A Type-Directed Operational Semantics For a Calculus with a Merge Operator

HUANG Xuejing
The University of Hong Kong
xjhuang@cs.hku.hk

Bruno C. d. S. Oliveira
The University of Hong Kong
bruno@cs.hku.hk

Abstract

Calculi with disjoint intersection types and a merge operator provide general mechanisms that can subsume various other features. Such calculi can also encode highly dynamic forms of object composition, which capture common programming patterns in dynamically typed languages (such as JavaScript) in a fully statically typed manner. Unfortunately, unlike many other foundational calculi (such as *System F*, *System $F_{<:}$* or *Featherweight Java*), recent calculi with the merge operator lack a (direct) operational semantics with standard and expected properties such as *determinism* and *subject-reduction*. Furthermore the metatheory for such calculi can only account for *terminating programs*, which is a significant restriction in practice.

This paper proposes a *type-directed operational semantics* (TDOS) for $\lambda_i^{:}$: a calculus with *intersection types* and a *merge operator*. The calculus is inspired by two closely related calculi by Dunfield (2014) and Oliveira et al. (2016). Although Dunfield proposes a direct small-step semantics for his calculus, his semantics lacks both *determinism* and *subject-reduction*. Using our TDOS we obtain a direct semantics for $\lambda_i^{:}$ that has both properties. To fully obtain determinism, the $\lambda_i^{:}$ calculus employs a disjointness restriction proposed in Oliveira et al.'s λ_i calculus. As an added benefit the TDOS approach deals with recursion in a straightforward way, unlike λ_i and subsequent calculi where recursion is problematic. To further relate $\lambda_i^{:}$ to the calculi by Dunfield and Oliveira et al. we show two results. Firstly, the semantics of $\lambda_i^{:}$ is sound with respect to Dunfield's small-step semantics. Secondly, we show that the type system of $\lambda_i^{:}$ is complete with respect to the λ_i type system. All results have been fully formalized in the Coq theorem prover.

2012 ACM Subject Classification ...

Keywords and phrases operational semantics, type systems, intersection types

1 Introduction

The *merge operator* was firstly introduced by Reynolds in the Forsythe language [43] over 30 years ago. It has since been studied, refined and used in some language designs by multiple researchers [2, 4, 8, 20, 35, 37]. At its essence the merge operator allows creating values that can have *multiple types* (encoded as *intersection types* [17, 39]). For example, with the merge operator, the following program is valid:

$$\mathit{let}\ x : \mathsf{Int} \,\&\, \mathsf{Bool} \ = \ 1 \,,,\, \mathsf{True}\ \mathit{in}\ (x + 1,\ \mathsf{not}\ x)$$

Here the variable x has two types, expressed by the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool}$. The corresponding value for x is built using the merge operator $(,,)$. Later uses of x, such as the expression $(x + 1,\ \mathsf{not}\ x)$ can use x both as an integer or as a boolean. For this particular example, the result of executing the expression is the pair $(2,\ \mathsf{False})$.

The merge operator adds expressive power to calculi with intersection types. Much work on intersection types has focused on *refinement intersections* [18, 21, 24], which only increase the expressive power of types. In systems with refinement intersections, types can simply be erased during compilation. However, in those systems the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool}$ is invalid since Int and Bool are not refinements of each other. In other systems, including many OO languages with intersection types [22,30,32,40], the type Int & Bool has no inhabitants and the simple program above is inexpressible. The merge operator adds expressiveness to terms and allows constructing values that inhabit the intersection type Int & Bool.

There are various practical applications for the merge operator. One benefit, as Dunfield [20] argues, is that the merge operator and intersection types provide "*general mechanisms that subsume many different features*". This is important because often a new type system feature involves extending the metatheory and implementation, which can be non-trivial. If instead we provide general mechanisms that can encode such features, then adding new features will become a lot easier. Dunfield has illustrated this point by showing that *multi-field records*, *overloading* and forms of *dynamic typing* can all be easily encoded in the presence of the merge operator. More recently, the merge operator has been used in calculi with disjoint intersection types [2, 4, 35] to encode several non-trivial object-oriented features, which enable highly dynamic forms of object composition not available in current mainstream languages such as Scala or Java. These include *first-class traits* [5], *dynamic mixins* [2], and forms of *family polymorphism* [4]. These features allow, for instance, capturing widely used and expressive techniques for object composition used by JavaScript programmers (and programmers in other dynamically typed languages), but in a completely *statically type-safe* manner [2,5]. For example, in the SEDEL language [5], which is based on disjoint intersection types, we can define and use first-class traits such as:

```
// addId takes a trait as an argument, and returns another trait
addId(super : Trait[Person], idNumber : Int) : Trait[Student] =
  trait inherits super ⇒ {  // dynamically inheriting from an unknown person
      def id : Int = idNumber
  }
```

Similarly to classes in JavaScript, first-class traits can be passed as arguments, returned as results, and they can be constructed dynamically (at run-time). In the program above inheritance is encoded as a merge in the core language with disjoint intersection types used by SEDEL.

Despite over 30 years of research, the semantics of the merge operator has proved to be quite elusive. Because of its foundational importance, we would expect a simple and clear *direct* semantics to exist for calculi with a merge operator. After all, this is what we get for other foundational calculi such as the *simply typed lambda calculus*, *System F*, *System* F_ω, the *calculus of constructions*, *System* $F_{<:}$, *Featherweight Java* and others. All these calculi have a simple and elegant *direct operational semantics* (often presented in a small-step style [51]). While for the merge operator there have been efforts in the past to define a direct operational semantics, these efforts have placed severe limitations that disallow many of the applications previously discussed or they lacked important properties. Reynolds [41] was the first to look at this problem, but in his calculus the merge operator is severely limited (for instance a merge of two functions is not possible). Castagna [8] studied another calculus, where only merges of functions are possible. Pierce [37] was the first to briefly consider a calculus with an unrestricted merge operator (called *glue* in his own work). He discussed an extension to $F_\wedge$ with a merge operator but he did not study the dynamic semantics with the extension. Finally, Dunfield [20] goes further and presents a direct operational semantics for a calculus with an unrestricted merge operator. However the problem is that subject-reduction and determinism are lost.

Dunfield also presents an alternative way to give the semantics for a calculus with the merge operator *indirectly by elaboration* to another calculus. This elaboration semantics is type-safe and offers, for instance, a reasonable implementation strategy, and it is also employed in more recent work on the merge operator with disjoint intersection types. However the elaboration semantics has two important drawbacks. Firstly, reasoning about the elaboration semantics is much more complex: to understand the semantics of programs with the merge operator we have to understand the translation and semantics of the target calculus. This complicates informal and formal reasoning. Secondly, a fundamental property in an elaboration semantics is *coherence* [41] (which ensures that the meaning of a program is not ambiguous). All existing calculi with disjoint intersection types prove coherence, but this currently comes at a high price: the calculi and proof techniques employed to prove coherence can only deal with terminating programs. A severe limitation in practice!

This paper presents a *type-directed operational semantics* (TDOS) for $\lambda_i^{;}$: a calculus with intersection types and a merge operator [43]. $\lambda_i^{;}$ is inspired by closely related calculi by Dunfield [20] and Oliveira et al. (λ_i) [35], but addresses two key difficulties in the dynamic semantics of calculi with a merge operator. The first one is the type dependent nature of the merge operator. This difficulty is addressed by using types in the TDOS to guide reduction, which is crucial to prove *subject-reduction*. The second difficulty is that a fully unrestricted merge operator is inherently ambiguous. For instance the merge $1\,,,\,2$ can evaluate to both 1 and 2. Therefore some restriction is still necessary for a deterministic semantics. To fully obtain determinism, the $\lambda_i^{;}$ calculus uses the disjointness restriction that is employed in λ_i and several other calculi using disjoint intersection types, and two important new notions: *typed reduction* and *consistency*. Typed reduction is a reduction relation that can further reduce values under a certain type. This is where types are used in the dynamic semantics. Consistency is an equivalence relation on values, that is key for the determinism result. Determinism in TDOS offers the same guarantee that coherence offers in an elaboration semantics (both properties ensure that the semantics is unambiguous), but it is much simpler to prove. Additionally, the TDOS approach deals with recursion in a straightforward way, unlike λ_i and subsequent calculi where recursion is very problematic for proving coherence.

To further relate $\lambda_i^{;}$ to the calculi by Dunfield and Oliveira et al. we show two results. Firstly, we show that the type system of $\lambda_i^{;}$ is complete with respect to the type system of λ_i. Secondly, the semantics of $\lambda_i^{;}$ is sound with respect to Dunfield's semantics. In our work we use two variants of $\lambda_i^{;}$: one that follows Dunfield's original formulation of subtyping, and another with a more powerful subtyping relation inspired by Bi et al. [4]. The more powerful subtyping relation enables $\lambda_i^{;}$ to account for merges of functions in a natural way, which was awkward in λ_i. For the variant with the extension we also require a minor extension to Dunfield's operational semantics. The two variants of the $\lambda_i^{;}$ calculus and its metatheory have been fully formalized in the Coq theorem prover.

In summary, the contributions of this paper are:

- **The $\lambda_i^{;}$ calculus and its TDOS:** We present a TDOS for $\lambda_i^{;}$: a calculus with intersection types and a merge operator. The semantics of $\lambda_i^{;}$ is both *deterministic* and it has *subject-reduction*.
- **Support for non-terminating programs:** Our new proof methods can deal with recursion, unlike the proof methods used in previous calculi with disjoint intersection types [4, 6], due to limitations of the current proof approaches for coherence.
- **Typed reduction and consistency:** We propose the novel notions of typed reduction and consistency, which are useful to prove determinism and subject-reduction.
- **Relation with other calculus with intersection types:** We relate $\lambda_i^{;}$ with the calculi proposed by Dunfield and Oliveira et al (λ_i). In short all programs that are accepted in λ_i can type-check with our type system, and the semantics of $\lambda_i^{;}$ is sound with respect to Dunfield's semantics.
- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover and they are available online[1].

2 Overview

This section gives an overview of the type-directed operational semantics for $\lambda_i^{;}$. We first introduce Dunfield's untyped semantics [20], and identify its problems: the non-determinism of the semantics

[1] **Note for reviewers:** Due to the anonymous review process the materials can be found in the supplementary materials.

$$
\begin{array}{ll}
\mathit{Type} & A, B ::= \mathsf{Top} \mid A \to B \mid A \,\&\, B \\
\mathit{Expr} & E ::= x \mid \top \mid \lambda x.E \mid E_1\ E_2 \mid \mathbf{fix}\,x.E \mid E_1 \,,, \, E_2 \\
\mathit{Value} & V ::= x \mid \top \mid \lambda x.E \mid V_1 \,,, \, V_2
\end{array}
$$

$\boxed{E \rightsquigarrow E'}$ *(Operational semantics of Dunfield's calculus)*

$$
\textsc{DStep-appl}\ \frac{E_1 \rightsquigarrow E_1'}{E_1\ E_2 \rightsquigarrow E_1'\ E_2}
\qquad
\textsc{DStep-appr}\ \frac{E_2 \rightsquigarrow E_2'}{V_1\ E_2 \rightsquigarrow V_1\ E_2'}
\qquad
\textsc{DStep-beta}\ \frac{}{(\lambda x.\, E)\ V \rightsquigarrow E[x \mapsto V]}
\qquad
\textsc{DStep-fix}\ \frac{}{\mathbf{fix}\, x.\, E \rightsquigarrow E[x \mapsto \mathbf{fix}\, x.\, E]}
$$

$$
\textsc{DStep-mergel}\ \frac{E_1 \rightsquigarrow E_1'}{E_1 \,,, \, E_2 \rightsquigarrow E_1' \,,, \, E_2}
\qquad
\textsc{DStep-merger}\ \frac{E_2 \rightsquigarrow E_2'}{V_1 \,,, \, E_2 \rightsquigarrow V_1 \,,, \, E_2'}
\qquad
\textsc{DStep-unmergel}\ \frac{}{E_1 \,,, \, E_2 \rightsquigarrow E_1}
\qquad
\textsc{DStep-unmerger}\ \frac{}{E_1 \,,, \, E_2 \rightsquigarrow E_2}
$$

$$
\textsc{DStep-split}\ \frac{}{E \rightsquigarrow E \,,, \, E}
$$

Figure 1 The syntax and non-deterministic small-step semantics of Dunfield's calculus

and the lack of subject-reduction. Dunfield's semantics is nonetheless used to guide the design of our own TDOS. We show how the TDOS of $\lambda_i^{;}$ uses type annotations to guide reduction, thus obtaining a deterministic semantics that also has the subject-reduction property.

