'' \bowtie A_2''$ in this case.

Case: $\forall\alpha.A_1' \bowtie \forall\alpha.A_2'$ where $A_1' \bowtie A_2'$
From inversion on the elimination pattern matching judgments (Lemma C.52), there exists a C such that $B_1' = A_1'[C/\alpha]$ and $B_2' = A_2'[C/\alpha]$.

Since mergeability is closed under substitution (Lemma C.35), $A_1'[C/\alpha] \bowtie A_2'[C/\alpha]$.

Case: $A_1' \sqcap A_1'' \bowtie A_2$ where $A_1' \bowtie A_2$ and $A_1'' \bowtie A_2$
By inversion on the elimination pattern matching judgments (Lemma C.52), $\hat{p}_1 = * \hookrightarrow A_1' \sqcap A_1''$ and either there exists

- σ_1' and C' such that $F/(* \hookrightarrow A_1') \Rightarrow \sigma_1' \hookrightarrow C'$ or
- σ_1'' and C'' such that $F/(* \hookrightarrow A_1'') \Rightarrow \sigma_1'' \hookrightarrow C''$.

In these cases, from the induction hypothesis we have $C' \bowtie B_2'$ or $C'' \bowtie B_2'$ respectively. Additionally, B_1' is either C' or C'' or $C' \sqcap C''$ depending on whether we are in the first case, second case, or both. If we are in both cases (and have both induction hypotheses), then applying the mergeability rules, $C' \sqcap C'' \bowtie B_2'$.

Case: $A_1' \sqcup A_1'' \bowtie A_2$ where $A_1' \bowtie A_2$ and $A_1'' \bowtie A_2$
By inversion on the elimination pattern matching judgment (Lemma C.52), $\hat{p}_1 = * \hookrightarrow A_1' \sqcup A_1''$ and there exists

- σ_1' and C' such that $F/(* \hookrightarrow A_1') \Rightarrow \sigma_1' \hookrightarrow C'$ and
- σ_1'' and C'' such that $F/(* \hookrightarrow A_1'') \Rightarrow \sigma_1'' \hookrightarrow C''$.

where $B_1' = C' \sqcup C''$.

Our induction hypotheses are that $C' \bowtie B_2'$ and $C'' \bowtie B_2'$. Applying the mergeability rules, it follows $C' \sqcup C'' \bowtie B_2'$.

Case: $\top \bowtie A_2$
According to Lemma C.52, our assumption that F matches $\hat{p}_1$, an elimination pattern for $\top$, is a contradiction.

Case: $A_1' \to A_1'' \bowtie \forall \alpha. A_2'$
From inversion on the elimination pattern matching judgments (Lemma C.52), there exists v and C such that $F = [\cdot]\, v = [\cdot][C]$, which is a contradiction.

□

Lemma C.58 (Adequacy of Matching). *If* $\cdot \vdash p \Rightarrow A \dashv \Delta$ *and* $\cdot \vdash v \Leftarrow A$ *and* $v/p \Rightarrow \sigma$ *then* $\cdot \vdash \sigma \Leftarrow \Delta$

Proof. By induction on $v/p \Rightarrow \sigma$.

Case: $l\, v'/l\, p' \Rightarrow \sigma$ where $v'/p' \Rightarrow \sigma$
Inverting the derivation of $l\, p'$, we have $A = l\, A'$ where $\cdot \vdash p' \Rightarrow A' \dashv \Delta$. Applying the inversion lemma (Lemma C.20) to the derivation of v, we have $\cdot \vdash v' \Leftarrow A'$. By induction, $\cdot \vdash \sigma \Leftarrow \Delta$.

Case: $v/p_1|p_2 \Rightarrow \sigma_1$ where $v/p_1 \Rightarrow \sigma_1$
Inverting the derivation of $p_1|p_2$ we have $A = A_1 \sqcup A_2$ and there exist Δ_1 and Δ_2 such that $\Delta = \Delta_1 \sqcup \Delta_2$ and $\cdot \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ and $\cdot \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$.

By the well-formedness of A, we have $A_1 \diamond A_2$. Applying the disjoint union inversion lemma (Lemma C.6), it follows either $\cdot \vdash v \Leftarrow A_1$ or $\cdot \vdash v \Leftarrow A_2$.

