JFP, 13 pages, 2022. © Cambridge University Press 2022
doi:10.1017/xxxxx

THEORETICAL PEARL

Type Soundness with Unrestricted Merges

BABER REHMAN BRUNO C. D. S. OLIVEIRA

The University of Hong Kong
(e-mail: {brehman, bruno}@cs.hku.hk)

Abstract

Dunfield (2014) presented a calculus with union and intersection types in the presence of an unrestricted term-level merge operator. However, the semantics of calculi with an unrestricted merge operator is challenging and creates difficulties for proving type-safety. In this paper we study a variant of Dunfield's calculus with a direct and type sound operational semantics, providing a lightweight framework to study the semantics of calculi with an unrestricted merge operator.

1 Introduction

The merge operator has been studied by various researchers in the literature (Reynolds, 1988; Castagna et al., 1995; Dunfield, 2014; Oliveira et al., 2016; Huang et al., 2021). The main reason of interest for the merge operator is its ability to encode several other language constructs. As Dunfield argued:

> *Designing and implementing typed programming languages is hard. Every new type system feature requires extending the metatheory and implementation, which are often complicated and fragile. To ease this process, we would like to provide general mechanisms that subsume many different features.*

With its ability to encode several other language features, the merge operator provides one such general mechanism. Instead of having to develop calculi to study each individual feature (and interactions with other features), we only need to show encodings of language features in terms of the merge operator. Thus, for many features, no new calculus is needed. The merge operator, together with intersection and union types, is naturally able to encode various advanced programming features such as *nested composition* (Bi et al., 2018), *multi-field records* from single-field records (Reynolds, 1988), and function *overloading* (Castagna et al., 1995) among others. The research programming language CP (Zhang et al., 2021) is an example of a language design that leverages on the merge operator to model complex source-level programming language features. CP enables a high-degree of modularity and reuse, while being built on top of a small core language.

Dunfield (2014) studied a calculus with intersection types, union types, and the merge operator. She defined a direct operational semantics, and has shown that this semantics is

neither type-sound nor deterministic. Nevertheless, the semantics is still useful to understand the behaviour of the merge operator. She then adopted an elaboration semantics into an ordinary λ-calculus with product and sum types. The main idea is to elaborate intersection types to product types, union types to sum types, and merges to pairs. Dunfield proved that the elaboration is type-preserving, thus enabling a proof of type-safety for a calculus with an unrestricted merge operator. Central to Dunfield's result is the fact that the elaboration is *type-directed*, thus being able to access and use typing information.

More recently, Huang et al. (2021) proposed a variant of operational semantics called Type-Directed Operational Semantics (TDOS). TDOS can directly model the semantics of languages with a merge operator, and enables both type-soundness and determinism. In a TDOS type annotations are needed to guide reduction, and reduction is type-dependent, which is key to prove type-soundness. However, Huang et al.'s work considered only a restricted version of the merge operator, and did not account for union types. The restrictions imposed by Huang et al. complicate the semantics and the metatheory, to enable determinism. Furthermore, they also preclude some applications, such as overloading. Thus it is not clear yet that this set of restrictions is the most adequate for calculi with the merge operator. Since the key argument for the merge operator is its general purpose nature and ability to encode other features, restrictions should be minimal to avoid precluding applications (Dunfield, 2014). Thus the study of calculi with unrestricted merges remains relevant, as such calculi remain open to other (potentially better) restrictions.

In this pearl we propose a direct and type-sound operational semantics for a variant of Dunfield's calculus. We employ a TDOS, but without imposing any restrictions on the merge operator. As such the calculus is also not deterministic, but it retains the general purpose nature of Dunfield's original calculus. Moreover, the calculus is considerably simplified compared to Huang et al.'s approach, because many artifacts used in their semantics for obtaining determinism are not needed. Thus, while our work does not present novel techniques, it provides a clean and simple framework to study the semantics of calculi with an unrestricted merge operator. Unlike Dunfield's work, our calculus can be understood independently from a target calculus and elaboration, while still achieving a type-safety proof. All the metatheory of this paper has been formalized in Coq theorem prover and is available at: `https://github.com/baberrehman/jfp23-artifact`.

2 Background: Dunfield's Calculus

We start by reviewing a variant of Dunfield (2014)'s calculus and its direct operational semantics, and illustrate how it lacks type preservation. We employ some minor modifications to Dunfield's calculus that we document and justify along the way.

Syntax and subtyping. The syntax for Dunfield's calculus is shown in Figure 1. Types consist of the top type $\top$, bottom type $\bot$, integers (Int), function types ($A \to B$), and intersection ($A \wedge B$) and union ($A \vee B$) types. Note that the type $\bot$ has not been studied in Dunfield's original calculus. Expressions consist of variables (x), integer literals (i), abstractions ($\lambda x.\ e$), applications ($e_1\ e_2$), the merge operator (e_1,,e_2) and a fixpoint operator ($\mathsf{fix}\ x.e$). We also add an explicit switch expression for union elimination. Similarly,