2.1 Background: Dunfield's Non-Deterministic Semantics

Dunfield studied the semantics of a calculus with intersection types and a merge operator. The most interesting aspect of his calculus is the merge operator, which takes two terms e_1 and e_2, of some types A and B, to create a new term that can behave both as a term of type A and as a term of type B. Intersection types and the merge operator in Dunfield's calculus are quite similar to pair types and pairs. Indeed, a program written with pairs that behaves identically to the program shown in Section 1 is:

$$\mathit{let}\ x : (\mathsf{Int}, \mathsf{Bool}) \ = \ (1, \mathsf{True})\ \mathit{in}\ (\,\mathsf{fst}\ x + 1,\ \mathsf{not}\ (\mathsf{snd}\ x))$$

However while for pairs both the introductions and eliminations are explicit, with the merge operator the eliminations (i.e. projections) are *implicit* and driven by the types of the terms. Dunfield exploits this similarity to give a type-directed elaboration semantics to his calculus. The elaboration transforms merges into pairs, intersection types into pair types and inserts the missing projections.

Syntax. The top of Figure 1 shows the syntax of Dunfield's calculus. Types include a top type Top, function types ($A \to B$) and intersection types (written as $A \,\&\, B$). Most expressions are standard, except the merges ($E_1 \,,, \, E_2$). The calculus also includes a canonical top value $\top$, and allows variables to be values. Note that the original Dunfield's calculus uses a different notation for intersection types ($A \wedge B$), and supports union types ($A \vee B$). Union types are not supported by $\lambda_i^{;}$, since it is based on the λ_i calculus [35] with disjoint intersection types, which does not have unions either. For a better comparison, we adjust the syntax and omit union types in Dunfield's system.

Operational Semantics. The bottom part of Figure 1 presents the reduction rules. The interesting construct is the merge operator, as all other rules not involving the merge operator are standard call-by-value reduction rules. The reduction of a merge construct in Dunfield's calculus is quite flexible: a merge of two expressions (which do not even need to be two values) can step to its left subexpression (by rule DSTEP-UNMERGEL) or the right one (by rule DSTEP-UNMERGER). Any expressions can split into two by rule DSTEP-SPLIT. Therefore, even though the reduction rules may have already reached a value form, it is still possible to step further using rule DSTEP-SPLIT.

Problem 1: No Subject-Reduction. A major problem of this operational semantics is that it does not preserve types. Note that reduction is oblivious of types, so a term can reduce to two other terms with potentially different (and unrelated) types. For instance:

$$1\,,,\,\mathsf{True} \rightsquigarrow 1 \qquad 1\,,,\,\mathsf{True} \rightsquigarrow \mathsf{True}$$

Here the merge of an integer and a boolean is reduced to either the integer (using rule DSTEP-UNMERGEL) or the boolean (using rule DSTEP-UNMERGER). In Dunfield's calculus the term $1\,,,\,\mathsf{True}$ can have multiple types, including Int or Bool or even $\mathsf{Int} \,\&\, \mathsf{Bool}$. As a consequence, the semantics is not type-preserving in general.

What is worse, a well-typed expression can reduce to an expression that is ill-typed:

$$(1\,,,\,\lambda x.\,x\,+\,1)\,2 \rightsquigarrow 1\,2$$

This reduction leads to an ill-typed term (with any type) because we drop the lambda instead of the 1 in the merge.

Problem 2: Non-determinism. Even in the case of type-preserving reductions there can be another problem. Because of the pair of unmerge rules (rule DSTEP-UNMERGEL and rule DSTEP-UNMERGER), the choice between a merge always has two options. This means that a reduced term can lead to two other terms of the same type, but with different meanings. For example:

$$1\,,,\,2 \rightsquigarrow 1 \qquad 1\,,,\,2 \rightsquigarrow 2$$

There is even a third option to reduce a merge with the split rule (rule DSTEP-SPLIT):

$$1\,,,\,2 \rightsquigarrow (1\,,,\,2)\,,,\,(1\,,,\,2)$$

In other words the semantics is non-deterministic.

Note that Dunfield's operational semantics is an overapproximation of the intended behavior. In his work, it is used to provide a soundness result for his elaboration semantics, which is type-safe (but still ambiguous).

2.2 A Type-Driven Semantics for Type Preservation

An essential problem is that the semantics cannot ignore the types if the reduction is meant to be type-preserving. Dunfield notes that "*For type preservation to hold, the operational semantics would need access to the typing derivation*" [20]. To avoid runtime type-checking, we design a type-driven semantics and use type annotations to guide reduction. Therefore our $\lambda_i^{;}$ calculus is explicitly typed, unlike Dunfield's calculus. Nevertheless, it is easy to design source languages that infer some of the type annotations and insert them automatically to create valid $\lambda_i^{;}$ terms as we will see in Section 5. We discuss the main challenges and key ideas of the design of $\lambda_i^{;}$ next.

Type-driven Reduction. Our operational semantics follows a standard call-by-value small-step reduction and it is closely related to Dunfield's semantics. However, type annotations play an important role in the reduction rules and are used to guide reduction. For example, in $\lambda_i^{:}$ we can write explicitly annotated expressions such as $(1\,,,\,\mathsf{True}) : \mathsf{Int}$ and $(1\,,,\,\mathsf{True}) : \mathsf{Bool}$. For those expressions the following reductions are valid:

$$(1\,,,\,\mathsf{True}) : \mathsf{Int} \hookrightarrow 1 \qquad (1\,,,\,\mathsf{True}) : \mathsf{Bool} \hookrightarrow \mathsf{True}$$

In contrast the following reductions are not possible:

$$(1\,,,\,\mathsf{True}) : \mathsf{Bool} \not\hookrightarrow 1 \qquad (1\,,,\,\mathsf{True}) : \mathsf{Int} \not\hookrightarrow \mathsf{True}$$

Note also that in $\lambda_i^{:}$ the meaning of expression $1\,,,\,\mathsf{True}$ without any type annotation can only be a corresponding value $1\,,,\,\mathsf{True}$ that does not drop any information.

Typed Reduction. The crucial component in the operational semantics that enables the use of type information during reduction is an auxiliary *typed reduction* relation $v \hookrightarrow_A v'$ that is used when we want some value to match a type. Typed reduction is where type information from annotations in $\lambda_i^{:}$ "filters" reductions that are invalid due to a type mismatch. Typed reduction takes a value and a type (which can be viewed as inputs), and gives a unique value of that type as output. Note that this process may result in further reduction of the value, unlike many other languages where values can never be further reduced. Typed reduction is used in two places during reduction:

$$\textsc{Step-annov}\quad \frac{v \hookrightarrow_A v'}{v : A \;\hookrightarrow\; v'} \qquad\qquad \textsc{Step-beta}\quad \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\, v \;\hookrightarrow\; (e[x \mapsto v']) : B}$$

The first place where typed reduction is used is in rule Step-annov. When reduction enconters a value with a type annotation A it uses typed reduction to do further reduction depending on the type A. To see typed reduction in action, consider a simple merge of primitive values such as $1\,,,\,\mathsf{True}\,,,\,\text{`}c\text{'}$ with an annotation $\mathsf{Int} \,\&\, \mathsf{Char}$. Using rule Step-annov typed reduction is invoked, resulting in:

$$1\,,,\,\mathsf{True}\,,,\,\text{`}c\text{'} \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Char}} 1\,,,\,\text{`}c\text{'}$$

We could have type-reduced the same value under a similar type but where the two types in the intersection are interchanged:

$$1\,,,\,\mathsf{True}\,,,\,\text{`}c\text{'} \hookrightarrow_{\mathsf{Char}\,\&\,\mathsf{Int}} \text{`}c\text{'}\,,,\,1$$

Both typed reductions are valid and they illustrate the ability of typed reduction to create a value that matches exactly with the shape of the type.

The second place where typed reduction is used is in rule Step-beta. In a function application, the actual argument could be a merge containing more components than what the function expects. One example is $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(1\,,,\,\mathsf{True})$. Since the merge term $(1\,,,\,\mathsf{True})$ provides an integer 1, the redundant components (the True in this case) are useless, and sometimes even harmful. Consider a function $\lambda x.\,(x\,,,\,\mathsf{False})$ with type $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$, applied to $(1\,,,\,\mathsf{True})$. If we performed direct substitution of the argument in the lambda body, this would result in $1\,,,\,\mathsf{True}\,,,\,\mathsf{False}$. This brings ambiguity, and the term is not well-typed, as we shall see in Section 2.4. Therefore, before substitution, the value must be further reduced with typed reduction under the expected type of the function argument. Thus the value that is substituted in the lambda body is 1 (but not $1\,,,\,\mathsf{True}$), and the final result is $1\,,,\,\mathsf{False}$.

These examples show some non-trivial aspects of typed reduction, which must decompose values, and possibly drop some of the components and permute other components. The details of the typed reduction relation will be discussed in Section 4. As we shall see functions introduce further complications.

2.3 The Challenges of Functions

One of the hardest challenges in designing the semantics of $\lambda_i^{:}$ was the design of the rules for functions. We discuss the challenges next.

Return Types Matter. As we have seen above, the input type annotation of lambdas is necessary during beta reduction. However, it is not enough to distinguish among multiple functions in a merge (e.g. $(\lambda x.\, x + 1)\,,,\,(\lambda x.\, \mathsf{True})$) without runtime type checking. Unlike primitive values, whose types can be told by their forms, for functions, we need the type of the function (including the output type) to select the right function from a merge. Therefore, in $\lambda_i^{:}$ all functions are annotated with both the input and output types. With such annotations we can deal with programs such as:

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

In this program we have a lambda that takes a function f as an argument and applies it to 1. The lambda is applied to the merge of two functions of types $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Int} \to \mathsf{Bool}$. To select the right function from the merge, the types of the functions are used to guide the reduction of the merge. This avoids the need for run-time type-checking, which would be otherwise necessary to recover the full type of functions.

Annotation Refinement. Given a value, for any of its supertypes, typed reduction gives a result. Since functions are values, sometimes this leads to the refinement of the type annotation of lambdas. Consider a single function $\lambda x.\, x\,,,\,\mathsf{True} : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$ to be reduced under type $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int}$. To let the function return an integer when applied to a merge of type $\mathsf{Int} \,\&\, \mathsf{Bool}$, we must change either the lambda body or the embedded annotation. Since reducing under a lambda body is not allowed in call-by-value, $\lambda_i^{:}$ adopts the latter option, and treats the input and output annotations differently. The input annotation should not be changed as it represents the expected input type of the function and helps to adjust the input value before substitution in beta reduction. The output annotation, in contrast, must be replaced by Int, representing a future reduction to be done after substitution. The output of the application then can be thought as an integer and can be safely merged with another boolean, for example. In short, the actual $\lambda_i^{:}$ reduction is:

$$\begin{aligned}&((\lambda x.\, x\,,,\,\mathsf{True}) : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}) : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int}\\ &\hookrightarrow (\lambda x.\, x\,,,\,\mathsf{True}) : \mathsf{Int} \to \mathsf{Int}\end{aligned}$$

2.4 Disjoint Intersection Types and Consistency for Determinism

Even if the semantics is type-directed and it rules out reductions that do not preserve types, it can still be non-deterministic. To solve this problem, we employ the disjointness restriction proposed by Oliveira et al. [35] and the novel notion of *consistency*. Both disjointness and consistency play a fundamental role in the proof of determinism.

