Intrinsically-typed definitional interpreters à la carte

van der Rest, C.R.; Poulsen, C.B.; Rouvoet, A.J.; Visser, Eelco; Mosses, P.D.

DOI
10.1145/3563355

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the ACM on Programming Languages

Intrinsically-Typed Definitional Interpreters à la Carte

CAS VAN DER REST, Delft University of Technology, The Netherlands
CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
EELCO VISSER†, Delft University of Technology, The Netherlands
PETER MOSSES, Delft University of Technology, The Netherlands

Specifying and mechanically verifying type safe programming languages requires significant effort. This effort can in theory be reduced by defining and reusing pre-verified, modular components. In practice, however, existing approaches to modular mechanical verification require many times as much specification code as plain, monolithic definitions. This makes it hard to develop new reusable components, and makes existing component specifications hard to grasp. We present an alternative approach based on intrinsically-typed interpreters, which reduces the size and complexity of modular specifications as compared to existing approaches. Furthermore, we introduce a new abstraction for safe-by-construction specification and composition of pre-verified type safe language components: *language fragments*. Language fragments are about as concise and easy to develop as plain, monolithic intrinsically-typed interpreters, but require about 10 times less code than previous approaches to modular mechanical verification of type safety.

CCS Concepts: • **Theory of computation → Program verification**; • **Software and its engineering → Formal language definitions**; *Software notations and tools.*

Additional Key Words and Phrases: Type Safety, Modularity, Reuse, Definitional Interpreters, Dependently Typed Programming

ACM Reference Format:
Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-Typed Definitional Interpreters à la Carte. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 192 (October 2022), 30 pages. https://doi.org/10.1145/3563355

1 INTRODUCTION

Type safety is a crucial aspect of designing typed programming languages. According to Pierce [2002], "a type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." The *type safety property* of a language defines precisely what program behaviors its type system is supposed to rule out. But it is challenging to define one's type system and interpreter[1] in a way that it satisfies the intended type safety property—or, put differently, that it rules out the "bad" program behaviors it is supposed to. For this reason, programming language researchers often rely on mathematical

†Eelco worked on this paper until his untimely passing on April 5, 2022.
[1]It is possible to define the dynamic semantics of a language in many different ways. In this paper, we assume that the dynamic semantics is given by an interpreter.

Authors' addresses: Cas van der Rest, Delft University of Technology, Delft, The Netherlands, c.r.vanderrest@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, The Netherlands, a.j.rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, The Netherlands; Peter Mosses, Delft University of Technology, Delft, The Netherlands, p.d.mosses@tudelft.nl.

2475-1421/2022/10-ART192
https://doi.org/10.1145/3563355

proofs to verify that a type system and interpreter satisfies the type safety property. However, constructing a type safety proof can be a labor intensive and complex task.

Our research objective in this paper is to make it as easy as possible for DSL developers to develop and verify the type safety of typed domain-specific languages (DSLs). We propose two sub-objectives that could address this goal together:

(1) Support *reuse* of common programming language components, so that DSL developers can focus on developing type safe DSL components.
(2) Make it easy to develop and debug type safe DSL components by automating the task of verifying that a language definition is type safe.

There is previous research that goes towards addressing these sub-objectives *individually*, but no previous research that we are aware of which provides a viable solution to *both* at the same time.

Previous work by Delaware et al. [2013a,b]; Keuchel and Schrijvers [2013] suggests a promising direction for addressing the first sub-objective, by modularizing *extrinsic* type safety proofs and interpreters. An extrinsic proof is an inductive proof on the structure of the syntax or typing rules of a language. By using modular extrinsic proof techniques [Delaware et al. 2013a,b; Keuchel and Schrijvers 2013], domain-specific language designers can compose their interpreter, type system, and type safety proof from pre-proven cases for off-the-shelf components, allowing them to focus on defining and proving the type safety of domain-specific components. However, for the second sub-objective that we gave above, the extrinsic proof style has a number of shortcomings. Most importantly, it is unclear how to automate the task of constructing modular, extrinsic type safety proof cases. We also argue that the extrinsic specification style is not as easy to work with as the alternative *intrinsically-typed* style which we discuss shortly. In particular, interpreters in an extrinsic specification style contain redundant cases for bad behavior that can never happen in a type safe language because the type system rules it out. Furthermore, understanding *when and why* an extrinsic type safety proof case does not hold, is key to finding and fixing type safety errors. But it requires previous experience with inductive proofs to identify what the error is. Domain-specific language designers, however, usually do not have the necessary experience to verify type safety.

A more concise and declarative style of verifying type safety is to write an *intrinsically-typed interpreter* [Augustsson and Carlsson 1999] in a *dependently-typed host language*. Such interpreters save language designers from having to read and write redundant cases for ill-typed expressions, as the host language type checker can automatically verify that these cases are unreachable in practice. This results in shorter, more declarative specifications that are *safe-by-construction*, in the sense that the type safety of the interpreter follows from the well-typedness of its definition. We do not have to establish type safety in a separate proof: the interpreter *is* the type safety proof. Wadler et al. [2020] observe that, in Agda [Norell 2009], "extrinsically-typed terms require about 1.6 times as much code as intrinsically-typed", leading them to suggest that "intrinsic typing is golden".[2] Another appeal of intrinsically-typed interpreters is that language designers can debug type safety issues by using compiler errors produced by the dependently-typed host language as a guide. While the quality of compile time errors depends on the host language, it does not require previous experience with inductive proof techniques. This makes intrinsically-typed interpreters an attractive approach to developing and debugging type safe languages, because it reduces the amount of work on behalf of DSL developers by taking extrinsic proof obligations for type safety and making them intrinsic to interpreter well-typing. However, intrinsically-typed interpreters fail to address our first sub-objective, since they do not in general support reuse.

[2] A pun referring to the ratio in code size between intrinsically and extrinsically typed, which approximates the golden ratio.

```
data Ty : Set where
  nat  : Ty
  bool : Ty

data Expr : Ty → Set where
  lit  : ℕ → Expr nat
  add  : Expr nat → Expr nat → Expr nat
  tt   : Expr bool
  ff   : Expr bool
  ite  : Expr bool → Expr t →
         Expr t → Expr t
```

```
Val : Ty → Set
Val nat  = ℕ
Val bool = Bool

interp : Expr t → Val t
interp (lit n)       = n
interp (add e₁ e₂)   = interp e₁ + interp e₂
interp tt            = true
interp ff            = false
interp (ite e e₁ e₂) = if interp e then interp e₁
                       else  interp e₂
```

Fig. 1. An intrinsically-typed interpreter for a small expression language

In this paper we adapt and combine techniques for modular meta-theory with the intrinsically-typed approach, and develop a new notion of *intrinsically-typed language fragments* and *language fragment composition* that makes it possible to reuse off-the-shelf pre-verified components. Intrinsically-typed language fragments are as concise and declarative and similarly easy to develop and debug as plain, monolithic intrinsically-typed interpreters. Unlike monolithic interpreters, language fragments can be developed and checked in isolation and combined with other fragments to compose type safe languages from reusable components.

1.1 Background: Intrinsically-Typed Interpreters

Figure 1 shows an intrinsically-typed interpreter for a simple language with arithmetic (lit, add) and Boolean (tt, ff, ite) expressions, implemented in Agda [Norell 2009]. It consists of:

(1) a data type of object language types, Ty : Set;
(2) a function that maps each object language type to a type, Val : Ty → Set;
(3) an indexed data type representing the well-typed object language expressions, Expr : Ty → Set;
(4) an *index-preserving evaluation function* that embeds a type safety theorem by mapping well-typed expression to a value of the same type, interp : Expr t → Val t (i.e., "well-typed expressions cannot go wrong", Milner [1978]).

The key ingredient that allows Agda to verify that the interpreter in Figure 1 is type safe is *dependent pattern matching* [Cockx 2017; Coquand 1992], enabling Agda to infer a precise type for each variable bound by a pattern match clause. For instance, in the clause interp (add e_1 e_2) = …, Agda infers that e_1 and e_2 have type Expr nat in the right hand side of the definition. Indeed, if e_1 or e_2 had any other type, the pattern match would be ill-formed according to the definition of Expr. Since e_1 and e_2 : Expr nat, Agda can deduce that the recursive calls in interp e_1 + interp e_2 must yield natural numbers, as the return type of the calls, Val nat, normalizes to ℕ. Thus no error handling of type mismatching is needed: thanks to Agda's type checker we know that *this will not happen*, and we do not have to spell out any redundant cases for going wrong.

1.2 Challenge: Intrinsically-Typed Programming Language Fragments

The interpreter in Figure 1 mixes arithmetic and Boolean expressions. If we want to extend or reuse (parts of) this language, we have no option but to modify or copy-paste existing code. A better approach is to assemble interpreters from reusable components. Figure 2 provides an informal illustration of how we might define and check such reusable components in isolation, and compose them with other fragments, to incrementally develop a verified and type safe interpreter. We can

```
data Ty : Set where
  {- ...1 -}
  nat : Ty

Val : Ty → Set
{- ...2 -}
Val nat = ℕ

data Expr : Ty → Set where
  {- ...3 -}
  lit  : ℕ → Expr nat
  add : Expr nat → Expr nat → Expr nat

interp : Expr t → Val t
{- ...4 -}
interp (lit n)       = n
interp (add e1 e2) = interp e1 + interp e2
```

```
data Ty : Set where
  {- ...5 -}
  bool : Ty

Val : Ty → Set
{- ...6 -}
Val bool = Bool

data Expr : Ty → Set where
  {- ...7 -}
  tt ff : Expr bool
  ite  : Expr bool → Expr t → Expr t → Expr t

interp : Expr t → Val t
{- ...8 -}
interp tt = true
interp ff = false
interp (ite e e1 e2) = if interp e then interp e1
                                   else interp e2
```

Fig. 2. Two intrinsically-typed interpreters for arithmetic expressions (left) and Boolean expressions (right).

compose such fragments by *concatenating* their constructor declarations and function clauses. The Agda comments {- ...$_i$ -} indicate program points where new constructors or clauses will be inserted during composition. For example, we recover the language from Figure 1 by inserting the constructors of Ty (right) at {- ...$_1$ -}, the clauses of Val (right) at {- ...$_2$ -}, and so forth. Throughout this paper we develop a semantics of intrinsically-typed language fragments that supports this kind of composition, without having to re-type-check existing language fragments.