Type	A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B$
Expr	e	$::=$	$x \mid i \mid \lambda x.\, e \mid e_1\, e_2 \mid e_1 ,, e_2 \mid \mathsf{fix}\, x.e \mid \mathsf{switch}\ e\ \{x \to e_1, y \to e_2\}$
Value	v	$::=$	$x \mid i \mid \lambda x.\, e \mid v_1 ,, v_2$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

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

$$\text{S-TOP}\ \frac{}{A <: \top} \qquad \text{S-INT}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \text{S-ARROW}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \text{S-BOT}\ \frac{}{\bot <: A} \qquad \text{S-ORA}\ \frac{A <: C \qquad B <: C}{A \vee B <: C}$$

$$\text{S-ORB}\ \frac{A <: B}{A <: B \vee C} \qquad \text{S-ORC}\ \frac{A <: C}{A <: B \vee C} \qquad \text{S-ANDA}\ \frac{A <: B \qquad A <: C}{A <: B \wedge C} \qquad \text{S-ANDB}\ \frac{A <: C}{A \wedge B <: C} \qquad \text{S-ANDC}\ \frac{B <: C}{A \wedge B <: C}$$

Fig. 1. Source syntax of Dunfield's calculus (with a switch expression), and subtyping.

variables, literals, abstractions and a merge of values constitute values. The typing context is standard. Note that we do not adopt a notion of evaluation contexts for union elimination as in Dunfield's original formalization. Instead, we use an explicit switch expression ($\mathsf{switch}\ e\ \{x \to e_1, y \to e_2\}$). We choose using a switch expression because it simplifies parts of the metatheory, while the calculus retains at least the same expressive power. The subtyping relation is standard for a calculus with intersection and union types.

Type System. Figure 2 shows the type system. The reader can ignore the gray parts in the rules, which are discussed in Section 4. Rules DTYP-INT, DTYP-VAR, DTYP-APP, and DTYP-ABS are standard rules for integers, variables, applications, and abstractions. Similarly, rule DTYP-SUB is the standard subsumption rule. Rules DTYP-MERGA and DTYP-MERGB type-check the merge operator. They express the idea that a merge can have the type of either one of its components. An important point to note about these rules is that they do not impose any type restriction in the other component. For instance, in rule DTYP-MERGA, e_2 is completely unrestricted (it could be ill-typed, for instance). Rule DTYP-AND is the standard introduction rule for intersection types (Coppo et al., 1981).

Rules DTYP-ANDL and DTYP-ANDR are for intersection elimination, and rules DTYP-ORL and DTYP-ORR are for union introduction. Note that those rules can be subsumed by the subsumption rule. Rule DTYP-FIX type-checks fixpoints. Interested readers may refer to the original paper (Dunfield, 2014) for more details. Finally, the rule DTYP-SWITCH is for union elimination. Dunfield's original formalization uses a notion of evaluation context for union elimination. We avoid such form of union elimination because it complicates proofs of inversion lemmas. However, we prove that the type system presented in Figure 2 is complete with respect to Dunfield's original type system. We also prove that the dynamic semantics shown in Figure 3 is complete with respect to Dunfield's original semantics. Both lemmas are straightforward and part of the Coq formalization of this paper.

Operational Semantics. Figure 3 shows the small-step operational semantics. Rules DSTEP-APPL, DSTEP-APPR, DSTEP-BETA, and DSTEP-FIX are standard reduction rules for applications, beta-reduction, and fixpoints. The remaining rules are for the

$\boxed{\Gamma \vdash_d e : A \rightsquigarrow e'}$ *(Dunfield's type system with elaboration to λ_m)*

DTYP-INT
$$\frac{}{\Gamma \vdash_d i : \mathsf{Int} \rightsquigarrow i}$$

DTYP-SUB
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e' \qquad A <: B}{\Gamma \vdash_d e : B \rightsquigarrow e'}$$

DTYP-VAR
$$\frac{x : A \in \Gamma}{\Gamma \vdash_d x : A \rightsquigarrow x}$$

DTYP-APP
$$\frac{\Gamma \vdash_d e_1 : A \to B \rightsquigarrow e_1' \qquad \Gamma \vdash_d e_2 : A \rightsquigarrow e_2'}{\Gamma \vdash_d e_1\ e_2 : B \rightsquigarrow e_1'\ e_2'}$$

DTYP-ABS
$$\frac{\Gamma, x : A \vdash_d e : B \rightsquigarrow e'}{\Gamma \vdash_d \lambda x.e : A \to B \rightsquigarrow \lambda x.e' : A \to B}$$

DTYP-MERGA
$$\frac{\Gamma \vdash_d e_1 : A \rightsquigarrow e_1'}{\Gamma \vdash_d e_1 ,, e_2 : A \rightsquigarrow e_1'}$$

DTYP-MERGB
$$\frac{\Gamma \vdash_d e_2 : B \rightsquigarrow e_2'}{\Gamma \vdash_d e_1 ,, e_2 : B \rightsquigarrow e_2'}$$

DTYP-AND
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e_1 \qquad \Gamma \vdash_d e : B \rightsquigarrow e_2}{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e_1 ,, e_2}$$

DTYP-ANDL
$$\frac{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e'}{\Gamma \vdash_d e : A \rightsquigarrow e'}$$

DTYP-ANDR
$$\frac{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e'}{\Gamma \vdash_d e : B \rightsquigarrow e'}$$