Disjointness. Two types are disjoint (written as $A * B$), if any common supertypes that they have are *top-like types* (i.e. supertypes of any type; written as $\rceil C \lceil$).

► **Definition 1** (Disjoint Types).

$$A * B \equiv \forall C, if\ A <: C\ and\ B <: C\ then\ \rceil C \lceil$$

Intuitively, if two types are disjoint (e.g. $\mathsf{Int} \,\&\, \mathsf{Char} * \mathsf{Bool}$), their corresponding values do not overlap (e.g. $1\,,,\,$'c' and True). The only exceptions are top-like types, as they are disjoint with any types [2].

Since every value of a top-like type has the same effect, typed reduction unifies them to a fixed result. Thus the disjointness checking in the following typing rule guarantees that e_1 and e_2 can be merged safely, without any ambiguities. For example, this typing rule does not accept $1\,,,\,2$ or $\mathsf{True}\,,,\,1\,,,\,\mathsf{False}$, as two subterms of the merge have overlapped types (in this case, the same type Int and Bool, respectively).

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

Consistency. Recall the split rule (rule DStep-split) in Dunfield's semantics: $E \rightsquigarrow E\,,,\,E$. It duplicates terms in a merge. Similar things can happen in our typed reduction if the type has overlapping parts, which is allowed, for example, in an expression $1 : \mathsf{Int} \,\&\, \mathsf{Int}$. Note that in this expression the term 1 can be given type annotation $\mathsf{Int} \,\&\, \mathsf{Int}$ since $\mathsf{Int} <: \mathsf{Int} \,\&\, \mathsf{Int}$. During reduction, typed reduction is eventually used to create a value that matches the shape of type $\mathsf{Int} \,\&\, \mathsf{Int}$ by duplicating the integer:

$$1 \hookrightarrow_{Int\&Int} 1\,,,\,1$$

Note that the disjointness restriction does not allow sub-expressions in a merge to have the same type: $1\,,,\,1$ cannot type-check with rule Etyp-merge. To obtain *type preservation*, there is a special typing rule for merges of values, where a novel consistency check is used (written as $v_1 \approx v_2$):

$$\frac{\cdot \vdash v_1 : A \qquad \cdot \vdash v_2 : B \qquad v_1 \approx v_2}{\Gamma \vdash v_1\,,,\,v_2 : A \,\&\, B}\ \textsc{Etyp-mergev}$$

Mainly, consistency allows values to have overlapped parts as far as they are syntactically equal. For example, $1\,,,\,\mathsf{True}$ and $1\,,,\,$'c' are consistent, since the overlapped part Int in both of merges is the same value. True and 'c' are consistent because they are not overlapped at all. But $1\,,,\,\mathsf{True}$ and 2 are *not consistent*, as they have different values for the same type Int. In $\lambda_i^{:}$, consistency is defined in terms of typed reduction:

▶ **Definition 2** (Consistency). *Two values v_1 and v_2 are said to be consistent (written $v_1 \approx v_2$) if, for any type A, the result of typed reduction for the two values is the same.*

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

Finally, note that λ_i [35] is stricter than $\lambda_i^{:}$ and forbids any intersection types which are not disjoint. That is to say, the term $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is not well-typed because the intersection $\mathsf{Int} \,\&\, \mathsf{Int}$ is not disjoint. The idea of allowing unrestricted intersections, while only having the disjointness restriction for merges, was first employed in the NeColus calculus [4]. $\lambda_i^{:}$ follows such an idea and $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is well-typed in $\lambda_i^{:}$. Allowing unrestricted intersections adds extra expressive power. For instance, in calculi with polymorphism, unrestricted intersections can be used to encode *bounded quantification* [7], whereas with disjoint intersections only such an encoding does not work [6].

3 The $\lambda_i^{:}$ Calculus: Syntax, Subtyping and Typing

This section presents the syntax, subtyping, and typing of $\lambda_i^{:}$: a calculus with intersection types and a merge operator. This calculus is a small variant of the λ_i calculus [35] (which itself is inspired by Dunfield's calculus [20]) extended with *annotated expressions*, *explicit subsumption* and *fixpoints*. The explicit type annotations and subtyping are necessary for the type-directed operational semantics of $\lambda_i^{:}$ and to preserve determinism. The addition of fixpoints illustrates the ability of TDOS to deal with non-terminating programs, which are still not supported by calculi that rely on elaboration and semantic coherence proofs [4, 6].

3.1 Syntax

The syntax of $\lambda_i^{:}$ is:

Type	$A, B ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B$
Expr	$e ::= x \mid i \mid \top \mid e : A \mid e_1\ e_2 \mid \lambda x.e : A \to B \mid e_1 ,, e_2 \mid \textbf{fix}\ x.e : A$
Value	$v ::= i \mid \top \mid \lambda x.e : A \to B \mid v_1 ,, v_2$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$

Types. Meta-variables A and B range over types. Two basic types are included: the integer type *Int* and the top type Top. Function types $A \to B$ and intersection types $A \,\&\, B$ can be used to construct compound types. Following the convention introduced by previous works [35], $\to$ has lower precedence than &, which means $A \to B \,\&\, C$ equals to $A \to (B \,\&\, C)$.

Expressions. Meta-variable e ranges over expressions. Expressions include some standard constructs: variables (x); integers (i); a canonical top value $\top$; annotated expressions ($e : A$); and application of a term e_1 to term e_2 (denoted by $e_1 e_2$). Lambda abstractions ($\lambda x.e : A \to B$) must have a type annotation $A \to B$, meaning that the input type is A and the output type is B. The expression $e_1 ,, e_2$ is the merge of expressions e_1 and e_2. Finally, fixpoints $\textbf{fix}\, x.\, e : A$ (which also require a type annotation) model recursion.

Values and Contexts. The meta-variable v ranges over values. Values include integers, the canonical $\top$ value, lambda abstractions and merges of values. Typing contexts are standard. Γ tracks the bound variables x with their type A.

3.2 Subtyping and Disjointness

The subtyping rules of the form $A <: B$ are shown on the top of Figure 2. These subtyping rules, except for rule S-toparr, were first introduced by Davies and Pfenning [18], and are used in λ_i as well. The original subtyping relation, is known to be reflexive and transitive [18]. We proved the reflexivity and transitivity of the extended subtyping relation as well. There are 3 rules regarding intersection types. Together they define $A \,\&\, B$ as the greatest lower bound of A and B.

Top-like Types and Arrow Types. Intuitively, a top-like type is both a supertype and a subtype of Top, including the Top type and intersections of top-like types. The newly added rule S-toparr enlarges top-like types to include arrow types when their return types are top-like. A simple unary relation that captures top-like types inductively is defined on the bottom of Figure 2. The following theorem states the correctness and completeness of the definition.

▶ **Lemma 3** (Soundness and Completeness of the Definition of Top-like Types)**.**

$\rceil A \lceil$ *if and only if* Top $<: A$

Rule S-toparr is inspired by the following rule in BCD-style subtyping [3] (and adopted by Bi et al. [4]):

$$\frac{}{\mathsf{Top} <: \mathsf{Top} \to \mathsf{Top}}\ \text{BCD-toparr}$$

Since BCD-style subtyping includes a transitivity rule as an axiom, with this rule, Int $\to$ Top and Int $\to$ (Top $\to$ Top) are supertypes (and also subtypes) of Top. Due to the lack of built-in transitivity rule in $\lambda_i^{:}$'s subtyping, the above consequence has to be expressed more explicitly in the adapted rule S-toparr. We will come back to our motivation for including rule S-toparr in Section 3.3.

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

$$\textsc{S-z}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{S-top}\ \frac{}{A <: \mathsf{Top}} \qquad \textsc{S-arr}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \textsc{S-andR}\ \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}$$

$$\textsc{S-topArr}\ \frac{\mathsf{Top} <: B_2}{A <: B_1 \to B_2} \qquad \textsc{S-andL1}\ \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3} \qquad \textsc{S-andL2}\ \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3}$$

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

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

Figure 2 Subtyping rules of $\lambda_i^{;}$ and definition of top-like types

Disjointness. In Section 2.4, the specification of disjointness is presented. Such specification is a slightly more liberal version of the definition originally used in λ_i. In particular in our definition A and B themselves can be *top-like types*, which was forbidden in λ_i. An equivalent algorithmic definition of disjointness ($A *_a B$) is presented in Appendix A, which is the same as the definition in the NeColus calculus [4].

▶ **Lemma 4** (Disjointness Properties). *Disjointness satisfies:*

1. $A * B$ *if and only if* $A *_a B$.
2. *if* $A * (B_1 \to C)$ *then* $A * (B_2 \to C)$.
3. *if* $A * B \,\&\, C$ *then* $A * B$ *and* $A * C$.

3.3 Typing

The expression typing judgment $\Gamma \vdash e : A$ is standard. It says that in the typing environment Γ the expression e is of type A. Unlike λ_i, there is no well-formedness restriction on types[2]. This generalization is inspired by the calculus NeColus [4], where the well-formedness constraints are removed from λ_i, and expressions like $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ are allowed. In other words the calculus supports *unrestricted intersections* as well as *disjoint intersection types* (which are the only kind of intersections supported in λ_i).

The type system, shown in Figure 3, is syntax-directed. Most typing rules directly follow the declarative type system of λ_i, including the merge rule Etyp-merge, where disjointness is used. When two expressions have disjoint types, any parts from each of them do not have overlapping types. Therefore, their merge does not introduce ambiguity. With this restriction, rule Etyp-merge does not accept expressions like $1\,,,\,2$ or even $1\,,,\,1$. On the other hand, the novel rule Etyp-mergev allows *consistent* values to be merged regardless of their types. It accepts $1\,,,\,1$ while still rejecting $1\,,,\,2$. It is for values only, and values are closed. Therefore the type judgments appearing in it as premises

[2] The full syntax and typing rules for λ_i can be found in Section 5.2.

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

$$\textsc{Etyp-top}\quad \frac{}{\Gamma \vdash \top : \mathsf{Top}} \qquad \textsc{Etyp-lit}\quad \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \textsc{Etyp-var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Etyp-anno}\quad \frac{\Gamma \vdash e : B \qquad B <: A}{\Gamma \vdash (e : A) : A}$$

$$\textsc{Etyp-abs}\quad \frac{\Gamma, x : A \vdash e : B \qquad C <: A \qquad B <: D}{\Gamma \vdash (\lambda x.\, e : A \to D) : C \to D} \qquad \textsc{Etyp-fix}\quad \frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash (\mathbf{fix}\, x.\, e : A) : A} \qquad \textsc{Etyp-app}\quad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}$$

$$\textsc{Etyp-merge}\quad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 : A \,\&\, B} \qquad \textsc{Etyp-mergev}\quad \frac{\cdot \vdash v_1 : A \qquad \cdot \vdash v_2 : B \qquad v_1 \approx v_2}{\Gamma \vdash v_1 \,,,\, v_2 : A \,\&\, B}$$

Figure 3 Type system of $\lambda_i^{:}$

should have empty context, which is denoted by $\cdot$. Together the two rules support the determinism and type preservation of the TDOS, as discussed in Section 2.4.

Top-Like Types and Merges of Functions. We can finally come back to the motivation to include rule S-toparr in subtyping and depart from both Dunfield calculus and λ_i, which do not have such a rule. *Without the rule* S-toparr *in subtyping*, no arrow types are top-like, therefore two arrow types $A \to B$ and $C \to D$ are never disjoint in terms of Definition 1, as they have a common supertype $A \,\&\, C \to \mathsf{Top}$. Consequently, we can never create merges with more than one function, which is quite restrictive. For Dunfield this is not a problem, because he does not have the disjointness restriction. So his calculus supports merges of any functions (but it is incoherent). In λ_i an ad-hoc solution is proposed, by forcing the matter and employing the syntactic definition of top-like types in Figure 2 in disjointness. However this means that in λ_i Lemma 3 is false, since top-like function types are not supertypes of Top. In contrast, the approach we take in $\lambda_i^{:}$ is to add the rule S-toparr in subtyping. Now $\mathsf{Top} <: (A \,\&\, C \to \mathsf{Top})$ is derivable and thus $A \,\&\, C \to \mathsf{Top}$ is genuinely a top-like type. In turn this makes merges of multiple functions typeable without losing the intuition behind top-like types.