The challenge with defining fragment composition is that not all extensions are well behaved. In particular, extensions that change the *canonical forms* of a type (i.e., removing or adding a new value constructor for an existing type) are problematic. For example, if we add Val bool = Maybe Bool at the {- ...$_6$ -} position in Figure 2, then the interp (ite e e_1 e_2) case on the right becomes ill-typed. If we were to re-type-check the composed definition, Agda would correctly reject this extended interpreter for being ill-typed. Such re-checking is, however, contrary to the goal of reusing pre-verified components. This raises the question: under which conditions is language fragment composition guaranteed to be well-typed?

This paper answers this question by introducing a subtyping relation for witnessing that canonical forms are preserved when values are extended. That is, the set of constructors for a value of a given type never changes. By using this subtyping relation in our definition of language fragments and language fragment composition, we automatically rule out bad extensions, such as the Val bool = Maybe Bool extension discussed above.

1.3 Contributions

Working in Agda[3], we make the following technical contributions:

- We extend (in Section 3) the techniques (which we recall in Section 2) from *data types à la carte* [Swierstra 2008] to intrinsically-typed interpreters, and introduce a subtyping relation

[3]The code in his paper is available in the accompanying artifact [Van der Rest et al. 2022].

for canonical forms. This relation allows us to define cases of modular intrinsically-typed interpreters in a way that supports type safe composition.
- We introduce (in Section 4) *intrinsically-typed language fragments*, which bundle the syntax and semantics of one or more language constructs. Using a general type union, we define *canonical form unions* to support composition of language fragments with overlapping values.
- We generalize (in Section 5) our framework from Agda's Set to a broader class of semantic domains. With this generalization, we can modularly define languages with effects such as name binding, exceptions, and mutable state, provided that we choose a semantic domain that supports these effects upfront.
- We demonstrate (in Section 5) how language fragments support reuse by developing a small library of pre-verified language components, and reusing these to compose different languages.

These contributions demonstrate that modern dependently typed languages such as Agda or Idris can take us far toward addressing the two sub-objectives from the introductory paragraphs of this paper. However, it would be an overstatement to say that our contributions address the research objective of making it "as easy as possible for DSL developers to develop and verify the type safety of typed domain-specific languages (DSLs)". Our long-term ambition is to take the model that underpins our generic Agda framework, and implement it in a new meta-language that lowers the barrier for entry, and allows language designers to build verified DSLs from reusable components. The language fragment composition operation that we introduce in Section 4 provides a promising model for how language component reuse could work in such a meta-language.

2 DATA TYPES À LA CARTE

We recall how *data types à la carte* [Swierstra 2008] lets us define *open* data types and functions. The remaining sections of this paper extend and build upon this framework. Our exposition closely follows the original exposition by Swierstra, with one difference: we encode types à la carte in Agda using *containers* [Abbott et al. 2005; Altenkirch et al. 2015].[4]

2.1 Composing Data Types

The idea behind data types à la carte is to encode data type definitions as data. By treating data type definitions as data we can explicitly manipulate them, but also recover their meaning by mapping to Agda data types. We explain how to encode data type definitions as *signatures* that can be mapped to plain Agda data types (Section 2.1.1), how to compose them using *signature composition* (Section 2.1.2), and how to define *open data type constructors* using *signature subtyping* (Section 2.1.3).

2.1.1 Signatures. A signature describes a set of data type constructors. The following *record type* in Agda defines a type of signatures. We dub the record type Signature, but it corresponds to what is commonly known as a *finitary container* [Abbott et al. 2005; Altenkirch et al. 2015].[5]

```
record Signature : Set where
  constructor _▷_
  field Symbols : Set
        Arity   : Symbols → ℕ
```

[4]The reason we use containers instead of the signature functors that Swierstra [2008] uses is that the fixpoint of signature functors is not strictly positive and hence rejected by Agda (and other dependently-typed languages). The idea of using containers to implement data types à la carte in a dependently-typed language is due to Keuchel and Schrijvers [2013].
[5]Most sections of this paper could also be defined in terms of plain containers whose Arity is not restricted to be finite. We use finite containers because they make the presentation of data types more uniform (sub-expressions are always given by a vector as we illustrate next). We could have used plain, infinitary containers instead.

Signature records can be constructed using _▷_ [6], as indicated by the keyword "constructor". Such records comprise a set[7] of constructor Symbols, and a function that associates an Arity (given by a natural number) with each constructor symbol.

To illustrate how signatures encode inductive data types, we compare the plain inductive definitions (top) with their encoding as signatures (bottom):[8]

```
data ArithExpr : Set where
  lit : ℕ → ArithExpr
  add : ArithExpr → ArithExpr → ArithExpr

data ArithExprSymbols : Set where
  lit : ℕ → ArithExprSymbols
  add : ArithExprSymbols

ArithExprΣ = ArithExprSymbols ▷ (λ where
  (lit n) → 0; add → 2)
```

```
data BoolExpr : Set where
  tt ff : BoolExpr
  ite : (e e₁ e₂ : BoolExpr) → BoolExpr

data BoolExprSymbols : Set where
  tt ff : BoolExprSymbols
  ite : BoolExprSymbols

BoolExprΣ = BoolExprSymbols ▷ (λ where
  ite → 3; tt → 0; ff → 0)
```

We recover an inductive data type from a signature by taking a fixpoint of the corresponding *signature functor*. The function ⟦_⟧ defines this functor, and the type μ its fixpoint—that is, the type of syntax trees whose constructors are given by σ. [9]

```
⟦_⟧ : Signature → (Set → Set)
⟦ σ ⟧ A = Σ[ s ∈ Symbols σ ] (Vec A (Arity σ s))
```

```
data μ (σ : Signature) : Set where
  ⟨_⟩ : ⟦ σ ⟧ (μ σ) → μ σ
```

Taking the least fixpoint of the ArithExprΣ signature yields a type that is equivalent to ArithExpr:

```
example₀ : ArithExpr
example₀ = lit 42

example₁ : ArithExpr
example₁ = add (lit 11) (lit 31)
```

```
example₀′ : μ ArithExprΣ
example₀′ = ⟨ lit 42 , [] ⟩

example₁′ : μ ArithExprΣ
example₁′ = ⟨ add , ⟨ lit 11 , [] ⟩ :: ⟨ lit 31 , [] ⟩ :: [] ⟩
```

2.1.2 Signature Composition. Signatures can be composed by taking the disjoint union of their symbols and arities. The function :+: defines this disjoint composition using the usual sum type (⊎):

```
_:+:_ : Signature → Signature → Signature
σ₁ :+: σ₂ = (Symbols σ₁ ⊎ Symbols σ₂) ▷ (λ where
  (inj₁ s) → Arity σ₁ s
  (inj₂ s) → Arity σ₂ s)
```

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

Using disjoint signature composition we can define signatures in isolation and compose them without having to re-check them. For example, we can compose the ArithExprΣ signature with BoolExprΣ, accommodating expressions that mix arithmetic and Boolean expressions. For example, the term below encodes a simple if-then-else expression (ite tt 42 0):

```
example₂ : μ (ArithExprΣ :+: BoolExprΣ)
example₂ = ⟨ inj₂ ite , ⟨ inj₂ tt , [] ⟩ :: ⟨ inj₁ (lit 42) , [] ⟩ :: ⟨ inj₁ (lit 0) , [] ⟩ :: [] ⟩
```

[6] The underscores in the name _▷_ indicate the argument positions of the mixfix operator.

[7] Set is the type of types in Agda. To rule out inconsistencies, Agda has an infinite hierarchy of Sets (i.e., Set : Set₁ : ...), and the Signature type that we define really lives in Set₁, since one of its fields is itself a Set. For presentation purposes, we abstract from universe levels and write Set everywhere.

[8] The notation tt ff : BoolExpr is syntactic sugar for two separate constructor declarations tt : BoolExpr and ff : BoolExpr.

[9] The notation Σ[x ∈ A] (B x) denotes a *dependent pair* of a value x : A and a value of type B x for some B : A → Set.

The repeated applications of inj_i make working with composed signatures cumbersome. Following Swierstra [2008], we address this using signature subtyping and smart constructors.

2.1.3 Signature Subtyping and Smart Constructors. Smart constructors construct instances of a data type whose full set of constructors is left open. For example, the following function constructs a literal in the syntax tree of any signature σ that contains the symbol/arity pairs of ArithExprΣ:[10]

```
lit′ : {{ ArithExprΣ ≤ σ }} → ℕ → μ σ
```

The σ in the type of lit′ is thus decided by the context that it is used in, enabling us to flexibly reuse lit′ in different contexts. The source of this flexibility is signature subtyping:[11]

```
record _≤_ (σ₁ σ₂ : Signature) : Set where
  field inj      : ⟦ σ₁ ⟧ A →          ⟦ σ₂ ⟧ A
        proj     : ⟦ σ₂ ⟧ A → Maybe (⟦ σ₁ ⟧ A)
        proj-inj : {x : ⟦ σ₁ ⟧ A}              → proj (inj x) ≡ just x
        inj-proj : {x : ⟦ σ₁ ⟧ A} {y : ⟦ σ₂ ⟧ A} → proj y ≡ just x → inj x ≡ y
```

The type $\sigma_1 \preceq \sigma_2$ witnesses that it is always possible to *inject* elements in the interpretation of σ_1 into the interpretation of σ_2, whereas the converse *projection* is only partial. The proj-inj and inj-proj fields establishes that injection and projection are partial inverses.

It is possible to automatically search for injections into co-products using instance parameters [Devriese and Piessens 2011]. We elide the definition of the necessary instances, but they are entirely analogous to the instances found in the original data types à la carte framework [Swierstra 2008]. The code accompanying this paper also contains the implementation.