DTYP-ORL
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e'}{\Gamma \vdash_d e : A \vee B \rightsquigarrow e'}$$

DTYP-ORR
$$\frac{\Gamma \vdash_d e : B \rightsquigarrow e'}{\Gamma \vdash_d e : A \vee B \rightsquigarrow e'}$$

DTYP-FIX
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e'}{\Gamma \vdash_d \mathsf{fix}\, x.e : A \rightsquigarrow \mathsf{fix}\, x.e' : A}$$

DTYP-SWITCH
$$\frac{\Gamma \vdash_d e : A \vee B \rightsquigarrow e' \qquad \Gamma, x : A \vdash_d e_1 : C \rightsquigarrow e_1' \qquad \Gamma, y : B \vdash_d e_2 : C \rightsquigarrow e_2'}{\Gamma \vdash_d \mathsf{switch}\ e\ \{x \to e_1, y \to e_2\} : C \rightsquigarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1', (y : B) \to e_2'\}}$$

Fig. 2. Source syntax and source typing of Dunfield's calculus.

merge operator and deserve more explanation. Rules DSTEP-UNMERGL and DSTEP-UNMERGR select a component from the merge operator. Rules DSTEP-MERGL and DSTEP-MERGR reduce the merge operator if either the left or the right component reduces. Rule DSTEP-SPLIT constructs a merge of an expression. Finally, rules DSTEP-SWITCHL, DSTEP-SWITCHR, and DSTEP-SWITCH are reduction rules for switch expressions, and are not included in Dunfield's original semantics.

Absence of type-preservation. Dunfield notes that her small-step semantics is not type preserving nor deterministic. We illustrate the lack of type-preservation with a simple example. If we pass a merge such as 1,,true to the successor function (succ) on integers, it may generate ill-typed terms during reduction. For example, a ill-typed derivation is:

$$\dfrac{\text{DSTEP-APPR}\ \dfrac{\text{DSTEP-UNMERGR}\ \dfrac{}{1 ,, \mathsf{true} \longrightarrow_d \mathsf{true}}}{\mathsf{succ}\ (1 ,, \mathsf{true}) \longrightarrow_d \mathsf{succ}\ \mathsf{true}}}{}$$

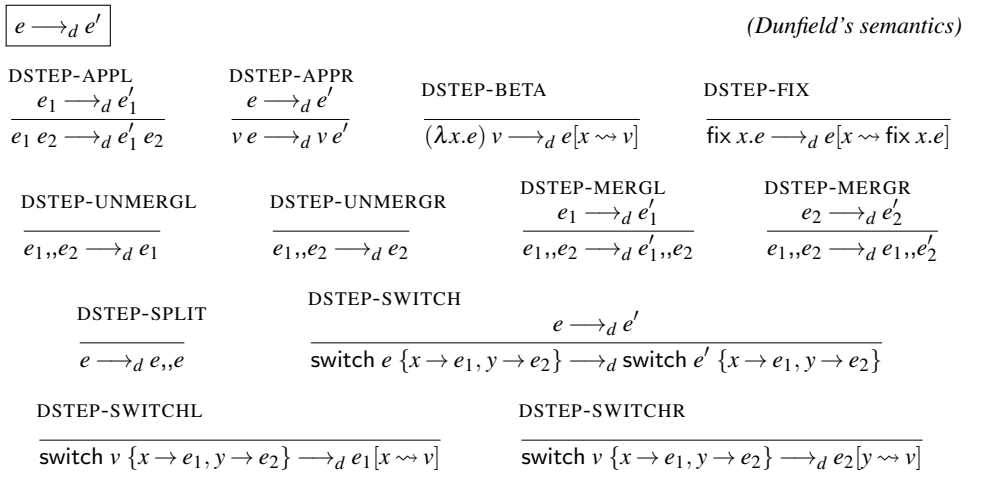

Fig. 3. Dynamic semantics of Dunfield's calculus.

The expression succ true is clearly not well-typed, illustrating the lack of type preservation. Note that the reason why this derivation is allowed is because when reducing a merge there are multiple (non-deterministic) choices. It is possible to select 1 using rule DSTEP-UNMERGL, which would lead to a type-preserving derivation. But it is also possible to select true using rule DSTEP-UNMERGR, leading to the ill-typed derivation above.

In Dunfield's work the small-step semantics is used to justify the elaboration semantics, which is the main result in her work. She shows that her calculus is type-safe via a type-preserving translation into an ordinary λ-calculus with pairs and product types. One of her results is that executions of elaborated programs correspond to (type-preserving) executions in her small-step semantics. Unfortunately, her small-step semantics does allow other executions that are not type-preserving and, consequently, conventional results such as type-soundness cannot be established.

3 A Type-Sound Calculus with a Type-Directed Operational Semantics

In this section we present a variant of Dunfield's calculus called λ_m. The small-step semantics of λ_m is type-sound. To achieve type-soundness we adopt a TDOS (Huang et al., 2021). Unlike Huang et al.'s work, we use a standard type-assignment type system instead of bidirectional type-checking, do not impose a disjointness restriction, and include union types. With a TDOS, reduction can use type information available from type annotations to choose how to reduce merges, avoiding ill-typed derivations during reduction.