Type-Checking for Lambda Abstractions. Rule Etyp-abs can be thought as a combination of the standard typing rule and the subsumption rule. A well-typed lambda abstraction can have multiple types with the same return type. Its type annotation indicates the *principal input type* and the return type. Thus the input type can be any subtype of the principal one, since arrow types are contravariant in their argument types. While the principal input type describes the lambda's expectation on its argument, the annotated return type ensures the type of the evaluated result of lambdas. It just needs to be a supertype of the inner expression of the lambda. Rule Etyp-abs is inspired by the "distributed" use of subsumption in the $\lambda\&$ calculus [8].

Explicit Subsumption. Unlike many calculi where there is a general subsumption rule that can apply anywhere, in $\lambda_i^{:}$ subsumption needs to be explicitly triggered by a type annotation (except for lambdas, as explained above). The annotation rule Etyp-anno acts as explicit subsumption and assigns supertypes to expressions, provided a suitable type annotation. There is a strong motivation not to include a general (implicit) subsumption rule in calculi with disjoint intersection types. With an implicit subsumption rule disjointness alone is insufficient to prevent some ambiguous terms, as shown in the following example.

$$\textsc{Subsumption}\ \dfrac{\textsc{Etyp-merge}\ \dfrac{\textsc{Subsumption}\ \dfrac{\cdot \vdash 1 : \mathsf{Int} \qquad \mathsf{Int} <: \mathsf{Top}}{\cdot \vdash 1 : \mathsf{Top}} \qquad \cdot \vdash 2 : \mathsf{Int} \qquad \mathsf{Top} * \mathsf{Int}}{\cdot \vdash 1\,,,\,2 : \mathsf{Top}\,\&\,\mathsf{Int}} \qquad \mathsf{Top}\,\&\,\mathsf{Int} <: \mathsf{Int}\,\&\,\mathsf{Int}}{\cdot \vdash 1\,,,\,2 : \mathsf{Int}\,\&\,\mathsf{Int}}$$

Via the typical implicit subsumption, type Top is assigned to integer 1. Then 1 can be merged with 2 of type Int since their types are disjoint. At that time, the merged term $1\,,,\,2$ has type $\mathsf{Top}\,\&\,\mathsf{Int}$, which is a subtype of $\mathsf{Int}\,\&\,\mathsf{Int}$. By applying the subsumption rule again, the ambiguous term $1\,,,\,2$ finally bypasses the disjointness restriction, having type $\mathsf{Int}\,\&\,\mathsf{Int}$. However, note that with rule Etyp-anno we can still type-check the term $(1 : \mathsf{Top}),,2$, and reducing that term under the type *Int* can only unambiguously result in 2. The type annotation is key to prevent using the value 1 as an integer. Finally, the use of an explicit subsumption rule is a simpler alternative to bidirectional type-systems employed in other calculi with disjoint intersection types. Bidirectional type-checking is also capable of controlling subsumption, but adds more complexity.

Principal Types. The principal type of a value is the most specific one among all of its types, i.e. it is the subtype of any other type of the term. Its definition is syntax-directed.

▶ **Definition 5** (Principal types). *$type_p\langle v\rangle$ calculates the principal type of value v.*

$$\begin{aligned} type_p\langle \top \rangle &= \mathsf{Top} \\ type_p\langle n \rangle &= \mathsf{Int} \\ type_p\langle \lambda x.\, e : A \to B \rangle &= A \to B \\ type_p\langle v_1\,,,\,v_2 \rangle &= type_p\langle v_1 \rangle \;\&\; type_p\langle v_2 \rangle \end{aligned}$$

▶ **Lemma 6** (Principal Types). *For any value v, if its principal type is A, then*

1. *if $\cdot \vdash v : B$ then $A <: B$.*
2. *if $\cdot \vdash v : B$ and $B * C$ then $A * C$.*
3. *$\cdot \vdash v : A$.*

4 A Type-Directed Operational Semantics for $\lambda_i^{;}$

This section introduces the type-directed operational semantics for $\lambda_i^{;}$. The operational semantics uses type information arising from type annotations to guide the reduction process. In particular, a new relation called *typed reduction* is used to further reduce values based on the contextual type information, forcing the value to match the type structure. We show two important properties for $\lambda_i^{;}$: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck.

4.1 Typed Reduction of Values

To account for the type information during reduction $\lambda_i^{;}$ uses an auxiliary reduction relation called *typed reduction* for reducing values under a certain type. Typed reduction $v \hookrightarrow_A v'$ reduces the value v under type A, producing a value v' that has type A. It arises when given a value v of some type, where A is a supertype of the type of value, the value needs to be converted to a value compatible with

$\boxed{v \hookrightarrow_A v'}$ *(Typed reduction)*

$$\textsc{TReduce-lit}\quad \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{TReduce-top}\quad \frac{}{v \hookrightarrow_{\mathsf{Top}} \top} \qquad \textsc{TReduce-topArr}\quad \frac{\rceil B \lceil}{v \hookrightarrow_{A \to B} \lambda x.\, \top : \mathsf{Top} \to B}$$

$$\textsc{TReduce-arrow}\quad \frac{\neg \rceil D \lceil \qquad C <: A \qquad B <: D}{\lambda x.\, e : A \to B \hookrightarrow_{C \to D} \lambda x.\, e : A \to D} \qquad \textsc{TReduce-and}\quad \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 ,, v_2} \qquad \textsc{TReduce-mergevl}\quad \frac{v_1 \hookrightarrow_A v_1' \qquad A \;\mathsf{ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_1'}$$

$$\textsc{TReduce-mergevr}\quad \frac{v_2 \hookrightarrow_A v_2' \qquad A \;\mathsf{ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_2'}$$

Figure 4 Typed reduction of $\lambda_i^{:}$

the supertype A. The typed reduction ensures that values and types have a strong correspondence. If a value is well-typed, its principal type can be told directly by looking at its syntactic form.

Figure 4 shows the typed reduction relation. Rule TReduce-top expresses the fact that Top is the supertype of any type, which means that any value can be reduced under type Top. Similarly, rule TReduce-topArr indicates that any value reduces to a lambda abstraction $\lambda x.\, \top : \mathsf{Top} \to B$ under a top-like arrow type $A \to B$. Although it is not the only inhabited value of type $A \to B$, the reduction result has to be fixed for determinism. Rule TReduce-lit expresses that an integer value reduced under the supertype *Int* is just the integer value itself. Rule TReduce-arrow states that a lambda value $\lambda x.\, e : A \to B$, under a *non-top-like type* $C \to D$, evaluates to $\lambda x.\, e : A \to D$ if $C <: A$ and $B <: D$. The restriction that $C \to D$ is not top-like avoids overlapping with rule TReduce-topArr. Importantly rule TReduce-arrow changes the return type of lambda abstractions. For example:

$$(\lambda x.\, x ,, 2 : \mathsf{Char} \to \mathsf{Char} \,\&\, \mathsf{Int}) \hookrightarrow_{(\mathsf{Char}\,\&\,\mathsf{Int} \to \mathsf{Char})} \lambda x.\, x ,, 2 : \mathsf{Char} \to \mathsf{Char}$$

Intersections and Merges. In the remaining rules, we first decompose intersections. Then we only need to consider types that are not intersections, which are called *ordinary types* [18]:

$\boxed{A \;\mathsf{ordinary}}$ *(Ordinary types)*

$$\textsc{O-top}\quad \frac{}{\mathsf{Top}\;\mathsf{ordinary}} \qquad \textsc{O-int}\quad \frac{}{\mathsf{Int}\;\mathsf{ordinary}} \qquad \textsc{O-arrow}\quad \frac{}{A \to B\;\mathsf{ordinary}}$$

We take care of the value by going through every merge, until both value and types are in a basic form. Rule TReduce-mergevl and rule TReduce-mergevr are a pair of rules for reducing merges under an ordinary type. Since the type is not an intersection, the result contains no merge. Usually, we need to select between the left part and right part of a merge according to the type. The values of disjoint types do not overlap on non-top-like types. For example, $1 ,, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \hookrightarrow_{\mathsf{Int}} 1$ selects the left part. For top-like types, no matter which rule is applied, the reduction result is determined by the type only, as rule TReduce-top and rule TReduce-topArr suggest.

Rule TReduce-and is the rule that deals with intersection types. It says that if a value v can be reduced to v_1 under type A, and can be reduced to v_2 under type B, then its reduction result under type $A \,\&\, B$ is the merge of two results $v_1 \,,,\, v_2$. Note that this rule may *duplicate values*. For example $1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1 \,,,\, 1$. Such duplication requires special care, since the merge violates disjointness. The specially designed typing rule (rule Etyp-mergev) uses the notion of consistency (discussed in Section 4.2) instead of disjointness to type-check a merge of two values. Note also that such duplication implies that sometimes it is possible to use either rule TReduce-mergevl or rule TReduce-mergevr to reduce a value. For example, $1 \,,,\, 1 \hookrightarrow_{\mathsf{Int}} 1$. The consistency restriction (Definition 2) in rule Etyp-mergev ensures that no matter which rule is applied in such a case, the result is the same.

Example. A larger example to demonstrate how typed reduction works is:

$$(\lambda x.\, x \,,,\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Char}) \,,,\, (\lambda x.\, x : \mathsf{Bool} \to \mathsf{Bool}) \,,,\, 1$$
$$\hookrightarrow_{\mathsf{Int}\,\&\,(\mathsf{Int}\to\mathsf{Int})} 1 \,,,\, (\lambda x.\, x \,,,\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Int})$$

The initial value is the merge of two lambda abstractions and an integer. The target type is $\mathsf{Int} \,\&\, (\mathsf{Int} \to \mathsf{Int})$. Because the target type is an intersection, typed reduction first employs rule TReduce-and to decompose the intersection into Int and $\mathsf{Int} \to \mathsf{Int}$. Under type Int the value reduces to 1, and under type $\mathsf{Int} \to \mathsf{Int}$ it will reduce to $\lambda x.\, x \,,,\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Int}$. Therefore, we obtain the merge $1 \,,,\, (\lambda x.\, x \,,,\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Int})$ with type $\mathsf{Int} \,\&\, (\mathsf{Int} \to \mathsf{Int})$.

Basic Properties of Typed Reduction. Some properties of typed reduction can be proved directly by induction on the typed reduction derivation. First, when typed reduction is under a top-like type, the result only depends on the type. Second, typed reduction produces the same result whenever it is done directly or indirectly. Third, if a well-typed value can be typed reduced by some type, its principal type must be a subtype of that type.

▶ **Lemma 7** (Typed reduction on top-like types). *If* $\rceil A \lceil$, $v_1 \hookrightarrow_A v_1'$ *, and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

When typed reduction is under a top-like type, the result only depends on the type.

▶ **Lemma 8** (Transitivity of typed reduction). *If* $v \hookrightarrow_A v_1$*, and* $v_1 \hookrightarrow_B v_2$*, then* $v \hookrightarrow_B v_2$.

Typed reduction produces the same result whenever it is done directly or indirectly.

▶ **Lemma 9** (Typed reduction respects subtyping). *If* $v \hookrightarrow_A v'$*, then* $type_p\langle v\rangle <: A$.

This lemma relates typed reduction and subtyping. It states that if a well-typed value can be typed reduced by type A, its principal type must be a subtype of A.

4.2 Consistency and Type Soundness of Typed Reduction

Consistent values, as specified in Definition 2, introduce no ambiguity in typed reduction. Consider one type, if two consistent values both can reduce under the type, they should produce the same result. The *consistency* restriction ensures that duplicated values in a merge type-check, but it still rejects merges with different values of the same type. A value of a top-like type is consistent with any other value. It only type reduces under top-like types, which leads to a fixed result decided by the type.