Using signature subtyping we can implement the above-mentioned smart constructor for lit′. We use the smart inject function on the left below to implement the smart constructor on the right:

```
inject : {{ σ₁ ≤ σ₂ }} → ⟦ σ₁ ⟧ (μ σ₂) → μ σ₂
inject x = ⟨ inj x ⟩
```

```
lit′ : {{ ArithExprΣ ≤ σ }} → ℕ → μ σ
lit′ n = inject (lit n , [])
```

By defining similar smart constructors for Boolean expressions, $\mathsf{example}_2$ from above can be implemented more concisely as follows:

```
example₂′ : μ (ArithExprΣ :+: BoolExprΣ)
example₂′ = ite′ tt′ (lit′ 42) (lit′ 0)
```

2.2 Composing Functions

We recall how to define a function by cases using data types à la carte.

2.2.1 Algebras. The function fold transforms a tree of type $\mu\ \sigma$ into a value of type A:

```
fold : (⟦ σ ⟧ A → A) → μ σ → A
```

The parameter of type $(\llbracket\ \sigma\ \rrbracket\ A \to A)$ is called an *algebra*, and determines how to turn a constructor whose sub-trees are already folded into a value of type A, into a value of type A. We abbreviate the type of algebras using the following alias (left):

[10] The double braces {{ ... }} declare an instance parameter. Instance parameters are similar to type class constraints in Haskell or implicit parameters in Scala: when invoking a function with an instance parameter, an automatic search is performed during type checking time at the call-site of the function. The search will either automatically find an instance that has the correct type, or cause type checking to fail if Agda cannot find an instance parameter of the expected type.

[11] The curly braces in proj-inj : $\{x : F\ A\} \to \ldots$ are implicit function arguments that Agda will attempt to automatically infer when we construct a record element. The ≡ type is propositional equality.

```
Algebra : Signature → Set → Set
Algebra σ A = ⟦ σ ⟧ A → A
```

```
ArithAlg : Algebra ArithExprΣ ℕ
ArithAlg (lit n ,          []) = n
ArithAlg (add , n₁ :: n₂ :: []) = n₁ + n₂
```

The ArithAlg function defines an algebra that evaluates arithmetic expressions to natural numbers. The patterns n and m bound by matching on the add constructor are not expressions, but rather the numbers that result from evaluating the expressions in those positions. The function fold takes care of evaluating sub-expressions, and is defined as follows:

```
fold : Algebra σ A → μ σ → A
fold f ⟨ s , v ⟩ = f (s , map-fold f v)
```

```
map-fold : Algebra σ A → Vec (μ σ) n → Vec A n
map-fold f []       = []
map-fold f (x :: v) = fold f x :: map-fold f v
```

To pass Agda's termination checker, we must inline the definition of map for lists (map-fold), which applies fold to recursive sub-expressions.

2.2.2 Algebra Composition. We can sum algebras using the ⊕ operator given below.

```
_⊕_ : Algebra σ₁ A → Algebra σ₂ A → Algebra (σ₁ :+: σ₂) A
(f ⊕ g) (inj₁ s , v) = f (s , v)
(f ⊕ g) (inj₂ s , v) = g (s , v)
```

Summing two algebras over two signatures σ_1 and σ_2 thus yields a larger algebra for the signature composition σ_1 :+: σ_2: This algebra sum operator only works for algebras with the *same carrier type* A : Set. This implies that the ArithAlg algebra can *only* be composed with algebras that also use ℕ as their carrier. This excludes, for example, the composition of ArithAlg with an algebra for Boolean expressions with Bool as its carrier type.

We can allow such compositions by defining algebras with an *open* carrier type; i.e., using signature subtyping. The idea is to represent values as signatures, and use signature subtyping to assert what value constructors each algebra *at least* requires; e.g.:

```
ArithAlg₀ : ⦃ NatValΣ ≤ σ ⦄ → ⦃ StuckValΣ ≤ σ ⦄ → Algebra ArithExprΣ (μ σ)
```

The carrier of this algebra is the fixpoint of *some* signature σ, about which we only know that it contains at least the constructors described by NatValΣ and StuckValΣ. As the names suggest, NatValΣ describes natural number values, and StuckValΣ represents a *stuck* value. Stuck values are needed because the recursive positions of ArithExprΣ are not intrinsically guaranteed to return numbers, and if they do not, interpretation gets stuck. By defining a similar algebra for BoolExprΣ, we can assemble an interpreter that we can use to evaluate $\text{example}_2'$:

```
interpArithBool : μ (ArithExprΣ :+: BoolExprΣ)
                → μ (NatValΣ :+: BoolValΣ :+: StuckValΣ)
interpArithBool = fold (ArithAlg₀ ⊕ BoolAlg)
```

```
test : interpArithBool example₂′
     ≡ inject (nat 42 , [])
test = refl
```

2.3 Discussion

The interpreters that we can write using the techniques shown in this section are inherently weakly typed. This weak typing means that we must define values as a data type with separate constructors for each kind of a value. As a result, the algebras we define describe partial functions, returning stuck if a recursively evaluated value is not tagged with the right constructor. By contrast, the intrinsically-typed interpreter shown in the introduction is tagless: Agda uses the type index to figure out what kind of value we are dealing with, allowing us to work with bare values instead. As

a result this interpreter is type-safe by construction. To define such interpreters modularly we lift data types à la carte to indexed types, defining the following types in a composable way:

Ty : Set	Val : Ty → Set	Expr : Ty → Set	interp : Expr t → Val t

The key challenge is that the clauses of interp may use dependent pattern matching on values Val t at an index type t : Ty. If we know exactly what t is, we only have to consider the cases that Val associates with that type. But if Val and Ty are open-ended, how do we know that these will remain to be the only possible values for t?

The answer to this question is to ensure that values have *canonical forms*. Canonical forms lemmas are a widely-used technique for making type safety proofs robust under extension and composition [Delaware et al. 2013a; Pierce 2002; Wright and Felleisen 1994]. In the next section we show that this technique, in conjunction with *indexed* data types à la carte, provides exactly the abstraction we need to encode composable intrinsically-typed interpreters.

3 INDEXED DATA TYPES À LA CARTE FOR COMPOSABLE INTRINSICALLY-TYPED INTERPRETERS

We extend the data types à la carte framework to *indexed* data types, and encode *intrinsically-typed interpreters* in this framework as follows:

- We encode types Ty : Set as a plain *signature* (Section 3.1).
- We encode values Val : Ty → Set as an *algebra* over object language type signatures (Section 3.2).
- We encode expressions Expr : Ty → Set as an *indexed signature* with an *open index type* (Section 3.3) such that we can add new expression- or type constructors, without having to modify or re-check existing expression constructors.
- We encode interpreters interp : Expr t → Val t as an *indexed algebra* over the indexed signatures of object language expressions (Section 3.4). These indexed algebras have an *open carrier type* and an *open index type*, allowing us to can add new expression constructors, interpreter cases, types, and values, without modifying or re-checking existing code. It is crucial that the carrier of these indexed algebras is only open to extensions that *preserve canonical forms*.

3.1 Composing Index Types

Since the index type Ty : Set is a plain data type, we can use plain data types à la carte to encode it as a signature. For example, below on the left is a data type NatTy representing a notion of object language type with a single type constructor, and on the right is its signature encoding:

```
data NatTy : Set where
  nat : NatTy
```

```
data NatTyShape : Set where nat : NatTyShape
NatTyΣ = NatTyShape ▷ (λ _ → 0)
```

By similarly encoding a Boolean type constructor as a signature BoolTyΣ, we can compose a signature that encodes the object language types of the interpreter from Figure 1:

```
ArithBoolTyΣ = NatTyΣ :+: BoolTyΣ
```

3.2 Composing Intrinsically-Typed Values

The intrinsically-typed interpreter in Figure 1 defines Val : Ty → Set as a function. This function maps object language types to their *canonical forms* (i.e., the set of possible value constructors) of that type. (Note that it is also possible to model Val as a data type, but a benefit of modeling Val as a function is that values are *tag-less* [Augustsson and Carlsson 1999], which avoids the need to tag and untag values in the interpreter.) Since Ty : Set is encoded as a signature, we can encode Val as an algebra over that signature. The following algebras define the canonical forms of nat and bool:

```
NatVal : Algebra NatTyΣ Set
NatVal (nat , []) = ℕ
```

```
BoolVal : Algebra BoolTyΣ Set
BoolVal (bool , []) = Bool
```

We can compose these algebras using the algebra sum operation from Section 2.2:

```
ArithBoolVal : Algebra ArithBoolTyΣ Set
ArithBoolVal = NatVal ⊕ BoolVal
```

Folding ArithBoolVal over ArithBoolTyΣ yields a function that is isomorphic to Val from Figure 1.

3.3 Composing Intrinsically-Typed Expressions

The expressions of the intrinsically-typed interpreter in Figure 1 are defined as an *indexed data type* Expr : Ty → Set whose index type is Ty. To define this type in a composable way, we lift the notion of signature from the data types à la carte framework discussed in Section 2 to *indexed signatures*, and obtain a framework for *indexed* data types à la carte.

3.3.1 Indexed Signatures. Below is the type of *indexed signatures* that describe I-indexed data types. We dub this type ISignature I, but this type is also commonly known as a *finitary indexed container* [Altenkirch et al. 2015]:

```
record ISignature (I : Set) : Set₁ where
  constructor _▶_
  field ISymbols : I → Set
        Indices  : {i : Ī} → ISymbols i → List I
```

The ISymbols field relates each index to a set of symbols, and the Indices field associates the symbols at each index with a list whose length represents the arity of each constructor symbol, and whose elements describe what the index (of type $\bar{I}$) of each recursive position is. We can interpret indexed signatures as indexed data types, just as we interpreted signatures as plain data types (Section 2):

```
I⟦_⟧ : ISignature I → (I → Set) → (I → Set)
I⟦ ζ ⟧ P i = Σ[ s : ISymbols ζ i ] (All P (Indices ζ s))
```

```
data Iμ (ζ : ISignature Ī) : I → Set where
  I⟨_⟩ : {i : Ī} → I⟦ ζ ⟧ (Iμ ζ) i → Iμ ζ i
```

The implementation of the I⟦_⟧ function uses the following All relation on lists, which asserts that each element in a given List I satisfies a given proposition $P : I \to \mathsf{Set}$:

```
data All (P : I → Set) : List I → Set where
  []  : All P []
  _::_ : {i : Ī} {xs : List Ī} → P i → All P xs → All P (i :: xs)
```

We can think of I⟦ ζ ⟧ as mapping an indexed signature ζ to its corresponding signature functor on I-indexed types, similarly to how ⟦ σ ⟧ maps a plain signature to its corresponding signature functor on plain types.