If v has type A_1, then by the induction hypothesis, $\cdot \vdash \sigma_1 \Leftarrow \Delta_1$. It immediately follows that $\cdot \vdash \sigma_1 \Leftarrow \Delta_1 \sqcup \Delta_2$.

If v has type A_2, then by Lemma C.53, v should match p_2. But according to Lemma C.39, v cannot match two patterns of the distinguishable types A_1 and A_2.

Case: $v/p_1 \& p_2 \Rightarrow \sigma_1 \sqcap \sigma_2$ where $v/p_1 \Rightarrow \sigma_1$ and $v/p_2 \Rightarrow \sigma_2$
Inverting the derivation of $p_1 \& p_2$ we have $A = A_1 \sqcap A_2$ and $\Delta = \Delta_1 \sqcap \Delta_2$ where $\cdot \vdash p_1 \Rightarrow A_1 \dashv \Delta_1$ and $\cdot \vdash p_2 \Rightarrow A_2 \dashv \Delta_2$. Applying the inversion lemma to the derivation of v, we have both $\cdot \vdash v \Leftarrow A_1$ and $\cdot \vdash v \Leftarrow A_2$.

By the induction hypothesis, we have $\cdot \vdash \sigma_1 \Leftarrow \Delta_1$ and $\cdot \vdash \sigma_2 \Leftarrow \Delta_2$.

Applying Lemma C.28, it follows $\cdot \vdash \sigma_1 \sqcap \sigma_2 \Leftarrow \Delta_1 \sqcap \Delta_2$.

Case: $\lambda M/(x : A^-) \Rightarrow [x \mapsto \lambda M]$
By inversion on the derivation of $x : A^-$, we have $A = A^-$ and $\Delta = x : A^-$. We need to show $\cdot \vdash [x \mapsto \lambda M] \Leftarrow x : A^-$, which is immediate.

Case: $v/(x : l\, A') \Rightarrow [x \mapsto l\, v']$ where $v/l\,(x : A') \Rightarrow [x \mapsto v']$
By inversion on the derivation of $x : l\, A'$, we have $A = l\, A'$ and $\Delta = x : l\, A'$.

From the pattern typing rules, $\cdot \vdash l\ (x : A') \Rightarrow l\ A' \dashv x : A'$. By the induction hypothesis, $\cdot \vdash [x \mapsto v'] \Leftarrow (x : A')$. This implies $\cdot \vdash l\ v' \Leftarrow l\ A'$, so we have what we need.

Case: $v/(x : A_1 \sqcup A_2) \Rightarrow \sigma$ where $v/(x : A_1)|(x : A_2) \Rightarrow \sigma$

By inversion on the derivation of $(x : A_1 \sqcup A_2)$, we have $A = A_1 \sqcup A_2$.

By the induction hypothesis, $\cdot \vdash \sigma \Leftarrow (x : A_1 \sqcup A_2)$. This is what we needed to show.

Case: $v/(x : \top) \Rightarrow [x \mapsto v]$

By inversion on the derivation of $(x : \top)$, we have $A = \top$. We then have $\cdot \vdash v \Leftarrow \top$, so it follows $\cdot \vdash [x \mapsto v] \Leftarrow (x : \top)$.

Case: $v/(x : A_1 \sqcap A_2) \Rightarrow \sigma$ where $v/(x : A_1)\&(x : A_2) \Rightarrow \sigma$

By inversion on the derivation of $(x : A_1 \sqcap A_2)$, we have $A = A_1 \sqcap A_2$.

By the induction hypothesis, $\cdot \vdash \sigma \Leftarrow (x : A_1 \sqcap A_2)$. This is what we needed to show. □

Lemma C.59 (Adequacy of Application). *Suppose*

(1) $\cdot \vdash \hat{p} \Rightarrow A \hookrightarrow B_1 \dashv \Delta$,
(2) $\cdot; A \vdash F \Rightarrow B_2$, *and*
(3) $F/\hat{p} \Rightarrow \sigma \hookrightarrow B_3$.

Then $\cdot; B_1 \vdash \sigma \Leftarrow \Delta \hookrightarrow B_3$ *and* $B_3 \leq B_2$.

Proof. Proceed by induction on $F/\hat{p} \Rightarrow \sigma \hookrightarrow B_3$.