Relating Disjointness and Consistency Assuming that two values have disjoint types, according to Lemma 6, their principal types must be disjoint as well. From Lemma 9, we can conclude that when the two values both reduce under a type, that type must be a common supertype of their principal types, which is known to be top-like (Definition 1). Furthermore, Lemma 7 implies that their reduction results are always the same under such top-like types, so they are consistent (Definition 2).

▶ **Lemma 10** (Consistency of disjoint values). *If* $A * B$*,* $\cdot \vdash v_1 : A$*, and* $\cdot \vdash v_2 : B$ *then* $v_1 \approx v_2$.

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

$$\textsc{Step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2} \qquad \textsc{Step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\, e_2 \hookrightarrow v_1\, e_2'} \qquad \textsc{Step-fix}\ \frac{}{\mathbf{fix}\, x.\, e : A \hookrightarrow e[x \mapsto \mathbf{fix}\, x.\, e : A]}$$

$$\textsc{Step-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,,\, e_2 \hookrightarrow e_1' \,,,\, e_2} \qquad \textsc{Step-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,,\, e_2 \hookrightarrow v_1 \,,,\, e_2'} \qquad \textsc{Step-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A}$$

$$\textsc{Step-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\, v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{Step-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'}$$

Figure 5 Call-by-value reduction of $\lambda_i^{:}$

Determinism and Type Soundness of Typed Reduction The merge construct makes it hard to design a deterministic operational semantics. Disjointness and consistency restrictions prevent merges like $1\,,,\,2$, and bring the possibility to deal with merges based on types. Typed reduction takes a well-typed value, which, if it is a merge, must be consistent (according to Lemma 10). When the two typed reduction rules for merges (rule TReduce-mergevl and rule TReduce-mergevr) overlap, no matter which one is chosen, either value reduces to the same result due to consistency (Definition 2). Indeed our typed reduction relation always produces a unique result for any legal combination of the input value and type. This serves as a foundation for the determinism of the operational semantics.

▶ **Lemma 11** (Determinism of Typed Reduction). *For every well-typed v (that is there is some type B such that $\cdot \vdash v : B$), if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.*

Via the transitivity lemma (Lemma 8) and the above determinism lemma, we obtain the following property: any reduction results of the given value are consistent.

▶ **Lemma 12** (Consistency after Typed Reduction). *If v is well-typed , and $v \hookrightarrow_A v_1$, and $v \hookrightarrow_B v_2$ then $v_1 \approx v_2$.*

The lemma shows that the reduction result of rule TReduce-and is always made of consistent values, which is needed in type preservation via the typing rule Etyp-mergev. Then a (generalized) type preservation lemma on typed reduction can be proved.

▶ **Lemma 13** (Preservation of Typed reduction). *If $\cdot \vdash v : B$ and $v \hookrightarrow_A v'$ then $\cdot \vdash v' : A$.*

In the particular case where $A = B$, this lemma shows that typed reduction preserves types. However, more generally, it shows that if a value is well-typed under a type B and it can be type reduced under another type A then the reduced value is always well-typed at type A. Finally, the typed reduction progress lemma is:

▶ **Lemma 14** (Progress of Typed Reduction). *If $\cdot \vdash v : A$, and $A <: B$, then $\exists v'$, $v \hookrightarrow_B v'$.*

4.3 Reduction

The reduction rules are presented in Figure 5. Most of them are standard. Rule Step-beta and rule Step-annov are the two rules relying on typed reduction judgments. Rule Step-beta says that a

$$
\begin{array}{rcl}
|\,i\,| & = & i \\
|\top| & = & \top \\
|\,\lambda x.\,e : A \to B\,| & = & \lambda x.\,|\,e\,| \\
|\,\mathbf{fix}\,x.\,e : A\,| & = & \mathbf{fix}\,x.\,|\,e\,| \\
|\,e : A\,| & = & |\,e\,| \\
|\,e_1\,e_2\,| & = & |\,e_1\,|\,|\,e_2\,| \\
|\,e_1\,,,\,e_2\,| & = & |\,e_1\,|\,,,\,|\,e_2\,|
\end{array}
$$

Figure 6 Type erasure for $\lambda_i^{;}$ expressions.

lambda value $\lambda x.\,e : A \to B$ applied to value v reduces by replacing the bound variable x in e by v'. Importantly v' is obtained by type reducing v under type A. In other words, in Rule Step-beta further (typed) reduction may be necessary on the argument depending on its type. This is unlike many other calculi where values are in a final form and no further reduction is needed (thus the value v can be directly substituted). The rule Step-annov says that an annotated $v : A$ can be reduced to v' if v type reduces to v' under type A.

Metatheory of Reduction When designing the operational semantics of $\lambda_i^{;}$, we want it to have two properties: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck. Similar lemmas on typed reduction were already presented, which are necessary for PROVING the following theorems, mainly in cases related to rule Step-annov and rule Step-beta.

▶ **Theorem 15** (Determinism of $\hookrightarrow$). *If* $\cdot \vdash e : A$, $e \hookrightarrow e_1$, $e \hookrightarrow e_2$, *then* $e_1 = e_2$.

▶ **Theorem 16** (Type Preservation of $\hookrightarrow$). *If* $\cdot \vdash e : A$, *and* $e \hookrightarrow e'$ *then* $\cdot \vdash e' : A$.

▶ **Theorem 17** (Progress of $\hookrightarrow$). *If* $\cdot \vdash e : A$, *then e is a value or* $\exists e'$, $e \hookrightarrow e'$.

5 Relationship to Dunfield's Calculus and λ_i

Dunfield's calculus [20] and λ_i [35] are two calculi that directly inspired $\lambda_i^{;}$. In this section, we discuss the relationship between $\lambda_i^{;}$ and them. First, we show that $\lambda_i^{;}$'s TDOS and a slightly extended version of Dunfield's non-deterministic operational are related. The need for extending Dunfield's original semantics is mostly due to the addition of the rule S-toparr in subtyping, which Dunfield does not have. In Section 6 we also discuss a variant of $\lambda_i^{;}$ (which does not include rule S-toparr) and show that such variant requires no changes to Dunfield's original semantics. The other relationship is between $\lambda_i^{;}$'s type system and λ_i's type system. The former comparison shows the soundness of the operational semantics of $\lambda_i^{;}$ with respect to Dunfield's semantics. The latter one proves that $\lambda_i^{;}$'s type system is at least as expressive as, if not stronger than, λ_i's.

Type Erasure. Differently from the other two systems, $\lambda_i^{;}$ uses type annotations in its syntax to obtain a direct operational semantics. $|\,e\,|$ erases annotations in term e. By erasing all annotations, terms in $\lambda_i^{;}$ can be converted to terms in Dunfield's system and λ_i. The only exception are fixpoints, which λ_i does not have. The annotation erasure function is defined in Figure 6. Note that for every value v in $\lambda_i^{;}$, $|\,v\,|$ is a value as well.

5.1 Soundness with respect to Dunfield's Operational Semantics

Dunfield's original reduction rules are presented in Fig 1. We extend his operational semantics with the following two rules [3].

$\boxed{E \rightsquigarrow E'}$ *(The extension of Dunfield's calculus)*

$$\frac{}{E \rightsquigarrow \top}\ \textsc{DStep-top} \qquad \frac{}{\top \rightsquigarrow \lambda x.\, \top}\ \textsc{DStep-topArr}$$

Rule DStep-top states that any value can be reduced to $\top$, corresponding to $A <: \mathsf{Top}$. Rule DStep-topArr says that the value $\top$ can be reduced to a lambda which returns $\top$, suggested by the subtyping rule S-topArr. Together with merge rules, the extended reduction can reduce any term to a value under a top-like type. Dunfield avoids having a rule DStep-top by performing a simplifying elaboration step advance:

$$\frac{}{\Gamma \vdash V : \mathsf{Top} \hookrightarrow \top}\ \textsc{Dunfield-Typing-T}$$

With such a rule values of type Top are directly translated into $\top$, and do not need any further reduction in the target language. We do not have such a step and we have added rule DStep-topArr, so instead, we extend the original semantics with the two rules.

Soundness. Given Dunfield's extended semantics, we can show a theorem that each step in the TDOS of λ_i corresponds to zero, one, or multiple steps in Dunfield's semantics.

▶ **Theorem 18** (Soundness of $\hookrightarrow$ with respect to Dunfield's semantics)**.** *If $e \hookrightarrow e'$, then $|\,e\,| \rightsquigarrow^* |\,e'\,|$.*

A necessary lemma for this theorem is the soundness of typed reduction.

▶ **Lemma 19** (Soundness of Typed Reduction with respect to Dunfield's semantics)**.** *If $v \hookrightarrow_A v'$, then $|\,v\,| \rightsquigarrow^* |\,v'\,|$.*

This lemma shows that although the type information guides the reduction of values, it does not add additional behavior to values. For example, a merge can step to its left part (or the right part) with rule TReduce-mergevl (or rule TReduce-mergevr), corresponding to rule DStep-unmergel (or rule DStep-unmerger). And rule TReduce-and ($v \hookrightarrow_{A\,\&\,B} v_1 \,,,\, v_2$ if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_B v_2$) can be understood as a combination of spliting (rule DStep-split $V \rightsquigarrow V \,,,\, V$) and further reduction on each component separately.

In Section 6, we present another variant of $\lambda_i^{;}$, which has the same subtyping relation as Dunfield's system (minus union types). The same soundness theorem is proved for that variant without any modifications to Dunfield's operational semantics.

5.2 Completeness with respect to the Type System of λ_i

λ_i drops union types and introduces the disjointness restriction to Dunfield's system. When introducing λ_i, Oliveira et al. proposed an algorithmic and a declarative type system. The two type systems were shown to be equally expressive. For the declarative type system there is still the possibility of

[3] The full reduction rules can be found in Appendix B

$\boxed{\Gamma \models A}$ *(Type wellformedness)*

$$\textsc{Wf-top}\ \frac{}{\Gamma \models \mathsf{Top}} \qquad \textsc{Wf-int}\ \frac{}{\Gamma \models \mathsf{Int}} \qquad \textsc{Wf-arr}\ \frac{\Gamma \models A \qquad \Gamma \models B}{\Gamma \models A \rightarrow B} \qquad \textsc{Wf-and}\ \frac{\Gamma \models A \qquad \Gamma \models B \qquad A * B}{\Gamma \models A \,\&\, B}$$

$\boxed{\Gamma \models E : A}$ *(Typing)*

$$\textsc{ITyp-top}\ \frac{}{\Gamma \models \top : \mathsf{Top}} \qquad \textsc{ITyp-lit}\ \frac{}{\Gamma \models i : \mathsf{Int}} \qquad \textsc{ITyp-var}\ \frac{x : A \in \Gamma}{\Gamma \models x : A} \qquad \textsc{ITyp-lam}\ \frac{\Gamma \models A \qquad \Gamma, x : A \models E : B}{\Gamma \models (\lambda x.\, E) : A \rightarrow B}$$

$$\textsc{ITyp-app}\ \frac{\begin{array}{c}\Gamma \models E_1 : A \rightarrow B \\ \Gamma \models E_2 : A\end{array}}{\Gamma \models E_1\, E_2 : B} \qquad \textsc{ITyp-merge}\ \frac{\begin{array}{c}\Gamma \models E_1 : A \\ \Gamma \models E_2 : B \qquad A * B\end{array}}{\Gamma \models E_1 \,,,\, E_2 : A \,\&\, B} \qquad \textsc{ITyp-sub}\ \frac{\Gamma \models E : A \qquad A <: B}{\Gamma \models E : B}$$

Figure 7 The declarative type system of λ_i.

ambiguity due to the presence of an (implicit) subsumption rule (see also the discussion in Section 3.3). However, annotations in the bidirectional algorithmic type system ensure that well-typed terms in λ_i are unambiguous and subsumption is kept under control.