3.3.2 Indexed Signature Composition. Two indexed signatures with the same index type I can be summed into a larger signature that comprises the shapes and indices of both:

```
_:++:_ : ISignature I → ISignature I → ISignature I
ISymbols (ζ₁ :++: ζ₂) i       = ISymbols ζ₁ i ⊎ ISymbols ζ₂ i
Indices  (ζ₁ :++: ζ₂) (inj₁ s) = Indices ζ₁ s
Indices  (ζ₁ :++: ζ₂) (inj₂ s) = Indices ζ₂ s
```

Using these ingredients we can now define composable indexed expression types corresponding to the Expr : Ty → Set from Figure 1 where the index type Ty is *fixed*. However, since new language fragments may introduce new object language type constructors, we need to model Ty in a way that allows such extensions.

3.3.3 Indexed Signatures with Open Index Types. We can define signatures whose index type is open by using the subtyping relation _≤_ (Section 2.1.3) to witness a lower bound on the index type, just like we did in Section 2.2.2. The indexed signature below thus describes intrinsically-typed arithmetic expressions whose type constructors are described by any signature σ that contains *at least* the constructors described by NatTyΣ:

```
data IArithExprSymbols ⦃ _ : NatTyΣ ≤ σ ⦄ : μ σ → Set where
  val : ℕ → IArithExprSymbols (inject (nat , []))
  add :     IArithExprSymbols (inject (nat , []))

IArithExprΣ : ⦃ _ : NatTyΣ ≤ σ ⦄ → ISignature (μ σ)
IArithExprΣ = IArithExprSymbols ▶ (λ where
  (val _) → []
  add     → inject (nat , []) :: inject (nat , []) :: [])
```

By defining Boolean expressions in a similar manner, we can compose indexed signatures with open index types:

```
IArithBoolExprΣ : ⦃ _ : NatTyΣ ≤ σ ⦄ → ⦃ _ : BoolTyΣ ≤ σ ⦄ → ISignature (μ σ)
IArithBoolExprΣ = IArithExprΣ :++: IBoolExprΣ
```

By defining a similar subtyping relation for indexed signatures as we did for plain signatures, we can define smart constructors for indexed types, similarly to how we defined smart constructors for plain types in Section 2.1.3. We elide this relation for brevity, and refer the interested reader to the code accompanying the paper where it can be found.

3.4 Composing Index-Preserving Functions

Our goal is to use indexed algebras to encode interpreters of type interp : Expr *t* → Val *t* whose index type Ty : Set and value type Val : Ty → Set are *open*. That is, we should define interp in a way that we can add new constructors to Ty : Set and Val : Ty → Set and ensure that pattern matches inside interp on values remain exhaustive. We realize this goal by defining a subtyping relation in Section 3.4.2 that characterizes such safe extensions. But first, we need indexed algebras.

3.4.1 Indexed Algebras. We can generically fold a tree of type Iμ ζ *i* into a value of type *P i* using the following function, where $P : I \to \mathrm{Set}$:

```
Ifold : ∀[ I⟦ ζ ⟧ P ⇒ P ] → ∀[ Iμ ζ ⇒ P ]
```

Its implementation is analogous to the implementation of fold from Section 2.2.1, and its type uses the following abbreviations for indexed types (both from the Agda Standard Library[12]):

```
∀[_] : (I → Set) → Set
∀[_] {I} P = {i : I} → P i

_⇒_ : (I → Set) → (I → Set) → (I → Set)
(P ⇒ Q) i = P i → Q i
```

We call the function parameter ∀[I⟦ ζ ⟧ *P* ⇒ *P*] an *indexed algebra*, which we will abbreviate using the following alias:

[12] https://github.com/agda/agda-stdlib

```
IAlgebra : (ζ : ISignature I) (P : I → Set) → Set
IAlgebra ζ P = ∀[ I⟦ ζ ⟧ P ⇒ P ]
```

Just like plain algebras, indexed algebras are closed under indexed signature sums:

```
_:⊕:_ : IAlgebra ζ₁ P → IAlgebra ζ₂ P → IAlgebra (ζ₁ :++: ζ₂) P
(f :⊕: g) (inj₁ s , a) = f (s , a)
(f :⊕: g) (inj₂ s , a) = g (s , a)
```

This sum operation assumes that both summands have the *same* carrier, P. To sum indexed algebras corresponding to intrinsically-typed interpreters with *different* notions of types and values, we need a subtyping relation that witnesses what values (or *canonical forms*) a carrier type *at least* has.

3.4.2 Canonical Forms Subtyping. Our goal is to define indexed algebras with open carrier types in the style illustrated by interpArithAlg below:

```
interpArithAlg : {| _ : NatVal ⊆ W |} → IAlgebra IArithExprΣ (fold W)
interpArithAlg (val n ,           []) = ↑ n
interpArithAlg (add   , n₁ :: n₂ :: []) = ↑ (↓ n₁ + ↓ n₂)
```

Here, _⊆_ denotes value subtyping. In order to define composable intrinsically-typed interpreters, this subtyping relation should witness that we can *safely* convert between NatVal and W. In the definition of interpArith, we use ↑ to represent a safe "upcast" from NatVal to W, and ↓ to represent a safe "downcast" from W to NatVal. We define _⊆_ in terms of a *type isomorphism*:

```
record _↔_ (A B : Set) : Set where
  field inj↔      : A → B
        proj↔     : B → A
        proj-inj↔ : {a : A} → proj↔ (inj↔ a) ≡ a
        inj-proj↔ : {b : B} → inj↔ (proj↔ b) ≡ b
```

The inj↔ and proj↔ fields lets us convert any A into a B and vice versa. The proj-inj↔ and inj-proj↔ fields restrict inj↔ and proj↔ to proper inverses.

Intuitively, a witness of the form $W_1 \subseteq W_2$ tells us that W_1 and W_2 define *the same values for the same types*, but W_2 may define values for more types than W_1.

```
record _⊆_ (W₁ : Algebra σ₁ Set) (W₂ : Algebra σ₂ Set) : Set where
  field {| ≤-type |} : σ₁ ≤ σ₂
        canonical   : {V : T → Set} → (t : ⟦ σ₁ ⟧ T) → W₁ (fmap V t) ↔ W₂ (fmap V (inj t))
```

The ≤-type[13] field establishes that W_2 is defined on the same (or more) types as W_1. The canonical field asserts that W_1 and W_2 define *the same value* (up to isomorphism) for every shared type. We express this fact by requiring that W_1 and W_2 are isomorphic for all shared types t, invariant of the type of its sub-trees (T), or the way sub-trees are mapped to values (V). To apply V to the sub-trees in t, we use the function fmap:

```
fmap : (f : A → B) → ⟦ σ ⟧ A → ⟦ σ ⟧ B
fmap f (s , v) = (s , map f v)
```

Finally, we can implement the safe up-casting (↑) and down-casting (↓) operations:

[13] By wrapping the ≤-type field of this record type in instance argument brackets, instance parameter search will be able to automatically resolve this field projection. For example, the inj field projection that occurs in the canonical field of the _⊆_ record type is implicitly projecting from ≤-type.

```
↑ : ⦃ _ : V ⊆ W ⦄ → ∀ {t} → V (fmap (fold W) t) → fold W (inject t)
↓ : ⦃ _ : V ⊆ W ⦄ → ∀ {t} → fold W (inject t) → V (fmap (fold W) t)
```

The implementation of these function uses the type isomorphism in the canonical field of the ⦃ _ : V ⊆ W ⦄ instance argument to safely convert between the values computed by the sub- and super algebras. Comparing with Section 2, we can think of ↑ as the intrinsically-typed counterpart to inject, and of ↓ as the intrinsically-typed counterpart to a weakly-typed projection function. The crucial difference is that ↓ is a total function, whereas project may return nothing.

3.4.3 Indexed Algebra Composition. Indexed algebras can be composed in the same way as plain algebras. For example, we can compose interpArithAlg with the following indexed algebra for evaluating intrinsically-typed Boolean expressions:

```
interpBoolAlg : ⦃ _ : BoolVal ⊆ W ⦄ → IAlgebra IBoolExprΣ (fold W)
interpBoolAlg (tt   ,              []) = ↑ true
interpBoolAlg (ff   ,              []) = ↑ true
interpBoolAlg (ite  , v :: v₁ :: v₂ :: []) = if ↓ v then v₁ else v₂

interpArithBoolAlg :  ⦃ _ : NatVal ⊆ W ⦄ → ⦃ _ : BoolVal ⊆ W ⦄
                   → IAlgebra IArithBoolExprΣ (fold W)
interpArithBoolAlg = interpArithAlg :⊕: interpBoolAlg
```

Using these indexed algebras, we can now evaluate expressions given by the fixpoint of an indexed signature, using Ifold:

```
Ifold : IAlgebra ζ P → ∀[ Iμ ζ ⇒ P ]
```

This yields an interpreter that is analogous to the interpreter from Figure 1 in the introduction:

```
interpArithBool : Iμ IArithBoolExprΣ t → fold (NatVal ⊕ BoolVal) t
interpArithBool = Ifold interpArithBoolAlg
```

Unlike the interpreter from the introduction, this interpreter is assembled from separately-defined, reusable components—i.e., interpArithAlg and interpBoolAlg.