Case: $[\cdot]v/(p \hookrightarrow B_3) \Rightarrow \sigma \hookrightarrow B_3$ where $v/p \Rightarrow \sigma$

By inversion on the derivation of $p \hookrightarrow B_3$, $\cdot \vdash p \Rightarrow A' \dashv \Delta$ and $A = A' \to B_3$ and $B_1 = B_3$.

By inversion of typing (Lemma C.20) on $\cdot; A' \to B_3 \vdash [\cdot]v \Rightarrow B_2$, we have $\cdot \vdash v \Leftarrow A'$ and $B_3 = B_2$. We have thus far shown $B_1 = B_2 = B_3$.

We next apply the adequacy of matching (Lemma C.58) to find $\cdot \vdash \sigma \Leftarrow \Delta$. As the result of value pattern matching, σ does not contain any type variables in its domain. It immediately follows that $\cdot; B_1 \vdash \sigma \Leftarrow \Delta \hookrightarrow B_1$.

Case: $[\cdot][A']/(\alpha \hookrightarrow B_1) \Rightarrow [\alpha \mapsto A'] \hookrightarrow B_1[A'/\alpha]$ where $B_3 = B_1[A'/\alpha]$

By inversion on the derivation of $\alpha \hookrightarrow B_1$, we have $\Delta = \alpha$ and $A = \forall\, \alpha.B_1$.

Applying inversion of typing to $\cdot; \forall\, \alpha.B_1 \vdash [\cdot][A'] \Rightarrow B_2$, we have $B_1[A'/\alpha] = B_2 = B_3$ and $\cdot \vdash A'$. It immediately follows that $\cdot; B_1 \vdash [\alpha \mapsto A'] \Leftarrow \alpha \hookrightarrow B_1[A'/\alpha]$.

Case: $F/(* \hookrightarrow (B_1' \to B_1)) \Rightarrow [* \mapsto [\cdot]v] \hookrightarrow B_1$ where $F/((x : B_1') \hookrightarrow B_1) \Rightarrow [x \mapsto v] \hookrightarrow B_1$

Note $B_3 = B_1$. By inversion on the derivation of $* \hookrightarrow (B_1' \to B_1)$, we have $A = B_1' \to B_1$ and $\Delta = \cdot$.

By the induction hypothesis, $B_2 \leq B_1 = B_3$ and $\cdot; B_1 \vdash [x \mapsto v] \Leftarrow (x : B_1') \hookrightarrow B_1$. By inversion on this derivation, $\cdot \vdash v \Leftarrow B_1'$. It follows $\cdot; B_1' \to B_1 \vdash [* \mapsto [\cdot]v] \Leftarrow \cdot \hookrightarrow B_1$.

Case: $F/(* \hookrightarrow \forall\, \alpha.B_1) \Rightarrow [* \mapsto [\cdot][C]] \hookrightarrow B_1[C/\alpha]$ where $F/(\alpha \hookrightarrow B_1) \Rightarrow [\alpha \mapsto C] \hookrightarrow B_1[C/\alpha]$

Note $B_3 = B_1[C/\alpha]$. By inversion on the derivation of $* \hookrightarrow \forall\, \alpha.B$, $A = \forall\, \alpha.B_1$ and $\Delta = \cdot$.

By the induction hypothesis, $B_2 \leq B_1[C/\alpha] = B_3$ and $\cdot; B_1 \vdash [\alpha \mapsto C] \Leftarrow \alpha \hookrightarrow B_1[C/\alpha]$. It follows $\cdot; \forall\, \alpha.B_1 \vdash [* \mapsto [\cdot][C]] \Leftarrow \cdot \hookrightarrow B_1[C/\alpha]$.

Case: $F/(* \hookrightarrow \bot) \Rightarrow [* \mapsto F] \hookrightarrow \bot$

By inversion on the derivation of $* \hookrightarrow \bot$, $A = B_1 = B_3 = \bot$.

Note $\cdot; \bot \vdash F \Rightarrow B_2$. By Lemma C.20, $B_2 \equiv \bot = B_3$.

It follows $\cdot; \bot \vdash [* \mapsto F] \Leftarrow \cdot \hookrightarrow \bot$.

Case: $F/(* \hookrightarrow A_1^- \sqcup A_2^-) \Rightarrow \sigma \hookrightarrow B_3' \sqcup B_3''$
where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_3'$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_3''$ and $\sigma = \sigma_1 \sqcap \sigma_2$

By inversion on the derivation of $* \hookrightarrow A_1^- \sqcup A_2^-$, $A = B_1 = A_1^- \sqcup A_2^-$ and $\Delta = \cdot$.