The type system of $\lambda_i^{:}$ is based on the declarative type system of λ_i, with three main changes:

1. $\lambda_i^{:}$ forces the subsumption rule to be explicitly triggered by a type annotation.
2. $\lambda_i^{:}$ supports fixpoints while λ_i does not.
3. $\lambda_i^{:}$ has an additional rule for the merge of values (rule Etyp-mergev), which is required to prove type preservation, since duplicated values can occur in merges after reduction.

Some details need to be explained before presenting the completeness theorem. Firstly, because they are irrelevant, rules related to products and projection operators in λ_i are dropped. Secondly, the subtyping in $\lambda_i^{:}$ is stronger due to the added rule S-topArr.

▶ **Theorem 20** (Completeness of Typing with respect to λ_i). *If $\Gamma \models E : A$, then there exists some e such that $\Gamma \vdash e : A$ and $E = |e|$.*

The above theorem shows that the type system of $\lambda_i^{:}$ is able to type check any well-typed terms in λ_i, with proper type annotations inserted based on the typing derivation. The result means that λ_i's type system (or any type system equivalent to it) can be used as a surface language where many of the explicit annotations of $\lambda_i^{:}$ are inferred automatically. That is to say, the λ_i calculus can be translated without loss of expressivity or flexibility into $\lambda_i^{:}$.

6 Discussion

This section discusses one variant of $\lambda_i^{:}$, which is also formalized in Coq. The variant follows the subtyping relation in λ_i and Dunfield's calculus strictly and does not support multiple functions in merges. Some possible extensions to our work are also discussed.

6.1 A Variant of $\lambda_i^{:}$

In Section 5.1, we validate the TDOS of $\lambda_i^{:}$ via a soundness theorem (Theorem 18) with respect to an extended operational semantics of Dunfield's calculus. In this section, we discuss a variant of $\lambda_i^{:}$ that requires no extension to Dunfield's operational semantics[4]. Instead of adding rule S-TOPARR, this variant keeps the same subtyping relation as Dunfield's and adapts the definition of top-like types and disjointness, losing the ability to have multiple functions in a merge. Consequently, it is possible to prove the following soundness theorem on this variant without any modifications on Dunfield's operational semantics [5].

▶ **Theorem 21** (Soundness of $\hookrightarrow$ in the simple variant)**.** *If $e \hookrightarrow e'$, then $|e| \rightsquigarrow^* |e'|$.*

The above theorem states that each step taken by the TDOS corresponds to a series of reduction in the original operational semantics of Dunfield's calculus.

Besides soundness, this variant keeps the other important properties of λ_i: *determinism*, *type preservation* and *progress*. A completeness theorem with respect to the type system of λ_i is established as well.

▶ **Theorem 22** (Completeness of Typing in the simple variant)**.** *If $\Gamma \models E : A$ in λ_i, then there exists some e such that $\Gamma \vdash e : A$ in the variant and $E = |e|$.*

Designing the Variant. As presented in Section 5.1, there are two reduction rules in $\lambda_i^{:}$ that are related to the extension of Dunfield's operational semantics: rule TREDUCE-TOP ($v \hookrightarrow_{\mathsf{Top}} \top$) and rule TREDUCE-TOPARR. They reduce values under top-like types into a unified form. Without rule S-TOPARR, no arrow types are top-like, thus the latter is removed from the variant. However there is still rule TREDUCE-TOP, which is not accounted for in Dunfield's original system. While we believe that such a rule fits in spirit well with the remaining non-deterministic rules, it is interesting to see if it is possible to model a calculus without it (and without extending Dunfield semantics at all).

Reducing a value v under type Top, in fact, can be thought as seeking an inhabited value of Top which acts like v. In Dunfield's original semantics there is no way to convert a value to $\top$, which is our source of difficulties. Dunfield solves this problem by having a typing rule for values that allows any value to have type Top. This works well in his setting because he can have ambiguous terms. Unfortunately, it does not work well in our setting because, as discussed in detail in Section 3.3, allowing values to be implicitly typed as Top provides a way to bypass the disjointness restrictions. We overcome the problem instead by introducing a new value construct $v : \mathsf{Top}$ in our variant. This new form of value inhabits the Top type but, unlike $\top$ it does not forget about the original value v (which can be of any type). Thus the original value v, can be recovered by erasing the wrapped annotation.

$$\text{TRED-TOP}\quad \frac{}{v \hookrightarrow_{\mathsf{Top}} v : \mathsf{Top}}$$

Although the new rule then corresponds to $v \rightsquigarrow v$ after annotation erasure, it breaks determinism as a merge can reduce to either its left or right component, leading to different results, e.g. $1,, \mathsf{True} \hookrightarrow_{\mathsf{Top}} 1 : \mathsf{Top}$ and $1,, \mathsf{True} \hookrightarrow_{\mathsf{Top}} \mathsf{True} : \mathsf{Top}$. To solve this problem we directly reduce the value before splitting merges by excluding Top from ordinary types.

To use the new construct for expressions and mix it with annotated terms, values and expressions are separated into two syntactic categories in the variant (but all values can be treated as

[4] The full syntax and typing rules for the variant can be found in Appendix C.
[5] For the syntax and rules of Dunfield's system, please refer to Section 2.

expressions $\langle v \rangle$). The partition results in some tedious rules in the reduction relation. For instance, $\langle v_1 \rangle \,,, \langle v_2 \rangle \hookrightarrow \langle v_1 \,,, v_2 \rangle$ reduces a merge of two values to a merged value.

6.2 Improvements and Extensions

Less Checks on Reduction. In rule TReduce-arrow (in Figure 4), the premise $C <: A$ is actually redundant for the purposes of reduction. Since we only care about well-typed terms being reduced, such a check has already been guaranteed by typing. Therefore an actual implementation could omit that check. The reason why we keep the premise is that typed reduction plays another role in our metatheory: it allows us to define consistency. Consistency is defined for any (untyped) values, and the extra check there tightens up the definition of consistency. With the premise, typed reduction directly implies a subtyping relation between the principal type of the reduced value and the reduction type. (See Lemma 9: If $v \hookrightarrow_A v'$, then $type_p\langle v \rangle <: A$). One could wonder if this property is unnecessary because it may be derived by type preservation of reduction. Note that whenever typed reduction is called in a reduction rule, the subtyping relation can be obtained from the typing derivation of the reduced term. For example, reducing $v : A$ will type reduce v under A. If $v : A$ is well-typed, then we could in principle prove that $type_p\langle v \rangle <: A$. Unfortunately, the above proof is hard to attain in practice. Because type preservation depends on consistency, and consistency is defined by typed reduction. Once the subtyping property relies on type preservation, there is a cyclic dependency between the properties. In future work we would like to look at this issue more closely and try to discard the premise by taking full advantage of the type system.

Distributive Subtyping. Although the subtyping of $\lambda_i^{;}$ allows multiple functions in a merge, it lacks the distributive subtyping rule for intersection types that has been employed in some recent calculi [4, 6]. The distributivity of intersections over arrows ($(A \to B_1) \,\&\, (A \to B_2) <: A \to B_1 \,\&\, B_2$) [3] is well accepted for its theoretical elegance. But it is also well-known for being troublesome. Mainly, there are two challenges for adapting distributive subtyping to $\lambda_i^{;}$.

- The rule indicates that a merge of functions can be applied. While the current typing rule can check such application with suitable annotations, designing new reduction rules is necessary. A promising solution is to have a rule allows parallel application like $(v_1 \,,, v_2)\, v \hookrightarrow v_1\, v \,,, v_2\, v$.
- Function types are no longer “ordinary". In $\lambda_i^{;}$, the intuition behind ordinary types is that their typed reduction results never contains merges, which is necessary for determinism. With distributivity, typed reduction may produce a merge under a single function type. For example, $\lambda x.\, \text{‘}c\text{’} : \mathsf{Int} \to \mathsf{Char} \,,, \lambda x.\, x : \mathsf{Int} \to \mathsf{Int} \hookrightarrow_{\mathsf{Int} \to \mathsf{Char} \,\&\, \mathsf{Int}} \lambda x.\, \text{‘}c\text{’} : \mathsf{Int} \to \mathsf{Char} \,,, \lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$. In the typed reduction of $\lambda_i^{;}$, intersections are split into basic units. However, it is not straightforward to split a function type.

7 Related Work

7.1 Calculi with the Merge Operator and a Direct Semantics

Intersection types with a merge operator are a key feature of Reynolds' Forsythe language [43]. Reynolds studied a core calculus [43] with similarities to $\lambda_i^{;}$. However, merges in Forsythe are restricted and use a syntactic criterion to determine what merges are allowed. A merge is permitted only when the second term is a lambda abstraction or a single field record, which makes the structure of merge always biased. To prevent potential ambiguity, the latter overrides the former when overlapped. Note that the structure of merge in Forsythe is always biased. If formalized as a tree, the right child of every node is a leaf. The only place for primitive types is the leftmost component. Forsythe follows the standard call-by-name small-step reduction, during which types are ignored. The reduction rules

	Dunfield's [20]	λ_i [35]	F_i [2]	NeColus [4]	F_i^+ [6]	$\lambda_i^{;}$
Disjointness	○	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●
Determinism or Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.
Coercion Free	●	○	○	○	○	●
Recursion	●	○	○	○	○	●
Direct Semantics	●	○	○	○	○	●
Subject-Reduction	○	-	-	-	-	●

Figure 8 Summary of intersection calculi with the merge operator (● = yes, ○ = no, - = not applicable)

deal with merges by continuously checking if the second component can be used in the context (abstractions for application, records for projection). This simple approach, however, is unable to reduce merges when (multiple) primitive types are required. Reynolds admits this issue in his later work [42]. In $\lambda_i^{;}$ types are used to select values from a merge and the disjointness restriction guarantees the determinism. Therefore the order of a value in a merge is not a deciding factor on whether the value is used or not.

The calculus $\lambda\&$ proposed by Castagna et al. [8] has a restricted version of the merge operator for functions only. The merge operator is indexed by a list of types of its components. The operational semantics uses the runtime types of values to select the "best approximate" branch of an overloaded function. $\lambda\&$ requires runtime type checking on values, while in TDOS, all type information is present already in type annotations. Another obvious difference is that $\lambda_i^{;}$ supports merges of any types (not just functions), which are useful for applications other than overloading of functions, including: multifield *extensible records with subtyping* [35]; encodings of *objects* and *traits* [5]; *dynamic mixins* [2]; or simple forms of *family polymorphism* [4].

Several other calculi with intersection types and overloading of functions have been proposed [9–11], but these calculi do not support a merge operator, and thus avoid the ambiguity problems caused by the construct.

7.2 Calculi with a Merge Operator and an Elaboration Semantics

Instead of a direct semantics, many recent works [2, 4, 6, 20, 35] on intersection types employ an elaboration semantics, translating merges in the source language to products (or pairs) in a target language. With an elaboration semantics the subtyping derivations are coercive [29]: they produce coercion functions that explicitly convert terms of one type to another in the target language. This idea was first proposed by Dunfield [20], where he shows how to elaborate a calculus with intersection and union types and a merge operator to a standard call-by-value lambda calculus with products and sums. Dunfield also proposed a direct semantics, which served as inspiration for our own work. However, his direct semantics is non-deterministic and lacks subject-reduction (as discussed in detail in Section 2.1). Unlike Forsythe and $\lambda\&$, Dunfield's calculus has unrestricted merges and allows a merge to work as an argument. His calculus is flexible and expressive and can deal with several programs that are not allowed in Forsythe and $\lambda\&$.

To remove the ambiguity issues in Dunfield's work, the λ_i calculus [35] forbids overlapping in intersections using the disjointness restriction for all well-formed intersections. In other words, λ_i does not support unrestricted intersections. Because of this restriction, the proof of coherence in λ_i is still relatively simple. Likewise, in following work on the F_i calculus [2], which extends λ_i with disjoint polymorphism, *all* intersections must be *disjoint*. However the disjointness restriction causes difficulties because it breaks *stability of type substitutions*. Stability is a desirable property in a polymorphic type system that ensures that if a polymorphic type is well-formed then any instantiation of that type is also well-formed. Unfortunately, with disjoint intersections only, this property is not true in general. Thus F_i can only prove a restricted version of stability, which makes its metatheory non-trivial.