3.5 Discussion

This section showed how to assemble intrinsically-typed interpreters using *indexed* data types à la carte and canonical forms subtyping. While this approach allows us to assemble languages, the notion of *language fragment* that we illustrated in Figure 2 in the introduction remains informal. There are three reasons why it is useful to define language fragments as a first-class abstraction instead. The first reason is that it would allow language designers to compose languages using a single, uniform notion of *language fragment composition*, instead of the four we used in this section to compose respectively types, values, expressions, and interpreters. The second reason is that intrinsically-typed interpreters as defined in this section are not closed under composition, since indexed algebra composition "grows" the number of canonical forms subtype constraints, as interpArithBoolAlg illustrates. The last reason is that it is possible for interpreters to make *conflicting assumptions* about values, as we illustrate below.

Consider the encoding below of an intrinsically-typed interpreter which uses *ternary* Booleans, akin to the Val bool = Maybe Bool clause discussed in Section 1.2. The null expression below constructs the third kind of Boolean value:

```
TernaryBoolVal : Algebra BoolTyΣ Set
TernaryBoolVal (bool , []) = Maybe Bool
```

```
data INullExprShape ⦃ _ : BoolTyΣ ≤ σ ⦄ : μ σ → Set where
  null : INullExprShape (inject (bool , []))

INullExprΣ    : ⦃ BoolTyΣ          ≤ σ ⦄  → ISignature (μ σ)
INullExprΣ = INullExprShape ▶ λ _ → []

interpNullAlg : ⦃ _ : TernaryBoolVal ⊆ W ⦄ → IAlgebra INullExprΣ (fold W)
interpNullAlg (null , []) = ↑ nothing
```

Using algebra composition, we can compose interpBoolAlg with interpNullAlg, growing the number of subtype constraints:

```
badAlgebra : ⦃ _ : BoolVal ⊆ V ⦄ → ⦃ _ : TernaryBoolVal ⊆ V ⦄
           → IAlgebra (IBoolExprΣ :++: INullExprΣ) (fold V)
badAlgebra = interpBoolAlg :⊕: interpNullAlg
```

This is unsatisfactory: the subtype constraints of badAlgebra represent an *unsound* identification of types, because BoolVal and TernaryBoolVal define different canonical forms for the bool type. In other words, the subtype constraints *conflict*—i.e., cannot be proven—for any V : Algebra BoolTyΣ Set. The next section introduces *intrinsically-typed language fragments*, with a composition operator that avoids the issues discussed above, and enables sound intrinsically typed fragment composition.

4 INTRINSICALLY-TYPED LANGUAGE FRAGMENTS

In this section we introduce *language fragments*: an abstraction that bundles a set of intrinsically-typed syntax constructors with the associated cases of an interpreter. This abstraction comes with a single composition operation, *language fragment composition*, that has nice closure properties, and that subsumes the four different notions of composition that we introduced in the previous section (for object language type signatures, value typing algebras, expression data type signatures, and interpreter algebras). It thus goes towards addressing both the first and the second sub-objective stated in the introduction of this paper, by making it easier to develop type safe DSL components in a way that supports reuse.

We first discuss (Section 4.1) how to bundle these four components in a way that each component is defined as being *open*. We then discuss (Section 4.2) how to compose language fragments, and why it is necessary to allow compositions with partially-overlapping canonical forms. Finally, we present (Section 4.3) language fragment composition, which makes language fragments closed under composition. The idea of language fragments, however, transcends their formulation in this section, which concerns the definition and composition fragments of simply-typed expression languages. In Section 5, we will consider how to apply the same techniques to a more expressive semantic domain.

4.1 Canons and Language Fragments

Language fragments (Fragment) bundle a piece of intrinsically-typed syntax with its interpretation, and are parameterized over the available *canonical forms* (Canon)—i.e., a signature of object language types together with an algebra over this signature.

```
record Canon : Set where
  constructor canon
  field ty  : Signature
        val : Algebra ty Set
```

```
record Fragment (c : Canon) : Set where
  field expr   : ISignature (μ (ty c))
        interp : IAlgebra expr (fold (val c))
```

A value of type Fragment c is a *self-contained description* of an intrinsically-typed interpreter. It is, however, defined in terms of the *fixed* set of canonical forms given by c, meaning that the composition of such fragments is limited to fragments that depend on the *same* canonical forms. To compose language fragments with *different* canonical forms, we must define them with extension of the canon in mind, similar to how we defined indexed algebras with open carriers in Section 3.4.2.

Because language fragments are a self-contained description of intrinsically-typed interpreters, we can view extensibility of canonical forms independent from the definition of fragments. To do this, we define the □ modifier, which *closes* a canon-indexed type over all possible future extensions:[14]

```
□ : (Canon → Set) → Canon → Set
□ P c = ∀ {c′} → ⦃ val c ⊆ val c′ ⦄ → P c′
```

We use the □ modifier to define *open* language fragments for the intrinsically-typed interpreter components from Section 3:

```
ArithFrag : □ Fragment (canon NatTyΣ
                              NatVal)
expr   ArithFrag = IArithExprΣ
interp ArithFrag = interpArithAlg
```

```
BoolFrag : □ Fragment (canon BoolTyΣ
                              BoolVal)
expr   BoolFrag = IBoolExprΣ
interp BoolFrag = interpBoolAlg
```

The □ modifier provides exactly the subtyping proof that interpBoolAlg and interpArithAlg need. Furthermore, it is always possible to transform a open fragment to a closed fragment, and extract an interpreter from a closed fragment:

```
extract : ∀[ □ P ⇒ P ]
extract ϕ = ϕ ⦃ ⊆-refl ⦄
```

```
toInterp : (ϕ : Fragment c) → ∀[ Iμ (expr ϕ) ⇒ fold (val c) ]
toInterp ϕ = Ifold (interp ϕ)
```

4.2 Fragment Composition and the Need for Partially-Overlapping Canons

We define a composition operation for open fragments with the same canonical forms as follows:

```
fcompose-eq : ∀[ □ Fragment ⇒ □ Fragment ⇒ □ Fragment ]
expr   (fcompose-eq ϕ₁ ϕ₂) = expr   ϕ₁ :++: expr   ϕ₂
interp (fcompose-eq ϕ₁ ϕ₂) = interp ϕ₁ :⊕:  interp ϕ₂
```

In many scenarios, however, fcompose-eq alone is insufficient: it is often necessary to compose fragments with *different* canonical forms. Indeed, ArithFrag and BoolFrag have different canonical forms, so we cannot compose them into a new open fragment with fcompose-eq.

It may seem tempting to use a composition operation that *sums* the canonical forms of fragments, but this is also problematic. Say we have a fragment that defines an interpreter for a binary less-than-or-equals expression that compares the results of evaluating its arguments:

```
LeqFrag : □ Fragment (canon (NatTyΣ :+: BoolTyΣ) (NatVal ⊕ BoolVal))
```

A fragment that combines LeqFrag and BoolFrag by summing their canons has the following type:

```
Fragment (canon (NatTyΣ :+: BoolTyΣ :+: BoolTyΣ) (NatVal ⊕ (BoolVal ⊕ BoolVal)))
```

The resulting fragment has two *distinct* notions of Boolean types and values! As a result, an expression such as ite (leq 0 1) 42 0 would be ill-typed; the bool type of leq 0 1 is not the same bool

[14] The □ notation is borrowed from the necessity modality of modal logic. Its implementation is essentially a shallow embedding of *Kripke Semantics* [Kripke 1963] and is inspired by Allais et al. [2018].

```
record Union (A B C : Set) : Set where
  field
    inja  : A → C
    injb  : B → C
    from : C → These A B

    inja-inv  : ∀ {a} →
      ⟪ _≡ a , ∅ , (_≡ a) ∘ proj₁ ⟫ (from (inja a))
    injb-inv  : ∀ {b} →
      ⟪ ∅ , _≡ b , (_≡ b) ∘ proj₂ ⟫ (from (injb b))
    from-inv : ∀ {c} →
      ⟪ (_≡ c) ∘ inja , (_≡ c) ∘ injb
      , (λ (a , b) → inja a ≡ c × injb b ≡ c) ⟫ (from c)
```

```
data These (A B : Set): Set where
  this  : A →       These A B
  that  :      B → These A B
  these : A → B → These A B

⟪_,_,_⟫ :  (A → X)
        → (B → X)
        → (A × B → X)
        → These A B → X
⟪ f , g , h ⟫ (this  a  ) = f a
⟪ f , g , h ⟫ (that     b ) = g b
⟪ f , g , h ⟫ (these a b ) = h (a , b)
```

Fig. 3. The proof-relevant relation Union $A\ B\ C$ specifies that C is the union of A and B. The types A and B can overlap in C, in which case inja $a \equiv$ injb b for some elements $a : A$ and $b : B$. The ternary relation uses the type These and its $\langle\!\langle \_,\_,\_ \rangle\!\rangle$ eliminator shown on the right.

type that ite expects. Rather, we should identify the Boolean types of the two fragments. In other words: their canonical forms are *partially overlapping*.

4.3 Fragment Composition with Partially-Overlapping Canons

We introduce a fragment composition operation for language fragments with partially-overlapping canons in three stages. First, we introduce *type unions* (Section 4.3.1), which precisely characterize how two types overlap. Then, we define a similar *union for canonical forms* (Section 4.3.2) in terms of this type union, which describes how two canons overlap. Finally, we define *fragment composition* (Section 4.3.3), which combines two open fragments, given a canon union that witnesses how their canonical forms are combined.

4.3.1 Overlapping Unions for Types. Figure 3 defines the Union relation, which describes how the elements of two types A : Set and B : Set map to elements in a third type C : Set, such that each element in C corresponds to an element of either A, B, or both. The correspondence is witnessed by the functions inja, injb, and from, which should be injective. [15]

There are two trivial unions which can always be constructed:

```
union-copy : Union A A A
```

```
union-disjoint : Union A B (A ⊎ B)
```

The function union-copy constructs a union of A with itself such that all its elements overlap with themselves. Conversely, union-disjoint constructs a union of two sets of types A and B such that none of their elements are identified in $A \uplus B$. We elide the proofs of the inverse laws (inja-inv, injb-inv, and from-inv) here; the code accompanying this paper contains the details.