3.1 Syntax and Type System

Syntax. Figure 4 shows the syntax and typing for λ_m. Most syntax simply follows Dunfield's original syntax, and differences are highlighted in gray color. Lambda expressions are annotated with input and output types i.e. $\lambda x.\ e : A \to B$. The fixpoint operator ($\mathsf{fix}\ x.e : A$) is also annotated, and the switch expression carries types for branch selection.

Type	A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B$
Expr	e	$::=$	$x \mid i \mid \lambda x.\, e : A \to B \mid e_1\ e_2 \mid e_1 ,, e_2 \mid \mathsf{fix}\, x.e : A \mid$ $\mathsf{switch}\ e\ \{(x:A) \to e_1, (y:B) \to e_2\}$
Value	v	$::=$	$i \mid \lambda x.\, e : A \to B \mid v_1 ,, v_2$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

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

$$\text{T-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{T-INT}\ \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \text{T-SUB}\ \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B} \qquad \text{T-APP}\ \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\text{T-ABS}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x.e : A \to B) : A \to B} \qquad \text{T-MERG}\ \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 ,, e_2 : A \wedge B} \qquad \text{T-FIX}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{fix}\, x.e : A) : A}$$

$$\text{T-SWITCH}\ \frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{switch}\ e\ \{(x:A) \to e_1, (y:B) \to e_2\} : C}$$

Fig. 4. Syntax and type system for λ_m.

Type System. The rules T-VAR, T-INT, T-SUB, and T-APP are same as in Dunfield's calculus. The notable differences are in rules T-ABS, T-MERG, and T-FIX. Rule T-ABS works with type annotated lambda expressions. Rule TYP-FIX is a standard typing rule for the fixpoint operator, but is annotated with the type of the expression. Another difference to Dunfield's type system lies in the treatment of typing for merges. Contrary to Dunfield's merge rules, the rule T-MERG does not allow ill-typed expressions inside a merge. In Section 4, we discuss this difference further, and also show that our rules do not limit expressiveness.

3.2 Casting

Casting lies at the core of calculi with a TDOS for the merge operator (Huang et al., 2021). Generally speaking, casting makes a value consistent with the type under which that value is cast. For example, when an expression $1,,\mathsf{true}$ is cast under type Int it gives 1:

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int}} 1 \quad \text{(applying rule CST-MERGL)}$$

Conversely, the same expression would result in true if cast under type Bool. In summary, casting enables extracting the value of a specific type from the merge operator. In Dunfield's calculus there is no casting relation because her semantics is untyped and there is no type information available during reduction.

Casting relation. The casting relation is shown in Figure 5. The relation $v \longrightarrow_A v'$ shows casting of a value v under type A to another value v'. Note that casting is only applicable to values. We explain the rules next.

When casting a value under $\top$ type it results in same value as stated in rule CST-TOP. Casting an integer under Int returns the same integer. Casting rules for union types are

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

$$\frac{}{\mathsf{Ord}\ \mathsf{Int}}\ \text{ORD-INT} \qquad \frac{}{\mathsf{Ord}\ A \to B}\ \text{ORD-ARROW}$$

$\boxed{v \longrightarrow_A v'}$ *(Type Casting)*

$$\frac{}{v \longrightarrow_\top v}\ \text{CST-TOP} \qquad \frac{}{i \longrightarrow_{\mathsf{Int}} i}\ \text{CST-INT} \qquad \frac{v \longrightarrow_A v'}{v \longrightarrow_{A \vee B} v'}\ \text{CST-ORL} \qquad \frac{v \longrightarrow_B v'}{v \longrightarrow_{A \vee B} v'}\ \text{CST-ORR} \qquad \frac{v \longrightarrow_A v_1 \qquad v \longrightarrow_B v_2}{v \longrightarrow_{A \wedge B} v_1 ,, v_2}\ \text{CST-MERG}$$

$$\frac{A_1 \to B_1 <: A_2 \to B_2}{\lambda x.e : A_1 \to B_1 \longrightarrow_{A_2 \to B_2} \lambda x.e : A_1 \to B_1}\ \text{CST-ARROW}$$

$$\frac{\mathsf{Ord}\ A \qquad v_1 \longrightarrow_A v_1'}{v_1 ,, v_2 \longrightarrow_A v_1'}\ \text{CST-MERGL} \qquad \frac{\mathsf{Ord}\ A \qquad v_2 \longrightarrow_A v_2'}{v_1 ,, v_2 \longrightarrow_A v_2'}\ \text{CST-MERGR}$$

Fig. 5. Type casting for λ_m.

interesting. If an expression casts under a part of a union type then that expression casts under the whole union type as stated in rules CST-ORL and CST-ORR. Casting a lambda expression under a function type returns the same lambda expression as stated in rule CST-ARROW. A merge casts under an ordinary type if either component of the merge casts under that ordinary type (rules CST-MERGL and CST-MERGL). Casting a value under an intersection type results in a merge (rule CST-MERG). Type casting has type preservation and progress properties:

Theorem 1 (Casting Preservation)**.** *If* $\cdot \vdash v : A$ *and* $v \longrightarrow_A v'$ *then* $\Gamma \vdash v' : A$.
Theorem 2 (Casting Progress)**.** *If* $\cdot \vdash v : A$ *then* $v \longrightarrow_A v'$.