Note $\cdot; A_1^- \sqcup A_2^- \vdash F \Rightarrow B_2$. By inversion on this (Lemma C.20), there exist B_2' and B_2'' such that $\cdot; A_1^- \vdash F \Rightarrow B_2'$ and $\cdot; A_2^- \vdash F \Rightarrow B_2''$ where $B_2' \leq B_2$ and $B_2'' \leq B_2$.

By the induction hypothesis, $\cdot; A_1^- \vdash \sigma_1 \Leftarrow \cdot \hookrightarrow B_3'$ and $B_3' \leq B_2'$ and $\cdot; A_2^- \vdash \sigma_2 \Leftarrow \cdot \hookrightarrow B_3''$ and $B_3'' \leq B_2''$.

It follows $B_3 = B_3' \sqcup B_3'' \leq B_2' \sqcup B_2'' \leq B_2$. Additionally, by Lemma C.28, $\cdot; A_1^- \sqcup A_2^- \vdash \sigma \Leftarrow \cdot \hookrightarrow B_3' \sqcup B_3''$.

Case: $F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow B_3$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma \hookrightarrow B_3$ and $F/(* \hookrightarrow A_2^-) \not\Rightarrow$

By inversion on the derivation of $* \hookrightarrow A_1^- \sqcap A_2^-$, $A = B_1 = A_1^- \sqcap A_2^-$ and $\cdot \vdash A_1^- \sqcap A_2^-$.

Inverting the well-formedness of $A_1^- \sqcap A_2^-$ yields $A_1^- \bowtie A_2^-$ in addition to the well-formedness of A_1^- and A_2^-.

Note that there cannot exist a B_2'' such that $\cdot; A_2^- \vdash F \Rightarrow B_2''$ because according to Lemma C.55 this would mean that F matches $* \hookrightarrow A_2^-$, which we have assumed to be false. Lemma C.17 on $\cdot; A_1^- \sqcap A_2^- \vdash F \Rightarrow B_2$ then tells us that there exists B_2' such that $\cdot; A_1^- \vdash F \Rightarrow B_2'$ and $B_2' \leq B_2$.

By the induction hypothesis we have $\cdot; A_1^- \vdash \sigma \Leftarrow \cdot \hookrightarrow B_3$ and $B_3 \leq B_2'$. By transitivity, $B_3 \leq B_2$.

It follows from monotonicity (Lemma C.10) that $\cdot; A_1^- \sqcap A_2^- \vdash \sigma \Leftarrow \cdot \hookrightarrow B_3$.

Case: $F/(* \hookrightarrow A_1^- \sqcap A_2^-) \Rightarrow \sigma \hookrightarrow B_3' \sqcap B_3''$ where $F/(* \hookrightarrow A_1^-) \Rightarrow \sigma_1 \hookrightarrow B_3'$ and $F/(* \hookrightarrow A_2^-) \Rightarrow \sigma_2 \hookrightarrow B_3''$ and $\sigma = \sigma_1 \sqcup \sigma_2$ and $B_3 = B_3' \sqcap B_3''$

By inversion on the derivation of $* \hookrightarrow A_1^- \sqcap A_2^-$, $A = B_1 = A_1^- \sqcap A_2^-$ and $\cdot \vdash A_1^- \sqcap A_2^-$.

The well-formedness of $A_1^- \sqcap A_2^-$ implies $A_1^- \bowtie A_2^-$ as well as the well-formedness of A_1^- and A_2^-.

Note $\cdot; A_1^- \sqcap A_2^- \vdash F \Rightarrow B_2$. Applying Lemma C.17 to this, we obtain $\cdot; A_1^- \vdash F \Rightarrow B_2'$ and $\cdot; A_2^- \vdash F \Rightarrow B_2''$ where $B_2' \leq B_2$ and $B_2'' \leq B_2$.

By the induction hypothesis,

- $\cdot; A_1^- \vdash \sigma_1 \Leftarrow \cdot \hookrightarrow B_3'$,
- $B_3' \leq B_2'$,
- $\cdot; A_2^- \vdash \sigma_2 \Leftarrow \cdot \hookrightarrow B_3''$, and
- $B_3'' \leq B_2''$.