4.3.2 Overlapping Canons. Using the Union relation, we define a similar ternary union relation for canons:

```
record _•_≣_ (c₁ c₂ c : Canon) : Set where
  field ⦃ ty-union ⦄ :  Union ⟦ (ty c₁) ⟧ T ⟦ (ty c₂) ⟧ T ⟦ (ty c) ⟧ T
        canonical_ι  :  {V : T → Set} {t : ⟦ (ty c₁) ⟧ T}
```

[15] For readers wondering how general this type union is: the Union type is a pushout in the category of Agda Sets, meaning it is union-like indeed.

```
                → (val c₁) (fmap V t) ↔ (val c) (fmap V (inja t))
canonical_r  :  {V : T → Set} {t : ⟦ (ty c₂) ⟧ T}
                → (val c₂) (fmap V t) ↔ (val c) (fmap V (injb t))
```

The canon union is directed by a type union: ty-union witnesses that the type constructors of c are a union of the type constructors in c_1 and c_2. The key, however, are the additional proofs canonical_l and canonical_r that witness that the values of c_1 and c_2 agree in the overlap (as described by the type union) of their types. This amounts to a modularization of canonicity lemmas.

Since canon union is based on type union, we can construct similar trivial unions, where either all type constructors are overlapping (•-copy), or no type constructors are overlapping (•-disjoint, which uses the auxiliary disjoint canon union on the left):

```
_⊎ᶜ_ : (c₁ c₂ : Canon) → Canon
ty  (c₁ ⊎ᶜ c₂) = (ty  c₁) :+: (ty  c₂)
val (c₁ ⊎ᶜ c₂) = (val c₁) ⊕  (val c₂)
```

```
•-copy     : c • c ≣ c
•-disjoint : c₁ • c₂ ≣ (c₁ ⊎ᶜ c₂)
```

We show how to use •-disjoint and •-copy to compose ArithFrag, BoolFrag, and LeqFrag in Section 4.3.4.

4.3.3 Fragment Composition Operation. Using canon union, we can now define a general fcompose operation which lets us compose fragments with partially-overlapping canons:

```
fcompose : □ Fragment c₁ → □ Fragment c₂ → c₁ • c₂ ≣ c → □ Fragment c
```

Perhaps surprisingly, we can implement this operation in terms of the fcompose-eq operation, by (1) exploiting the *comonadic structure* of □, and (2) observing that we can recover subtyping (_⊆_) witnesses from union (_•_≣_) witnesses. For □, we already showed that it has an extract function Section 4.1. Additionally, we can define a duplicate function.[16], following from transitivity of _⊆_:

```
duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate px ⦃ w₁ ⦄ ⦃ w₂ ⦄ = px ⦃ ⊆-trans w₁ w₂ ⦄
```

The type of duplicate says that we can "weaken" the canon that is implicitly quantified by ∀[_],

Next, we observe that canon union implies value subtyping:

```
•-to-⊆_l : c₁ • c₂ ≣ c → (val c₁) ⊆ (val c)
```

```
•-to-⊆_r : c₁ • c₂ ≣ c → (val c₂) ⊆ (val c)
```

Using these ingredients, we can define fcompose in terms of fcompose-eq as follows:

```
fcompose : □ Fragment c₁ → □ Fragment c₂ → c₁ • c₂ ≣ c → □ Fragment c
fcompose ϕ₁ ϕ₂ u = fcompose-eq (duplicate ϕ₁ ⦃ •-to-⊆_l u ⦄) (duplicate ϕ₂ ⦃ •-to-⊆_r u ⦄)
```

The duplicate function is used to "weaken" the c_1 and c_2 canons of the left and right fragment into the canon union c. The lemmas •-to-⊆_l and •-to-⊆_r witness that this weakening is safe.

4.3.4 Fragment Composition Examples. With fcompose we compose languages from fragments:

```
ArithBoolFrag = fcompose ArithFrag BoolFrag •-disjoint
```

From this fragment, we can derive the intrinsically typed interpreter discussed in Section 1:

```
InterpArithBool = toInterp (extract ArithBoolFrag)
```

We can also compose ArithBoolFrag with LeqFrag, to obtain a larger language:

```
LeqArithBoolFrag = fcompose LeqFrag ArithBoolFrag •-copy
```

[16]The extract and duplicate functions correspond to the operations of a *comonad*.

In summary, fcompose addresses the three concerns we discussed in Section 3.5. It provides a single, uniform composition operation for language fragments with compatible but possibly different canonicity assumptions, such that language designers do not have to manually assemble types, values, expressions, and interpreters separately, using four different composition operators. It ensures that language fragments are closed under composition using fcompose. And the interpreters we extract from language fragments cannot have conflicting canonicity assumptions.

5 LANGUAGE FRAGMENTS WITH LEXICAL VARIABLES AND EFFECTS

Language fragments presuppose both the notion of typing and semantics. The definition of a fragment in Section 4 takes well-typed terms to be indexed families (Expr : Type → Set), and their semantics to be an interpreter (interp : Expr t → Val t). These presupposed notions limit the expressive power of fragments. For example, well-typed expressions with lexical binding are usually presented as indexed families that are additionally parameterized by a typing context. Their interpreter correspondingly requires an environment of values for variables. Well-typed expressions for ML-style references, on the other hand, have an intrinsically-typed interpretation in the category of monotone predicates [Bach Poulsen et al. 2018]. This motivates this section, in which we show that the notion of intrinsically-typed language fragments introduced in the previous section can be transported to more expressive semantic domains.

In this section we generalize language fragments to a class of semantic domains (Section 5.1) that we show can be used to define intrinsically-typed interpreters for the simply-typed λ-calculus (Section 5.2), exceptions (Section 5.3), and ML-style references (Section 5.4). Although intrinsically-typed language fragments can be transported to this more general setting, fragment composition only combines fragments that are interpreted into the same semantic domain. Since the examples in Sections 5.2 to 5.4 are interpreted in different domains, they cannot be combined into the same language using fragment composition. Instead, we can manually lift these fragments into a common "super domain" where they can be composed, as we demonstrate with our case study in Section 5.5. Furthermore, additional innovation may be required to modularize richer semantic domains that do not fit into the description we introduce in Section 5.1.

5.1 Fragments for a Class of Semantic Domains

We generalize the definition of language fragments to permit: (1) expressions that are typed relative to a context for de-Bruijn encoded lexical variables, (2) interpretation functions that accept a lexical environment as parameter, and (3) side-effects—in particular ML-style references. The generalization is based on a generalization of the codomain of interpretation functions to some *cartesian category* $\mathscr{C}$—that is, a category with all products and a terminal object. Informally, the semantic domains presupposed by the definition of fragments from Section 4 vs. the definition we consider in this section differ as follows:

	Typing	Interpretation of types	Interpretation of typed terms
Section 4	$e : t$	Val t : Set	interp ($e : t$) : Val t
Section 5	$\Gamma \vdash e : t$	Val t : obj($\mathscr{C}$)	interp ($\Gamma \vdash e : t$) : $\mathscr{C}$(Env Γ, Val t)

Here, obj($\mathscr{C}$) represents the objects of $\mathscr{C}$, and $\mathscr{C}$(Env Γ, Val t) represents the morphisms from Env Γ to Val t in $\mathscr{C}$.

In Agda, we can define a type of categories $\mathsf{Category}_0$ and a generalized canon Canon_0 that maps types to objects of some category $\mathscr{C}_0$ as follows:

```
record DomainDesc : Set where
  field obj   : (T : Set) → Set
        morph : (V : T → obj T) → (X Y : obj T) → Set
```

```
record Canon : Set where
  constructor canon
  field
    ty  : Signature
    val : ∀ {T} → ⟦ ty ⟧ (T × obj 𝒟 T)
                → obj 𝒟 T
```

```
record Fragment (c : Canon) : Set where
  field
    expr   : ISignature (List (μ (ty c))) × μ (ty c))
    interp : IAlgebra expr
                 (λ (Γ , t) → morph 𝒟 (fold (val c))
                                (Env c Γ)
                                (fold (val c) t))
```

Fig. 4. Definition of categories, canons, and language fragments. The domain description $\mathscr{D}$ is a module parameter, and Env c Γ is a de Bruijn indexed environment comprising value objects whose canonical forms are given by the canon c.

```
record Category₀ : Set where
  field obj   : Set
        morph : obj → obj → Set
```

```
record Canon₀ : Set where
  field ty  : Signature
        val : Algebra ty (obj 𝒞₀)
```

However, it is not possible to define a canon for ML-style reference values in this style! The problem is that the Canon_0 type assumes that we can compositionally map types onto objects of $\mathscr{C}_0$. This is not true for ML-style references. To see the issue, consider the following mock case of a refCanon : Canon_0 that we wish to define, assuming that $\mathscr{C}_0$ is a category of monotone predicates; i.e., obj $\mathscr{C}_0$ = List T → Set.

```
val refCanon (ref , V :: []) = λ Σ → ???
```

Following how references are traditionally typed [Harper 1994; Pierce 2002], the right hand side is supposed to witness that there exists an location of type t in the store type Σ, where t is a subterm of ref t. However, since Canon_0 defines values as plain algebras, the mock case above does not have access to the subterm t; only to the object V : obj $\mathscr{C}_0$ resulting from folding over the sub-term t.

It is possible to construct a compositional interpretation of reference types—that is, in terms of the object V—in the context of *semantic typing*, where types are viewed as a *set of values*. In our setting this would correspond to defining store types as a list of Sets, and mapping reference types to a proof that V is a member of this list. If we naively attempt to define values this way, however, their definition becomes inconsistent, since the type of these membership proofs is simultaneously bigger than and included in the set of values! Although we can escape this paradox by stratifying the interpretation of types [Ahmed 2004; Ahmed et al. 2002], we would need to find a way to adapt the intrinsically-typed semantics for ML-style references by Bach Poulsen et al. [2018] to interpret into such a layered domain.