Theorems 1 and 2 are vital in proving the type-soundness of the calculus. Notice that the casting relation is non-deterministic, which results in non-deterministic semantics. For illustration purposes, consider what will be the result of casting 1,,true under the type $\mathsf{Int} \vee \mathsf{Bool}$. It can either result in 1 or true depending on the casting rule that we apply:

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int} \vee \mathsf{Bool}} 1 \quad \text{(applying rule CST-ORL)}$$
$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int} \vee \mathsf{Bool}} \mathsf{true} \quad \text{(applying rule CST-ORR)}$$

Despite its non-determinism, the use of casting in our small-step semantics makes the semantics "more" deterministic than in Dunfield's calculus. Unlike Dunfield's semantics, only well-typed casts and reductions are allowed.

3.3 Operational semantics

The operational semantics for λ_m is shown in Figure 6. Rules STEP-APPL and STEP-APPR are standard reduction rules. Rule STEP-BETA is the beta reduction rule. This rule first casts the argument under the input type and then substitutes the argument in the body of the lambda expression. Casting drops the unnecessary part from the argument and makes the input value consistent with the input type of the applied function. For example:

$\boxed{e \longrightarrow e'}$ *(Small-step operational semantics)*

$$\textsc{step-appl}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \textsc{step-dispatch}\ \frac{(v_1,,v_2) \rhd v \longrightarrow e'}{(v_1,,v_2)\ v \longrightarrow e'} \qquad \textsc{step-mergl}\ \frac{e_1 \longrightarrow e_1'}{e_1,,e_2 \longrightarrow e_1',,e_2}$$

$$\textsc{step-switchl}\ \frac{v \longrightarrow_A v'}{\mathsf{switch}\ v\ \{(x:A) \to e_1, (y:B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v']} \qquad \textsc{step-beta}\ \frac{v \longrightarrow_A v'}{(\lambda x.e : A \to B)\ v \longrightarrow e[x \rightsquigarrow v']}$$

$$\textsc{step-switchr}\ \frac{v \longrightarrow_B v'}{\mathsf{switch}\ v\ \{(x:A) \to e_1, (y:B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v']} \qquad \textsc{step-fix}\ \frac{}{\mathsf{fix}\ x.e : A \longrightarrow e[x \rightsquigarrow \mathsf{fix}\ x.e : A]}$$

$$\textsc{step-switch}\ \frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x:A) \to e_1, (y:B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x:A) \to e_1, (y:B) \to e_2\}} \qquad \textsc{step-mergr}\ \frac{e \longrightarrow e'}{v,,e \longrightarrow v,,e'}$$

Fig. 6. Operational semantics for λ_m.

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

A merge of 1,,false is the input being passed to the lambda expression annotated with an input type of Int. Since the dynamic type of 1,,false is $\mathsf{Int} \wedge \mathsf{Bool}$, which is a subtype of Int, the above application type-checks. However, before passing 1,,false to the lambda body, we cast it under the input type (Int) to make the input value and the input type consistent. Rules STEP-MERGL and STEP-MERGR reduce the left and right part of the merge operator. The rules STEP-SWITCHL and STEP-SWITCHR are interesting. Casting is essential in those rules to ensure that the correct branch is selected. Finally, rule STEP-FIX is a standard reduction rule for fixpoints. It replaces x in the body of the fixpoint in itself. The rule STEP-DISPATCH requires detailed explanation and is discussed next.

3.4 Applicative Dispatching

Sometimes lambda expressions appear inside a merge. This enables a merge to be used as a function in applications. Therefore, lambda expressions need to be extracted from the merge and then applied to arguments. The rule STEP-DISPATCH deals with such cases by employing another relation called applicative dispatch (we use the same terminology as Xue et al. (2022)). The applicative dispatching relation is shown in the upper part of Figure 7. Applicative dispatch selects the appropriate function for application from the merge depending upon the argument type. Moreover, it enables function overloading.

Applicative dispatch relation. Rules ADP-MLEFT and ADP-MRIGHT apply either the left or right part of a merge to the argument. There are two premises. One premise states that the dynamic type of argument is a subtype of the input type of one part of merge. While the other premise states that the dynamic type of the argument is not a subtype of the input type of the other part of the merge. The input type relation returns the input type of a functional

$$\boxed{(v_1 ,, v_2) \rhd v \longrightarrow e'} \qquad \textit{(Applicative dispatch)}$$

$$\text{APD-MLEFT}\ \frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda} \qquad \neg(\lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda})}{(v_1 ,, v_2) \rhd v \longrightarrow v_1\ v} \qquad \text{APD-MRIGHT}\ \frac{\neg(\lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda}) \qquad \lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda}}{(v_1 ,, v_2) \rhd v \longrightarrow v_2\ v}$$

$$\text{APD-BOTH}\ \frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda} \qquad \lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda}}{(v_1 ,, v_2) \rhd v \longrightarrow v_1\ v ,, v_2\ v}$$