It follows from Lemma C.28 that $\cdot; A_1^- \sqcap A_2^- \vdash \sigma \Leftarrow \cdot \hookrightarrow B_3' \sqcap B_3''$. From the subtyping rules, $B_3 = B_3' \sqcap B_3'' \leq B_2' \sqcap B_2'' \leq B_2$. □

Lemma C.60 (Preservation of Application). *If* $\cdot \vdash M \Rightarrow A$ *and* $\cdot; A \vdash F \Leftarrow B$ *then* $\cdot \vdash \mathrm{disp}(M, F) \Leftarrow B$.

Proof. Let $M = \{\hat{p}_1 \mapsto e_1, \ldots, \hat{p}_n \mapsto e_n\}$. Let I be the set of indices i such that F matches $\hat{p}_i$. That is, $i \in I$ when there exists σ_i such that $F/\hat{p}_i \Rightarrow \sigma_i \hookrightarrow B_i$. Then by the definition of apply,

$$\mathrm{disp}(M, F) = \bigsqcup_{i \in I} (\sigma_i(e_i) : B_i)$$

From the typing derivation for M, we have $A = A_1 \sqcap \ldots \sqcap A_n$. By regularity of typing (Lemma C.18), $\cdot \vdash A_1 \sqcap \ldots \sqcap A_n$. Let J be the set of indices j where there exists A_j' such that $\cdot; A_j \vdash F \Rightarrow A_j'$. From Lemma C.37, we get $\bigsqcap_{j \in J} A_j' \leq B$. Thus, at this point, we need only prove

$$\cdot \vdash \bigsqcup_{i \in I} (\sigma_i(e_i) : B_i) \Leftarrow \bigsqcap_{j \in J} A_j'$$

From the typing rules, we have

$$\cdot \vdash \bigsqcup_{i \in I} (\sigma_i(e_i) : B_i) \Rightarrow \bigsqcap_{i \in I} B_i$$

provided that we show $\cdot \vdash \sigma_i(e_i) \Leftarrow B_i$ for each $i \in I$ and $B_1 \bowtie \ldots \bowtie B_n$. In other words, we have three proof obligations remaining:

(1) $\bigsqcap_{i\in I} B_i \leq \bigsqcap_{j\in J} A'_j$
(2) For each $i \in I$ we have $\cdot \vdash \sigma_i(e_i) \ \Leftarrow\ B_i$.
(3) $B_i \bowtie B_j$ for any distinct $i, j \in I$.

To discharge the first, it suffices to show that for every $j \in J$, we have $j \in I$ and $B_j \leq A'_j$. To this end, first recall that for each $j \in J$ we have

$$\cdot; A_j \vdash F \ \Rightarrow\ A'_j \text{ and } \cdot \vdash \hat{p}_j \Rightarrow A_j \hookrightarrow C_j \dashv \Delta_j$$

By coverage of application (Lemma C.55), there exists a σ_j such that $F/\hat{p}_j \Rightarrow \sigma_j \hookrightarrow B_j$. Thus, $J \subseteq I$.

For $j \in J$, we have

- $\cdot \vdash \hat{p}_j \Rightarrow A_j \hookrightarrow C_j \dashv \Delta_j$,
- $\cdot; A_j \vdash F \ \Rightarrow\ A'_j$, and
- $F/\hat{p}_j \Rightarrow \sigma_j \hookrightarrow B_j$.

These enable us to apply the adequacy of application (Lemma C.59) to obtain $B_i \leq A'_i$. This proves the first obligation.

We next need to establish that $I \subseteq J$. Note this claim means that F *dynamically* matching a pattern implies that F *statically* has a particular type. The key here is Lemma C.42.

Recall we have $\cdot; A_1 \sqcap \ldots \sqcap A_n \vdash F \ \Leftarrow\ B$. For each $i \in I$, we also have $F/\hat{p}_i \Rightarrow \sigma_i \hookrightarrow B_i$, so Lemma C.42 gives us A'_i such that $\cdot; A_i \vdash F \ \Rightarrow\ A'_i$. Thus, $i \in J$.