Disjointness of all well-formed intersections is only a sufficient (but not necessary) restriction to ensure an unambiguous semantics. The NeColus calculus [4] relaxes the restriction without introducing ambiguity. In NeColus $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is allowed, but the same term is rejected in λ_i. In other words, NeColus employs the disjointness restriction *only* on merges, but otherwise allows *unrestricted intersections*. Unfortunately, this comes at a cost: it is much harder to prove the coherence of elaboration. Both NeColus and F_i^+ [6] (a calculus derived from F_i that allows unrestricted intersections) deal with this problem by establishing coherence using contextual equivalence and a *logical relation* [38,46,47] to prove it. The proof method, however, cannot deal with non-terminating programs. In fact none of the existing calculi with disjoint intersection types supports recursion, which is a severe restriction.

We retain the essence of the power of Dunfield's calculus (modulo the disjointness restrictions to rule out ambiguity), and gain benefits from the direct semantics. Figure 8 summarizes the key differences between our work and prior work, focusing on the most recent work on disjoint intersection types. Note that the row titled "Coercion Free" denotes whether subtyping generates coercions or not. $\lambda_i^{;}$ is coercion free, while all other calculi based on an elaboration semantics employ coercive subtyping. Next we give more detail on the advantages of a direct semantics over the elaboration semantics and proof methods employed in previous work on disjoint intersection types.

Shorter, more Direct Reasoning. Programmers want to understand the meaning of their programs. A formal semantics can help with this. With our TDOS semantics we can essentially employ a style similar to equational reasoning in functional programming to directly reason about programs written in $\lambda_i^{;}$. For example, it takes a few reasoning steps to work out the result of $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2 \,,,\, `c')$:

$$\begin{array}{ll}
\quad (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2 \,,,\, `c') & \\
\hookrightarrow (2 + 1) : \mathsf{Int} & \{\text{by } \textsc{Step-beta} \text{ and typed reduction of argument under } \mathsf{Int}\} \\
\hookrightarrow 3 : \mathsf{Int} & \{\text{by } \textsc{Step-anno} \text{ and usual reduction rules for arithmetic}\} \\
\hookrightarrow 3 & \{\text{by } \textsc{Step-annov} \text{ and typed reduction of 3 under } \mathsf{Int}\}
\end{array}$$

Here reasoning is easily justifiable from the small-step reduction rules and type-directed reduction. In fact building tools (such as some form of debugger), that automate such kind of reasoning should be easy using the TDOS rules.

However, with an elaboration semantics, the (precise) reasoning steps to determine the final result are much more complex. Firstly the expression has to be translated into the target language before reducing to a similar target term. Figure 9 shows this elaboration process in λ_i, where an expression in the source language is translated into an expression in a target language with products. The source term $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2 \,,,\, `c')$ is elaborated into the target term $(\lambda x.\, x + 1)\,(\mathsf{fst}\,(2 \,,\, `c'))$. As we can see the actual derivation is rather long, so we skip the full steps. Also, for simplicity's sake, here we assume the subtyping judgement produces the most straightforward coercion fst. This elaboration step together with the introduction of coercions into the program makes it much harder for programmers to precisely understand the semantics of a program. Moreover while the coercions inserted in this small expression may not look too bad, in larger programs the addition of coercions can be alot more severe, hampering the understanding of the program. After elaboration we can then

$$\textsc{T-app}\ \dfrac{\textsc{T-ann}\ \dfrac{\dots}{\cdot \vdash (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int}) \Rightarrow \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, x + 1} \qquad \textsc{T-sub}\ \dfrac{\textsc{T-merge}\ \dfrac{\dots}{\cdot \vdash (2 \,,, \, \text{‘}c\text{’}) \Rightarrow \mathsf{Int} \,\&\, \mathsf{Char} \rightsquigarrow (2 \,, \text{‘}c\text{’})} \qquad \dfrac{\textsc{Sub-AndL}}{\mathsf{Int} \,\&\, \mathsf{Char} <: \mathsf{Int} \rightsquigarrow \mathsf{fst}}}{\cdot \vdash (2 \,,, \, \text{‘}c\text{’}) \Leftarrow \mathsf{Int} \rightsquigarrow \mathsf{fst}\, (2 \,, \text{‘}c\text{’})}}{\cdot \vdash (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\, (2 \,,, \, \text{‘}c\text{’}) \Rightarrow \mathsf{Int} \rightsquigarrow (\lambda x.\, x + 1)\, (\mathsf{fst}\, (2 \,, \text{‘}c\text{’}))}$$

Figure 9 Elaboration of the expression $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\, (2 \,,, \, \text{‘}c\text{’})$ into a target calculus with products.

use the target language semantics, to determine a target language value.

$$\begin{array}{lll}
 & (\lambda x.\, x + 1)\, (\mathsf{fst}\, (2 \,, \text{‘}c\text{’})) & \\
\hookrightarrow & (\lambda x.\, x + 1)\, 2 & \{\text{by a rule similar to } \textsc{Step-appr} \text{ and reduction rules for pairs}\} \\
\hookrightarrow & 2 + 1 & \{\text{by beta reduction rule}\} \\
\hookrightarrow & 3 & \{\text{by usual reduction rules for arithmetic}\}
\end{array}$$

A final issue is that sometimes it is not even possible to translate back the value of the target language into an equivalent "value" on the source. For instance in the NeColus calculus [4] $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ results in $(1,\ 1)$, which is a pair in the target language. But the corresponding source value $1 \,,, \, 1$ is not typable in NeColus. In essence, with an elaboration, programmers must understand not only the source language, but also the elaboration process as well as the semantics of the target language, if they want to precisely understand the semantics of a program. Since the main point of semantics is to give clear and simple rules to understand the meaning of programs, a direct semantics is a better option for providing such understanding.

Simpler Proofs of Unambiguity. For calculi with an elaboration semantics, unrestricted intersections make it harder to prove the coherence. Our $\lambda_i^{;}$ calculus, on the other hand, has a deterministic semantics, which implies unambiguity directly. For instance, $(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int}$ only steps to 1 in $\lambda_i^{;}$. But it can be elaborated into two target expressions in the NeColus calculus corresponding to two typing derivations:

$$(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{fst}\, (1,\ 1)$$

$$(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{snd}\, (1,\ 1)$$

Thus the coherence proof needs deeper knowledge about the semantics: the two different terms are known to both reduce to 1 eventually. Therefore they are related by the logical relation employed in NeColus for coherence. Things get more complicated for functions. The following example shows two possible elaborations of the same function. To relate them requires reasoning inside the binders and a notion of contextual equivalence.

$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{fst}\ x + 1$$

$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{snd}\ x + 1$$

Furthermore, the two target expressions above are *clearly not equivalent* in the general case. For instance, if we apply them to (1, 2) we get different results. However, the target expressions will always behave equivalently when applied to arguments *elaborated from the* NeColus *source calculus*. NeColus, forbids terms like (1 , , 2) and thus cannot produce a target value (1, 2). Because of elaboration and also this deeper form of reasoning required to show the equivalence of semantics, calculi defined by elaboration require a lot more infrastructure for the source and target calculi and the elaboration between them, while in a direct semantics only one calculus is involved and the reasoning required to prove determinism is quite simple.

Not Limited to Terminating Programs. The (basic) forms of logical relations employed by NeColus and F_i^+ has cannot deal with non-terminating programs. In principle, recursion could be supported by using a *step-indexed logical relation* [1], but this is left for future work (and it is non-trivial). $\lambda_i^{:}$ smoothly handles unrestricted intersections and recursion, using TDOS to reach determinism with a significantly simpler proof method. It also makes other features that lead to non-terminating programs, such as *recursive types*, feasible.

7.3 Languages and Calculi with Type-Dependent Semantics

Typed Operational Semantics Goguen [25] uses types in its reduction judgment, similarly to typed reduction in $\lambda_i^{:}$. However, Goguen's typed operational semantics is designed for studying meta-theoretic properties, especially strong normalization, and is not aimed to describe type-dependent semantics. Unlike TDOS, in typed operational semantics the reduction process does not use the additional type information to guide reduction. Instead, the combination of well-typedness and computation provides inversion principles for proving various metatheoretical properties. Typed operational semantics has been applied to several systems. These include *simply typed lambda calculi* [26], calculi with *dependent types* [23, 25] and *higher-order subtyping* [16]. Note that the semantics of these systems does not depend on typing, and the untyped (type-erased) reduction relations are still presented to describe how to evaluate programs.

Type classes [28, 48] are an approach to parametric overloading used in languages like Haskell. The commonly adopted compilation strategy for it is the dictionary passing style elaboration [12, 13, 27, 48]. Other mechanisms inspired by type classes, such as Scala's *implicits* [34], Agda's *instance arguments* [19] or Ocaml's *modular implicits* [50] have an elaboration semantics as well. In one of the pioneering works of type classes, Kaes [28] gives two formulations for a direct operational semantics. One of them decides the concrete type of the instance of overloaded functions at *run-time*, by analyzing all arguments after evaluating them. In both Kaes' work and a following work by Odersky et al. [33], the run-time semantics has some restrictions with respect to type classes. For example, overloading on return types (needed for example for the *read* function in Haskell) is not supported. Interestingly, the semantics of $\lambda_i^{:}$ allows overloading on return types, which is used whenever two functions coexist on a merge.

Gradual typing [44] has become popular over the last few years. Gradual typing is another example of a type-dependent mechanism, since the success or not of an (implicit) cast may depend on the particular type used for the implicit cast. Thus the semantics of a gradually typed language is type-dependent. Like other type-dependent mechanisms the semantics of gradually typed source languages is usually given by a (type-dependent) elaboration semantics into a cast calculus, such as the *Blame calculus* [49] or the *Threesome calculus* [45]. *Multiple dispatching* [14, 15, 31, 36] generalizes object-oriented dynamic dispatch to determine the overloaded method to invoke based on the runtime type of all its arguments. Similarly to TDOS, much of the type information is recovered from type annotations in multiple dispatching mechanisms, but, unlike TDOS, they only use input types to determine the semantics.

8 Conclusion

In this work we presented a TDOS for $\lambda_i^{:}$: a calculus that includes intersection types and an expressive unbiased merge operator. Among all similar calculi, $\lambda_i^{:}$ is the first to have a direct operational semantics that is both deterministic and has subject-reduction. Compared with the elaboration approach, having a direct semantics avoids the translation process and a target calculus. This simplifies both informal and formal reasoning. For instance, establishing the coherence of elaboration in NeColus [4] requires much more sophistication than obtaining the determinism theorem in $\lambda_i^{:}$. Furthermore the proof method for coherence in NeColus cannot deal with non-terminating programs, whereas dealing with recursion is straightforward in $\lambda_i^{:}$. The semantics of $\lambda_i^{:}$ exploits type annotations to guide reduction. The key component of TDOS is *typed reduction*, which allows values to be further reduced depending on their type. For the future we would like to develop further the TDOS approach in the setting of disjoint intersection types. Some interesting extensions include support for *distributive subtyping* [3], and *disjoint polymorphism* [2].

References

1 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP '06*, 2006.

2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *ESOP '17*, 2017.

3 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic*, 48(4), 1983.

4 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In *ECOOP '18*, 2018.

5 Xuan Bi and Bruno C. d. S. Oliveira. Typed first-class traits. In *ECOOP '18*, 2018.

6 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In *ESOP '19*, 2019.

7 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, page 471–523, December 1985.

8 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1), 1995.

9 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *POPL '15*, 2015.

10 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL '14*, 2014.

11 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP '11*, 2011.

12 Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05*, 2005.

13 Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05*, 2005.

14 Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In *OOPSLA '99*, 1999.

15 Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, 2000.

16 Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher order subtyping. In *Information and Computation*, 1997.

17 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

18 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages

198–208, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/351240.351259`, `doi:10.1145/351240.351259`.