Dynamic Type $\lfloor v \rfloor$

$\lfloor i \rfloor$	=	Int
$\lfloor \lambda x.e : A \to B \rfloor$	=	$A \to B$
$\lfloor v_1 ,, v_2 \rfloor$	=	$\lfloor v_1 \rfloor \wedge \lfloor v_2 \rfloor$

Input Type $\lfloor v \rfloor^{\lambda}$

$\lfloor \lambda x.e : A \to B \rfloor^{\lambda}$	=	A
$\lfloor v_1 ,, v_2 \rfloor^{\lambda}$	=	$\lfloor v_1 \rfloor^{\lambda} \vee \lfloor v_2 \rfloor^{\lambda}$
$\lfloor i \rfloor^{\lambda}$	=	$\bot$

Fig. 7. Dynamic dispatch, dynamic type and input type relation for λ_m.

value. The functional value is either a lambda expression, or a merge containing at least one lambda expression. The first case of the input type function deals with lambda expressions. In this case the input type simply returns the input type of the given lambda expression. The second case deals with functional merges. In this case the output is the union of the input types of all the lambda expressions in the merge. The other values cannot appear on the left side of a valid application. Therefore, they cannot have an input type. We simply return $\bot$ in such cases. To illustrate the rules, consider the following application:

$$((\lambda x.\mathsf{succ}\ x : \mathsf{Int} \to \mathsf{Int}) ,, (\lambda x.\mathsf{not}\ x : \mathsf{Bool} \to \mathsf{Bool}) ,, \mathsf{true})\ 1$$

The above application is valid and type-checks. We need to extract $\lambda x.\mathsf{succ}\ x : \mathsf{Int} \to \mathsf{Int}$ from this merge and then apply it to argument 1. Note that it is essential to choose $\lambda x.\mathsf{succ}\ x : \mathsf{Int} \to \mathsf{Int}$ in this case. The calculus will not be type preserving otherwise. This is due to the fact that $(\mathsf{true}\ 1)$ is not a valid application and neither is $(\lambda x.\mathsf{not}\ x : \mathsf{Bool} \to \mathsf{Bool})\ 1)$. Therefore, special care needs to be taken to choose the appropriate function for applications. Applicative dispatch compares the argument type with the input type of each expression in the merge. It then applies all of the possible matching functions. The partial derivation for $(\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int} ,, \lambda x.\mathsf{not}\ x : \mathsf{Bool} \to \mathsf{Bool} ,, \mathsf{true})\ 1$ is shown below:

$$\text{STEP-DISPATCH}\ \frac{\text{APD-MLEFT}\ \dfrac{\mathsf{Int} <: \mathsf{Int} \vee \mathsf{Bool} \qquad \neg(\mathsf{Int} <: \bot)}{(\lambda x.\mathsf{succ}\ x ,, \lambda x.\mathsf{not}\ x ,, \mathsf{true}) \rhd 1 \longrightarrow ((\lambda x.\mathsf{succ}\ x) ,, (\lambda x.\mathsf{not}\ x))\ 1}}{(\lambda x.\mathsf{succ}\ x ,, \lambda x.\mathsf{not}\ x ,, \mathsf{true})\ 1 \longrightarrow ((\lambda x.\mathsf{succ}\ x) ,, (\lambda x.\mathsf{not}\ x))\ 1}$$

We omit type annotations in the derivation for space reasons. The expression $((\lambda x.\mathsf{succ}\ x) ,, (\lambda x.\mathsf{not}\ x))\ 1$ further reduces to $(\lambda x.\mathsf{succ}\ x)\ 1$. It is evident that the only function that can be applied to 1 is the first one. Finally $(\lambda x.\mathsf{succ}\ x)\ 1$ becomes a standard application with a lambda expression and reduces using beta reduction.

Rule ADP-BOTH is interesting. It applies both parts of the merge to the argument and returns a merge of the outputs of both functions. This rule is applicable in cases where the

dynamic type of the argument is a subtype of both parts of the merge. Both the succ and pred functions are applicable to 1 in the following application, resulting in 2,,0.

$$((\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}),,(\mathsf{pred} : \mathsf{Int} \to \mathsf{Int}))\ 1$$

Applicative dispatching and partial applications. In the previous example, it seems that a simpler alternative design choice for applicative dispatching is sufficient. An idea is to simply have rules APD-MLEFT and APD-MRIGHT, while dropping the *non subtyping* premises and rule APD-BOTH. However, such design, where we non-deterministically select an applicable function, does not work. The issue of such design is visible with partial applications of overloaded functions. Consider the following application:

$$(\lambda x.\ \lambda y.\ \mathsf{succ}\ y : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int},,\lambda x.\ \lambda y.\ \mathsf{not}\ y : \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Bool})\ 1\ \mathsf{true}$$

Note that the merge consists of two functions in the example above. Importantly, the input type of both of the functions is Int. The first argument in the application is 1 with (dynamic) type Int. With non-deterministic selection of a function, we could end up in a stuck expression later on. For instance if we selected the first function above, then we would get stuck when applying the resulting function $(\lambda y.\ \mathsf{succ}\ y : \mathsf{Int} \to \mathsf{Int})$ to true (since the argument is of the wrong type). With our design, both functions are selected in the first application to 1 using rule APD-BOTH. Then, on the second application, we select the second function in the merge. Thus the rule APD-BOTH avoids getting into a stuck (and ill-typed) state during reduction, by not committing too early to selecting a function.