Previously we applied the adequacy lemma for each $j \in J$. Now that we know $I = J$, we can take advantage of its other conclusion: $\cdot; C_i \vdash \sigma_i \Leftarrow \Delta_i \hookrightarrow B_i$ for $i \in I$. By inversion on the derivation of M, we find $\Delta_i \vdash e_i \ \Leftarrow\ C_i$. Now applying the substitution σ_i to the expression e_i using Lemma C.33, we obtain $\cdot \vdash \sigma_i(e_i) \ \Leftarrow\ B_i$ to complete the proof of the second obligation.

The third and final proof obligation follows from the preservation of mergeability (Lemma C.57). □

Lemma C.61 (Preservation). *If $\cdot \vdash e \ \Leftarrow\ A$ and $e \mapsto e'$ then $\cdot \vdash e' \ \Leftarrow\ A$.*

Proof. By induction on $e \mapsto e'$.

Case: $E[e_0] \mapsto E[e'_0]$ where $e_0 \mapsto e'_0$

By inversion using Lemma C.26, $\cdot \vdash e_0 \ \Rightarrow\ A'$. The induction hypothesis gives us $\cdot \vdash e'_0 \ \Leftarrow\ A'$. By the replacement lemma (Lemma C.27), $\cdot \vdash E[e'_0] \ \Leftarrow\ A$.

Case: $(v_1 : A_1^-) \shortparallel (v_2 : A_2^-) \mapsto \lambda\{* \hookrightarrow A_1^- \mapsto v_1, * \hookrightarrow A_2^- \mapsto v_2\}$

By inversion of the typing relation, $A = A_1^- \sqcap A_2^-$ and $A_1^- \bowtie A_2^-$ and $\cdot \vdash v_1 \ \Leftarrow\ A_1$ and $\cdot \vdash v_2 \ \Leftarrow\ A_2$.

Applying the typing rules, it follows $\cdot \vdash \lambda\{* \hookrightarrow A_1^- \mapsto v_1, * \hookrightarrow A_2^- \mapsto v_2\} \ \Leftarrow\ A_1^- \sqcap A_2^-$

Case: $F[v] \mapsto e'$

Inverting the typing derivation of e with Lemma C.20 yields $\cdot \vdash v \ \Rightarrow\ B$ and $\cdot; B \vdash F \ \Leftarrow\ A$. By the canonical forms lemma, v has the form λM. We can now determine from the operational semantics that $e' = \mathrm{disp}(M, F)$. By application preservation (Lemma C.60), $\cdot \vdash \mathrm{disp}(M, F) \ \Leftarrow\ A$.

Case: $(v : B) \mapsto v$

By inversion of the typing relation, $\cdot \vdash v \ \Leftarrow\ B$ where $B \leq A$. It follows $\cdot \vdash v \ \Leftarrow\ A$. □

C.10 Soundness

Theorem C.62 (Type Soundness). *If $\cdot \vdash e \ \Leftarrow\ A$ and $e \mapsto^* e'$ then either e' is a value or e' can take another step.*

Proof. Follows from Lemma C.56 and Lemma C.61. □

D DETERMINISM

Lemma D.1 (Determinism of Pattern Matching).

(1) *If* $\Gamma \vdash p \Rightarrow A \dashv \Delta$ *and* $\Gamma \vdash v \Leftarrow A$ *and* $v/p \Rightarrow \sigma_1$ *and* $v/p \Rightarrow \sigma_2$ *then* $\sigma_1 = \sigma_2$.

(2) *If* $\Gamma \vdash \hat{p} \Rightarrow A \hookrightarrow B \dashv \Delta$ *and* $\Gamma \vdash F \Rightarrow \hat{V}$ *and* $F/\hat{p} \Rightarrow \sigma_1 \hookrightarrow B_1$ *and* $F/\hat{p} \Rightarrow \sigma_2 \hookrightarrow B_2$ *then* $\sigma_1 = \sigma_2$ *and* $B_1 = B_2$.

Proof. The only source of nondeterminism in the definition of pattern matching is when p has the form $p_1|p_2$ where v matches both p_1 and p_2.

By inversion on the typing derivation for $p_1|p_2$, the type of p_1 must be distinguishable from that of p_2. This contradicts Lemma C.38.

Now that we have established value pattern matching is deterministic, it is easy to check that elimination matching is as well. □

Theorem D.2 (Determinism of Evaluation). *Suppose* $\cdot \vdash e \Rightarrow A$. *If* $e \mapsto e_1$ *and* $e \mapsto e_2$ *then* $e_1 = e_2$.

Proof. Follows from Lemma D.1. □