19 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *ICFP '11*, 2011.

20 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3), 2014.

21 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS '03*, 2003.

22 Facebook. Flow. `https://flow.org/`, 2014.

23 Yangyue Feng and Zhaohui Luo. Typed operational semantics for dependent record types. In *TYPES '09*, 2010.

24 Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. URL: `http://doi.acm.org/10.1145/113445.113468`, `doi:10.1145/113445.113468`.

25 Healfdene Goguen. *A typed operational semantics for type theory*. PhD thesis, University of Edinburgh, 1994.

26 Healfdene Goguen. Typed operational semantics. In *TLCA '95*, 1995.

27 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), 1996.

28 Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88*, 1988.

29 Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

30 Microsoft. Typescript. `https://www.typescriptlang.org/`, 2012.

31 Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *OOPSLA '08*, 2008.

32 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

33 Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95*, 1995.

34 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA '10*, 2010.

35 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *ICFP '16*, 2016.

36 Gyunghee Park, Jaemin Hong, Guy L Steele Jr, and Sukyoung Ryu. Polymorphic symmetric multiple dispatch with variance. *POPL '19*, 2019.

37 Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

38 Gordon Plotkin. Lambda-definability and logical relations, 1973.

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

40 Redhat. Ceylon. `https://ceylon-lang.org/`, 2011.

41 John C Reynolds. The coherence of languages with intersection types. In *STACS '91*.

42 John C Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University.

43 John C Reynolds. Preliminary design of the programming language Forsythe. Technical report, 1988.

44 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

45 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *POPL '10*, 2010.

46 Richard Statman. Logical relations and the typed λ-calculus. *Information and control*, 65(2-3):85–97, 1985.

47 William W Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(2):198–212, 1967.

48 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, 1989.

49 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP '09*, 2009.
50 Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops*, 2015.
51 Andrew K Wright, Matthias Felleisen, et al. A syntactic approach to type soundness. *Information and computation*, 115(1), 1994.

A Algorithmic Disjointness

$\boxed{A *_a B}$ *(Algorithmic Disjointness)*

$$\textsc{D-TopL}\ \frac{}{\mathsf{Top} *_a A} \qquad \textsc{D-TopR}\ \frac{}{A *_a \mathsf{Top}} \qquad \textsc{D-AndL}\ \frac{A_1 *_a B \qquad A_2 *_a B}{A_1 \,\&\, A_2 *_a B} \qquad \textsc{D-AndR}\ \frac{A *_a B_1 \qquad A *_a B_2}{A *_a B_1 \,\&\, B_2} \qquad \textsc{D-IntArr}\ \frac{}{\mathsf{Int} *_a A_1 \to A_2}$$

$$\textsc{D-ArrInt}\ \frac{}{A_1 \to A_2 *_a \mathsf{Int}} \qquad \textsc{D-ArrArr}\ \frac{A_2 *_a B_2}{A_1 \to A_2 *_a B_1 \to B_2}$$

B The Full Rules of the Extended Dunfield's Semantics

This appendix presents the full set of rules of extended Dunfield's Semantics which is discussed in Section 5.1.

$\boxed{E \rightsquigarrow E'}$ *(Extended Dunfield's Operational Semantics)*

$$\textsc{DStep-top}\ \frac{}{E \rightsquigarrow \top} \qquad \textsc{DStep-toparr}\ \frac{}{\top \rightsquigarrow \lambda x.\, \top} \qquad \textsc{DStep-appl}\ \frac{E_1 \rightsquigarrow E_1'}{E_1\, E_2 \rightsquigarrow E_1'\, E_2} \qquad \textsc{DStep-appr}\ \frac{E_2 \rightsquigarrow E_2'}{V_1\, E_2 \rightsquigarrow V_1\, E_2'}$$

$$\textsc{DStep-beta}\ \frac{}{(\lambda x.\, E)\, V \rightsquigarrow E[x \mapsto V]} \qquad \textsc{DStep-fix}\ \frac{}{\mathbf{fix}\, x.\, E \rightsquigarrow E[x \mapsto \mathbf{fix}\, x.\, E]} \qquad \textsc{DStep-mergel}\ \frac{E_1 \rightsquigarrow E_1'}{E_1 \,,,\, E_2 \rightsquigarrow E_1' \,,,\, E_2}$$

$$\textsc{DStep-merger}\ \frac{E_2 \rightsquigarrow E_2'}{V_1 \,,,\, E_2 \rightsquigarrow V_1 \,,,\, E_2'} \qquad \textsc{DStep-unmergel}\ \frac{}{E_1 \,,,\, E_2 \rightsquigarrow E_1} \qquad \textsc{DStep-unmerger}\ \frac{}{E_1 \,,,\, E_2 \rightsquigarrow E_2} \qquad \textsc{DStep-split}\ \frac{}{E \rightsquigarrow E \,,,\, E}$$

C The Variant of $\lambda_i^{:}$

This appendix presents the full set of rules of the variant of $\lambda_i^{:}$ which is discussed in Section 6.1.

C.1 Syntax of the Variant of $\lambda_i^{:}$

Type	$A, B ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B$
Expr	$e ::= x \mid i \mid \top \mid e : A \mid e_1\, e_2 \mid \lambda x.\, e : A \to B \mid e_1 \,,,\, e_2 \mid \mathbf{fix}\, x.\, e : A \mid \langle v \rangle$
Value	$v ::= i \mid \top \mid v : \mathsf{Top} \mid \lambda x : A.\, e : C \to D \mid v_1 \,,,\, v_2$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$

C.2 Ordinary Types and Top-like Types in the Variant of $\lambda_i^{:}$

$\boxed{A\ \mathsf{ordinary}}$ *(Ordinary Types in the variant of $\lambda_i^{:}$)*

$$\textsc{O-int}\ \frac{}{\mathsf{Int\ ordinary}} \qquad \textsc{O-arrow}\ \frac{}{A \to B\ \mathsf{ordinary}}$$

$\boxed{\rceil A \lceil}$ *(Top-like Types in the variant of $\lambda_i^{:}$)*

$$\textsc{TL-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{TL-top}\ \frac{}{\rceil \mathsf{Top} \lceil}$$

C.3 Typing Rules of the Variant of $\lambda_i^{:}$

$\boxed{A <: B}$ *(Subtyping of the Variant of $\lambda_i^{:}$)*

$$\textsc{S-z}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{S-top}\ \frac{}{A <: \mathsf{Top}} \qquad \textsc{S-arr}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \textsc{S-andR}\ \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}$$

$$\textsc{S-andL1}\ \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3} \qquad \textsc{S-andL2}\ \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3}$$

$\boxed{A *_a B}$ *(Algorithmic Disjointness in the Variant of $\lambda_i^{:}$)*

$$\textsc{D-topL}\ \frac{}{\mathsf{Top} *_a A} \qquad \textsc{D-topR}\ \frac{}{A *_a \mathsf{Top}} \qquad \textsc{D-andL}\ \frac{A_1 *_a B \qquad A_2 *_a B}{A_1 \,\&\, A_2 *_a B} \qquad \textsc{D-andR}\ \frac{A *_a B_1 \qquad A *_a B_2}{A *_a B_1 \,\&\, B_2} \qquad \textsc{D-IntArr}\ \frac{}{\mathsf{Int} *_a A_1 \to A_2}$$

$$\textsc{D-ArrInt}\ \frac{}{A_1 \to A_2 *_a \mathsf{Int}}$$

$\boxed{\Gamma \vdash e : A}$ *(Typing for Expressions of the Variant of $\lambda_i^{:}$)*

$$\textsc{ExpTyp-top}\ \frac{}{\Gamma \vdash \top : \mathsf{Top}} \qquad \textsc{ExpTyp-lit}\ \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \textsc{ExpTyp-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{ExpTyp-app}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \qquad \textsc{ExpTyp-abs}\ \frac{\Gamma, x : C \vdash e : B}{\Gamma \vdash (\lambda x.\, e : C \to B) : C \to B}$$

$$\textsc{ExpTyp-fix}\ \frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash (\mathbf{fix} x.\, e : A) : A} \qquad \textsc{ExpTyp-anno}\ \frac{\Gamma \vdash e : B \qquad B <: A}{\Gamma \vdash (e : A) : A} \qquad \textsc{ExpTyp-merge}\ \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 : A \,\&\, B} \qquad \textsc{ExpTyp-val}\ \frac{v : A}{\Gamma \vdash \langle v \rangle : A}$$

$\boxed{v : A}$ *(Typing for Values of the Variant of $\lambda_i^{:}$)*

$$\textsc{ValTyp-top}\ \frac{}{\top : \mathsf{Top}} \qquad \textsc{ValTyp-topv}\ \frac{v : A}{(v : \mathsf{Top}) : \mathsf{Top}} \qquad \textsc{ValTyp-lit}\ \frac{}{i : \mathsf{Int}} \qquad \textsc{ValTyp-merge}\ \frac{v_1 : A \quad v_2 : B \qquad v_1 \approx v_2}{v_1 \,,,\, v_2 : A \,\&\, B}$$

$$\textsc{ValTyp-absv}\ \frac{\cdot, x : A \vdash e : B \quad B <: D \qquad C <: A}{(\lambda x : A.\, e : C \to D) : C \to D}$$

C.4 Reduction Rules of the Variant of $\lambda_i^{:}$

$\boxed{v \hookrightarrow_A v'}$ *(Typed Reduction of the Variant of $\lambda_i^{:}$)*

$$\textsc{TRed-lit}\ \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{TRed-top}\ \frac{}{v \hookrightarrow_{\mathsf{Top}} v : \mathsf{Top}} \qquad \textsc{TRed-mergevl}\ \frac{v_1 \hookrightarrow_A v_1' \qquad A \text{ ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_1'} \qquad \textsc{TRed-and}\ \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 ,, v_2}$$

$$\textsc{TRed-mergevr}\ \frac{v_2 \hookrightarrow_A v_2' \qquad A \text{ ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_2'} \qquad \textsc{TRed-arrow}\ \frac{C <: B_1 \qquad B_2 <: D}{\lambda x : A.\, e : B_1 \to B_2 \hookrightarrow_{(C \to D)} \lambda x : A.\, e : C \to D}$$

$\boxed{e \hookrightarrow e'}$ *(Reduction of the Variant of $\lambda_i^{:}$)*

$$\textsc{R-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2} \qquad \textsc{R-appr}\ \frac{e_2 \hookrightarrow e_2'}{\langle v_1 \rangle\, e_2 \hookrightarrow \langle v_1 \rangle\, e_2'} \qquad \textsc{R-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 ,, e_2 \hookrightarrow e_1' ,, e_2} \qquad \textsc{R-merger}\ \frac{e_2 \hookrightarrow e_2'}{\langle v_1 \rangle ,, e_2 \hookrightarrow \langle v_1 \rangle ,, e_2'}$$

$$\textsc{R-fix}\ \frac{}{\mathbf{fix} x.\, e : A \hookrightarrow e[x \mapsto \mathbf{fix} x.\, e : A]} \qquad \textsc{R-beta}\ \frac{v \hookrightarrow_A v'}{(\langle \lambda x : A.\, e : B \to D \rangle)\, \langle v \rangle \hookrightarrow (e[x \mapsto \langle v' \rangle]) : D}$$

$$\textsc{R-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A} \qquad \textsc{R-annov}\ \frac{v \hookrightarrow_A v'}{\langle v \rangle : A \hookrightarrow \langle v' \rangle} \qquad \textsc{R-mergev}\ \frac{}{\langle v_1 \rangle ,, \langle v_2 \rangle \hookrightarrow \langle v_1 ,, v_2 \rangle}$$

$$\textsc{R-abs}\ \frac{}{\lambda x.\, e : A \to D \hookrightarrow \langle \lambda x : A.\, e : A \to D \rangle} \qquad \textsc{R-topv}\ \frac{}{\top \hookrightarrow \langle \top \rangle} \qquad \textsc{R-litv}\ \frac{}{i \hookrightarrow \langle i \rangle}$$