Metatheory. Lemma 3 states that if the dynamic type of value v_1 is a subtype of the input type of v (i.e. $\lfloor v_1 \rfloor <: \lfloor v \rfloor^{\lambda}$) then v must contain a function and the value v_1 is (type) compatible with that function. That is the type of v checks against a function type where the dynamic type of v_1 is the input type of the function. Lemma 3 is essential for the type preservation of λ_m. Similarly, Lemma 4 is essential for progress. It states that a well-typed application of the merge operator must take a step.

Lemma 3 (Applicative dispatch compatibility)**.** *If* $\cdot \vdash v : A$ *and* $\lfloor v_1 \rfloor <: \lfloor v \rfloor^{\lambda}$ *then* $\cdot \vdash v : \lfloor v_1 \rfloor \to \top$

Lemma 4 (Applicative dispatch progress)**.** *If* $\cdot \vdash v_1,,v_2 : A \to B$ *and* $\cdot \vdash v : A$ *then* $\exists e'$ $(v_1,,v_2) \rhd v \longrightarrow e'$.

3.5 Type Soundness of λ_m

The standard properties of type preservation and progress hold in λ_m. Type preservation (Theorem 5) states that the types are preserved during reduction. Progress (Theorem 6) states that a well-typed expression is either a value or it reduces. Note that the operational semantics is type dependent i.e. operational semantics depends upon the casting relation. Therefore, type preservation depends upon type casting preservation (Theorem 1), and progress depends on type casting progress (Theorem 2).

Theorem 5 (Type Preservation). *If* $\cdot \vdash e : A$ *and* $e \longrightarrow e'$ *then* $\cdot \vdash e' : A$.

Theorem 6 (Progress). *If* $\cdot \vdash e : A$ *then either* e *is a value; or* $e \longrightarrow e'$ *for some* e'.

4 Relating λ_m and Dunfield's Calculus

This section presents several results relating λ_m's type system and semantics to those in the variant of Dunfield's calculus presented in Section 2. An important remark is that λ_m's dynamic semantics cannot be complete with respect Dunfield's small-step semantics. The semantics of her calculus allows ill-typed reductions, while λ_m's semantics does not.

4.1 Soundness and Completeness with Respect to Dunfield's Type System

The type system of λ_m is sound and complete with respect to Dunfield's type system, meaning that all the programs that type-check in Dunfield's type system, can also be encoded via an elaboration in λ_m.

Elaboration to λ_m. The gray parts of the rules in Figure 2 show the elaborated terms in λ_m. Most parts of the elaboration are straightforward, simply elaborating an expression in Dunfield's calculus to the same kind of expression in λ_m. Rule DTYP-ABS elaborates an unannotated lambda expression from Dunfield's calculus to a type annotated lambda expression in λ_m. Rule DTYP-AND is also interesting. It elaborates an expression that checks against an intersection of two types into a merge of the two elaborated expressions. The case of unannotated lambda expressions is of particular interest in combination with rule DTYP-AND. Dunfield's calculus has unannotated lambda expressions and the typing rule DTYP-AND is able to encode the following program:

$$\text{DTYP-AND}\ \frac{\cdot \vdash \lambda x.x : \mathsf{Int} \to \mathsf{Int} \qquad \cdot \vdash \lambda x.x : \mathsf{Bool} \to \mathsf{Bool}}{\cdot \vdash \lambda x.x : \mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}}$$

Whereas λ_m cannot encode the above program directly. Therefore, we elaborate this program into a program in λ_m using a merge, in order to preserve the completeness with respect to Dunfield's type system. The elaboration for such a program is shown next:

$$\frac{}{\cdot \vdash \lambda x.x : \mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool} \rightsquigarrow (\lambda x.x : \mathsf{Int} \to \mathsf{Int}),,(\lambda x.x : \mathsf{Bool} \to \mathsf{Bool})}$$

The expression $\lambda x.x$ type-checks against $\mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}$ in Dunfield's type system and elaborates to $(\lambda x.x : \mathsf{Int} \to \mathsf{Int}),,(\lambda x.x : \mathsf{Bool} \to \mathsf{Bool})$ in λ_m.

A note on Dunfield's type system. Besides allowing ill-typed reductions, Dunfield's type system allows certain ill-typed parts of the programs. For example, the program $1,,\mathsf{succ}\ \mathsf{true}$, type-checks following the rule DTYP-MERGA. Importantly, a part of this program ($\mathsf{succ}\ \mathsf{true}$) is ill-typed. Such ill-typed parts of the programs have no practical purpose and are not essential. Indeed, in her elaboration, Dunfield simply discards such untyped expressions, which never get elaborated. Therefore, we do not account for such ill-typed

$$
\begin{array}{rcl}
|i| & = & i \\
|x| & = & x \\
|\lambda x.e : A \to B| & = & \lambda x.|e| \\
|e_1\ e_2| & = & |e_1|\ |e_2| \\
|e_1 ,, e_2| & = & |e_1| ,, |e_2| \\
|\mathsf{fix}\ x.e : A| & = & \mathsf{fix}\ x.|e| \\
|\mathsf{switch}\ e\ \{(x:A) \to e_1, (y:B) \to e_2\}| & = & \mathsf{switch}\ |e|\ \{x \to |e_1|, y \to |e_2|\}
\end{array}
$$

Fig. 8. Type Erasure function.

expressions in our calculus either. We prove completeness of the type system upto the point where all (sub)expressions of a program are well-typed.