Instead, we opt to generalize canons and categories to reflect that the set of objects and morphisms comprise components assembled from indexed signatures and algebras. This generalized definition is shown and Figure 4. Although similar, the explicit injection of types and values means that the resulting structure (defined in the DomainDesc record) is not quite a category. We will refer to this

structure as a *domain description*, using $\mathscr{D}$ to range over it. The move from categories to domain descriptions requires three generalizations:

(1) Canons are generalized to use a flavor of algebra that supports *paramorphisms* [Meertens 1992]—i.e., a recursion scheme that provides access to each sub-term both before and after we have recursively folded over it. The val field of Canon in Figure 4 shows the generalization, which maps a pair of the original sub-term of type T and its folded counterpart of type obj $\mathscr{D}$ T to an object of type obj $\mathscr{D}$ T, where T is the final set of object language types.
(2) Objects of a domain description may depend on a final type of object language types, as the T : Set parameter of the obj field of DomainDesc in Figure 4 indicates. This generalization is used to define modular paramorphic algebras for which we only learn the final type of object language types after we are done composing all canons and fragments.
(3) Morphisms of domain descriptions may depend on the final value typing $T \to$ obj $\mathscr{D}$ T, as the first parameter of the morph field of Fragment in Figure 4 indicates. This generalization is used to define domain descriptions whose morphisms depend on value typings. For example, to define ML-style references Section 5.4 we use morphisms that implicitly thread stores with values that depend on value typings.

Using these generalizations, the Fragment type in Figure 4 ensures that a fragment closure □ Fragment will use the final canon of the language as the definition of the final set of types and the final set of value typings which objects and morphisms depend on.

By deriving $\mathscr{D}$ from the category of Agda Sets (i.e, objects are types in Set and morphisms are Agda functions) we regain the framework as defined in Section 4, but now extended with the necessary infrastructure for variables and stores. It is straightforward to transport the fragments we developed in Section 4 to this more expressive setup.

Before we can define intrinsically-typed fragment instances in this more expressive setting, we also need to transport the definitions of isomorphism, canon union, and canon subtyping that we introduced in Section 3.1 and Section 4. The essential ingredient of these definitions is the notion of isomorphism. Lifting this notion isomorphism to relate objects of domain descriptions via morphisms of domain descriptions, all of the definitions from before translate straightforwardly. The safe upcasting and downcasting operations for translating between between the value typings of the fragment canon and the final fragment look more involved because of the switch to paramorphisms:

```
↑ : ∀ ⦃ _ : c₁ ⊆ c ⦄ {t} → morph 𝒟 _ (val c₁ (fmap (λ t → t , fold (val c) t) t))
                                      (fold (val c) (inject t))
↓ : ∀ ⦃ _ : c₁ ⊆ c ⦄ {t} → morph 𝒟 _ (fold (val c) (inject t))
                                      (val c₁ (fmap (λ t → t , fold (val c) t) t))
```

In the remainder of this section we illustrate how this more expressive variant of language fragments allows us to define language fragments for different language features and effects.

5.2 Simply-Typed Lambda Calculus

As our first example, we consider how to define a fragment for the simply-typed λ-calculus. We instantiate the definitions from Section 5.1 with a domain description based on the category of Agda Sets:

```
Sets : DomainDesc
obj    Sets T     = Set
morph Sets V A B = A → B
```

We use the following canon, interpreting types in Set:

```
data TFunShape : Set where
  fun : TFunShape
TFunΣ = TFunShape ▷ λ where
  fun ⟶ 2
```

```
funCanon : Canon
ty  funCanon = TFunΣ
val funCanon (fun , (s , V) :: (t , W) :: []) = V ⟶ W

fun′ : ⦃ TFunΣ ≤ σ ⦄ ⟶ (s t : μ σ) ⟶ μ σ
fun′ s t = inject (fun , (s :: t :: []))
```

Here, we see the impact of using a paramorphism: the arguments to fun are not just replaced by their value (V/W), but paired with the original type (s/t) as well. The interpretation of function types is defined solely in terms of V and W, but we will need access to the uninterpreted recursive argument to define a language fragment for ML-style references (Section 5.4).

The signature LamExprΣ defines the three standard constructs of the λ calculus:

```
data LamExprShape ⦃ _ : TFunΣ ≤ σ ⦄ : List (μ σ) × μ σ ⟶ Set where
  var : t ∈ Γ ⟶      LamExprShape (Γ , t)
  abs :              LamExprShape (Γ , fun′ s t)
  app : {s : μ σ} ⟶ LamExprShape (Γ , t)

LamExprΣ : ⦃ TFunΣ ≤ σ ⦄ ⟶ ISignature (List (μ σ) × μ σ)
LamExprΣ = LamExprShape ▶ λ where
  (var  x)        ⟶                          []
  (abs {Γ}{s}{t}) ⟶ (s :: Γ , t         )    :: []
  (app {Γ}{t}{s}) ⟶ (Γ      , fun′ s t ) :: (Γ , s) :: []
```

The var and abs constructors demonstrate the need for tracking a type context Γ. To reference a variable we must supply a witness $t \in \Gamma$ proving that it is in scope, and the type context of the function body is extended with the argument type s when constructing a λ-abstraction.

We then define a language fragment for the simply-typed λ-calculus as follows:

```
stlc : □ Fragment funCanon
expr   stlc = LamExprΣ
interp stlc (var x ,          []) nv = fetch x nv
interp stlc (abs   , e      :: []) nv = ↑ (λ v ⟶ e (v , nv))
interp stlc (app   , e₁ :: e₂ :: []) nv = ↓ (e₁ nv) (e₂ nv)
```

Variables are interpreted by invoking the function fetch : $t \in \Gamma \longrightarrow$ Env $\Gamma \longrightarrow$ fold (val c) t, which performs a safe lookup in the environment. Since function types are mapped to Agda functions, we can reuse Agda's function abstraction and application to interpret the abs and app constructors.

5.3 Exceptions

Next, we consider a language fragment for *safe division*, which raises an exception when the divisor is zero. In general, we can define effectful fragments by choosing a suitable *monad* [Moggi 1989] that encapsulates the effects, and instantiating with a domain description based on the corresponding *Kleisli category* (which applies the monad to the target object of morphisms). For exceptions we use Maybe, which is a monad on the category Sets. The corresponding domain description is defined as follows:

```
MSets : DomainDesc
obj    MSets _     = Set
morph MSets _ A B = A ⟶ Maybe B
```

With this description, terms are interpreted as a function with type Env Γ ⟶ Maybe (fold (val *c*) *t*). This allows us to implement an interpreter for div, which returns nothing if the divisor is zero, where the function _/_ takes an (automatically inferred) proof that the divisor is greater than zero.

```
divide : □ Fragment natCanon
expr   divide = DivExprΣ
interp divide (div , m₁ :: m₂ :: []) nv = do
  v₁ ← m₁ nv >>= ↓
  v₂ ← m₂ nv >>= ↓
  case v₂ of λ where
    zero    → nothing
    (suc n) → ↑ (v₁ / suc n)
```

Rather than working with values of the Maybe type directly, we use Agda's **do**-notation[17] as syntactic sugar for monadic computation, as well as the >>= operator which denotes monadic bind [Moggi 1991]. Note that since the domain description is based on a Kleisli category, the result of up- and down casting (which are defined as morphisms) is now also wrapped in a Maybe.

5.4 ML-Style References

Finally, we consider how to define a language fragment for ML-style references, based on the intrinsically-typed semantics by Bach Poulsen et al. [2018]. We interpret into a domain based on a Kleisli category generated from the description ST, which has store-type-indexed Sets as objects and index-preserving functions as morphisms. We will discuss the relevant monad shortly.

```
ST : DomainDesc
obj    ST T     = List T → Set
morph ST V P Q = ⦃ Weakenable V ⦄ → ∀[ P ⇒ Q ]
```

In the definition of ST, we require access to the syntax of types (*T*) and their interpretation (*V*) to define the sets of objects and morphisms. Store types are defined in terms of *T*, and to interpret ML-style references, we need to express the assumption that *V* is *weakenable*: every value that is well-typed relative to a given store, can also be typed with a *bigger* store.

Note that the ST description defined above admits non-monotone predicates as objects; that is, objects are not guaranteed to be Weakenable. We could rectify this by requiring objects to be weakenable; i.e.:

```
ST′ : DomainDesc
obj    ST′ T     = ∃ λ (P : List T → Set) → Weakenable (const P)
morph ST′ V P Q = ∀[ proj₁ P ⇒ proj₁ Q ]
```

However, that would clutter the resulting interpreter, which relies on dependent pattern matching on objects. For that reason, we use ST which has less structure, but which allows us to explicitly assume that predicates are weakenable when we need it.

The canon for references shows why val needs to be a paramorphism: the interpretation of the type ref *t* is a proof of the form $t \in \Sigma$, which makes val non-compositional.