Soundness and completeness. Lemma 7 states that if an expression e type-checks in Dunfield's type system with type A and elaborates to e' in λ_m, then e' has type A in λ_m. Lemma 8 states that if an expression e type-checks in λ_m against type A, then after erasure the expression e type checks against A in Dunfield's system. The erasure function, which simply drops type annotations, is presented in Figure 8.

Lemma 7 (Completeness Lemma)**.** *If* $\Gamma \vdash_d e : A \rightsquigarrow e'$ *then* $\Gamma \vdash e' : A$.

Lemma 8 (Soundness Lemma)**.** *If* $\Gamma \vdash e : A$ *then* $\Gamma \vdash_d |e| : A$.

4.2 Soundness with Respect to Dunfield's Semantics

The direct operational semantics of λ_m is sound with respect to Dunfield's dynamic semantics (Lemma 9). Dunfield's semantics is shown in Figure 3 and the semantics for λ_m is shown in Figure 6. In our soundness result we also employ the type erasure ($|e|$) shown in Figure 8. The completeness to Dunfield's dynamic semantics does not hold. Note that the soundness holds for a multi-step relation, meaning that a single step in λ_m corresponds to zero, one, or more steps in Dunfield's calculus.

Lemma 9 (Soundness of semantics)**.** *If* $e \longrightarrow e'$ *then* $|e| \longrightarrow_d^* |e'|$.

5 Discussion and Conclusion

We propose a type-sound semantics for a variant of the calculus studied by Dunfield (2014). Calculi with an unrestricted merge operator are important due to their general purpose nature and ability to encode several other language constructs. The direct operational semantics adopted in this paper reduces the metatheoretical complexity compared to the elaboration approach by Dunfield, since there is no need for a target language, and the semantics of programs can be directly understood. We argue that, among other things, our semantics is valuable for studying possible extensions with other features, and the interactions of the merge operator with such extensions. Nonetheless, we view the TDOS and elaboration approaches as complementary. Both approaches provide valuable insights

about the merge operator and they can be used for different purposes. For instance, the elaboration semantics suggests an easy way to obtain efficient implementations of languages with the merge operator via an elaboration to a standard target programming language. While the TDOS approach is implementable, its direct implementation is not efficient.

This pearl does not provide novel techniques with respect to Huang et al. (2021)'s TDOS. However, our work is free from several artifacts imposed by Huang et al.'s restrictions to obtain a deterministic semantics. These artifacts complicate the semantics considerably. Huang et al. (2021)'s calculi employ type disjointness to obtain determinism. Only merges with disjoint types are allowed, which prevent applications such as overloading. Union types are also not supported. They also have to adopt bidirectional typing, as a type assignment system would lead to other sources of non-determinism. Finally, a notion of consistency for values is needed to prove determinism. In our work we can avoid all that since we do not impose restrictions on merges and we do not aim to have determinism.

Closest to us is the work by Xue et al. (2022), which also considers a TDOS with unrestricted merges, but without union types. However, Xue et al.'s work is still focused on algorithmic aspects and it retains several techniques from Huang et al.'s approach, including bidirectional typing, which complicate the semantics. The absence of union types and their elimination constructs (switches) means that the calculus lacks some expressive power in Dunfield's calculus. The addition of union types simplifies applicative dispatching considerably: union types can be used to provide the input type of merged functions.

In conclusion, this work provides a simple calculus with unrestricted merges and comparable expressive power to Dunfield's original calculus. However, unlike Dunfield's original calculus, λ_m has a type-sound direct operational semantics.

References

Bi, X., Oliveira, B. C. d. S. & Schrijvers, T. (2018) The Essence of Nested Composition. European Conference on Object-Oriented Programming (ECOOP).

Castagna, G., Ghelli, G. & Longo, G. (1995) A calculus for overloaded functions with subtyping. *Information and Computation*. **117**(1), 115–135.

Coppo, M., Dezani-Ciancaglini, M. & Venneri, B. (1981) Functional characters of solvable terms. *Mathematical Logic Quarterly*. **27**(2-6), 45–58.

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

Huang, X., Zhao, J. & Oliveira, B. C. d. S. (2021) Taming the merge operator. *Journal of Functional Programming*. **31**, e28.

Oliveira, B. C. d. S., Shi, Z. & Alpuim, J. (2016) Disjoint intersection types. Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 364–377.

Reynolds, J. C. (1988) Preliminary design of the programming language forsythe. Technical Report CMU-CS-88-159. Carnegie Mellon University.

Xue, X., Oliveira, B. C. d. S. & Xie, N. (2022) Applicative intersection types. Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings. Springer. pp. 155–174.

Zhang, W., Sun, Y. & Oliveira, B. C. (2021) Compositional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. **43**(3), 1–61.