[17] https://agda.readthedocs.io/en/v2.6.2.2/language/syntactic-sugar.html

```
Ref : T → List T → Set
Ref t Σ = t ∈ Σ

record Mem (M : Monad ST) : Set where
  field
    alloc    : ∀[ V t ⇒ M V (Ref t) ]
    retrieve : ∀[ Ref t ⇒ M V (V t) ]
    write    : ∀[ Ref t ⇒ V t ⇒ M V U ]
```

```
refs : ⦃ Mem M ⦄ → □ Fragment refCanon
expr refs = RefExprΣ
interp refs (init , m :: []) nv = do
  v ← m nv
  alloc v >>= ↑
interp refs (deref , m :: []) nv = do
  l ← m nv >>= ↓
  retrieve l
interp refs (update , m₁ :: m₂ :: []) nv = do
  (l , nv) ← ( m₁ nv >>= ↓ ) ^ nv ⟨ wk-env ⟩
  (v , l  ) ←   m₂ nv        ^ l  ⟨ wk-ref ⟩
  write l v >>= ↑
```

Fig. 5. Definition of the Mem interface, and a fragment for ML-style references

```
data TRefShape : Set where
  ref unit : TRefShape
TRefΣ = TRefShape ▷ λ where
  ref  → 1
  unit → 0
```

```
refCanon : Canon
ty  refCanon = TRefΣ
val refCanon (ref  , (t , V) :: []) Σ = t ∈ Σ
val refCanon (unit ,         []) Σ = ⊤
```

Interpreting ML-style references has side effects, in the sense that the interpreter can modify a global store. To define the interpreter for ML-style references, we require a monad over store predicates that encapsulates this interaction with the store. Rather than settling on a particular monad, we keep it abstract, and assume that it satisfies the Mem interface (Figure 5, left), which provides the operations alloc, retrieve, and write. Following Bach Poulsen et al. [2018], we also require that the chosen monad has *tensorial strength* [Moggi 1991], meaning it supports the following operation, where ∩ is the usual product type lifted to predicates:

```
_^_ : ∀[ M P ⇒ Q ⇒ M (P ∩ Q) ]
```

We use this operation whenever we compute a value, but need to perform more computations before we can return that value. This situation occurs, for example, in the update case of the interpreter in Figure 5. Since computations may change the store, it is not immediately clear that previously computed values are still typeable relative to the updated store after running these computations. The strength operation allows us to pass these values back into the monad by pairing them with a computation, updating their store typing along the way. The key to implementing strength is to assume that the store only ever increases monotonically during execution (i.e., values can be added or changed, but never deleted), and to require that strength can only be applied to values that are weakenable with respect to this ordering. The interpreter itself (Figure 5, right) is then defined in terms of the operations provided by these interfaces, and defines interpretation for init, deref, and update expressions, which respectively create, read from, and update a reference. Hence, we can use this fragment with any monad that satisfies the Mem interface and has tensorial strength.

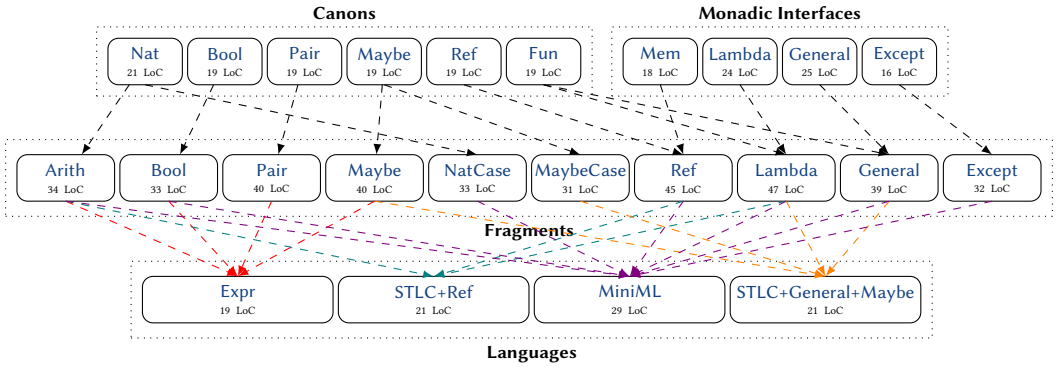

Fig. 6. Overview of canons, interfaces, fragments, and languages, together with their dependencies.

5.5 Case Study

To evaluate our approach we defined a small library of fragments as a case study.[18] Figure 6 shows an overview of the Canons, monadic interfaces, Fragments, and languages that we implemented. Nodes are Agda **module**s, and dashed arrows are **import**s. For each module, we indicate the line count of the corresponding file. In addition to the Mem interface from Section 5.4, we assume three additional monadic interfaces: Lambda (which provides operations for function abstraction and application), General (which provides an operation for general recursion), and Except (which provides operations for throwing and catching exceptions). It is possible to construct many more languages than shown in the figure, since any unique combination of fragments can be composed into a unique language.

The combination of ML-style references and functions means that, even without assuming the General interface, we can encode general recursion using Landin's Knot. Thus, our interpreter must also be able to assign a semantics to non-terminating programs. A common technique for representing (potentially) non-terminating computations in a total language like Agda, that we also use for our case study, is to use a *fueled interpreter* [Amin and Rompf 2017; Owens et al. 2016]. A computation that may either return an A or diverge, is represented as a function of type $\mathbb{N} \to \text{Maybe } A$ that returns a just if it finishes computing before running out of fuel, and nothing otherwise.

The type of fragment composition necessitates that we define all the fragments from this case study using the same semantic domain. To support a fragment for ML-style references, this must be the same semantic domain that we developed in Section 5.4, hence, we cannot use the exact fragments for the simply-typed lambda calculus and exceptions that we defined in Section 5.2 and Section 5.3. Instead we must re-define them to interpret into the new semantic domain. To extract an interpreter for composed languages, we must also provide a monad that instantiates monadic interfaces, such as the Mem interface in Figure 5. Our case study achieves this by using a monad that simultaneously instantiates *all* of the interfaces in Figure 6.

The Lambda fragment maps a lambda expression to a monadic operation which accepts a monadic operation (the function body) as input, and produces a closure value as output. Choosing a monad that has *both* this operation *and* satisfies the Mem interface discussed in Section 5.4 requires some care. We cannot use Agda functions to represent function values, as this results in a *recursive domain equation*: a mutual dependency between values and stores which Agda's termination checker will

[18] The case study is available as part of the accompanying artifact [Van der Rest et al. 2022].

(rightly) reject. To solve this problem, we adopt a flavor of effect handler that is similar to the *latent effect handlers* of Van den Berg et al. [2021].

5.6 Discussion

In this section, we set out to show that our approach to intrinsically-typed and intrinsically compositional language fragments can be transported from the rather simple semantic domain for which we presented the ingredients in Section 4 to a much richer semantic domain from the state-of-the-art in intrinsically-typed interpreters. The key idea of our approach is that the extrinsic proof obligations of both type safety and fragment composition can be made intrinsic to fragment well-typing. We argued that the benefits of this approach are that it helps language component developers to get their semantics right from the start, and that it reduces code size. Our case study indeed demonstrated that these benefits are upheld when we employ the same approach in the relevant semantic domain.

At the same time, the case study also showed that the specifics of particular semantic domains come with their own challenges regarding compositionality. Furthermore, reuse is limited to the chosen semantic domain, which is selected upfront. To use the presented model for compositional fragments as the underpinning for a meta-language for DSL development, one needs to select a class of domains that is expressive enough to cover a large set of desired DSLs. Making this selection and tackling the compositionality challenges that are specific to that class of domains is not addressed by this paper.

6 RELATED WORK

We have presented an approach to constructing type-safe languages from composable, type-safe language fragments. As described in previous sections, our approach builds on *data types à la carte* [Swierstra 2008]. Here, we describe other related work.

Meta-Theory à la Carte. Our work is closely related to previous work on *Meta-Theory à la Carte* (MTC) [Delaware et al. 2013a], *Modular Monadic Meta-Theory* (3MT) [Delaware et al. 2013b], and *Generic Data Types à la Carte* (GDTC) [Keuchel and Schrijvers 2013].

Delaware et al. [2013a] and Keuchel and Schrijvers [2013] focus on *pure* language features, and support binders by using *parameterized higher-order abstract syntax* [Chlipala 2008]. Other than binding, they do not consider effectful language features. Delaware et al. [2013b] extend the MTC approach with effects, using monadic interfaces similar to Mem in Figure 5. They also construct their monads modularly using *monad transformers* [Liang et al. 1995]. We did not build our monad modularly, as that would require composing different domain descriptions, which is an open question.

We need on average 37 LoC (counted with Al Danial's `cloc` tool) to implement verified language fragments. If we also count lines of code for definitions of canonical forms and monadic interfaces, this number approximately doubles. In comparison, MTC and GDTC report needing respectively on average 1100 LoC and 1050 LoC to define and verify similar language features. In other words, the difference in code size between our intrinsically-typed approach and the extrinsic approach found in previous work is about an order of magnitude.

Our framework code is also more concise: we use 859 LoC, whereas MTC uses 2500 LoC, GDTC uses 3500 LoC, and 3MT uses 4400 LoC. We use the Agda Standard Library for lists, relations over lists, functions for working with functions and predicates, and more. Previous works also seem to use Coq's standard library, but perhaps to a lesser extent.

Some of the difference in code sizes can be ascribed to code that deals with "wrong" cases, since these cases are absent in intrinsically-typed interpreters. Our language fragment abstraction also

saves code when composing languages. This abstraction does not exist in the MTC frameworks, where language features are the sum of their parts (i.e., types, expressions, values, type system, interpreter, and type safety proof). In our framework all of these parts are summed using a single operation: fragment composition.

Generic Programming and Meta-Theory. A key technique in our work is to encode data type descriptions as signatures. The Signature type we used as our data type description is known as a *container* [Abbott et al. 2005; Altenkirch et al. 2015]. The *universe of syntaxes* encoding [Chapman et al. 2010; Dagand 2013] (which we will call just "universe encoding") encodes data type descriptions in a different way but encodes the same class of data types as containers do. The difference is in how the encoding is defined: a universe encoding is akin to the syntactic representation of a grammar, whereas the container encoding corresponds to a bag of symbols with associated arities. For generic programming applications that operate on the *syntax of data types itself*, the universe encoding is often preferred. For example, *ornamentation* is a technique for "decorating" data types with additional structure [Dagand 2017; Dagand and McBride 2014; Ko and Gibbons 2017; McBride 2011].

Allais et al. [2018] use the universe encoding to implement a generic framework for *syntaxes with variable binding*, enabling binding-aware generic programming, with generic correctness guarantees. Whereas our paper focuses on the problem of defining an intrinsically-typed interpreter in a modular way, the work of Allais et al. [2018] suggests a promising direction for meta-theoretical reasoning about object languages and interpreters defined using similar generic programming techniques as our framework. Whether it is possible to apply the generic meta-theoretical reasoning techniques of Allais et al. [2018] to the style of modular intrinsically-typed interpreters we develop in this paper is an interesting question for future work.

Final tag-less interpreters [Carette et al. 2009] represent a different flavor of generic programming for implementing type safe definitional interpreters. In particular Carette et al. [2009] showed that it is possible to implement type safe interpreters by modeling object language expression constructors as an abstract interface which, behind the scenes, is constructing a meta-language program that corresponds to interpreting the object language expression. The resulting interpretation of the object language expression is type safe because meta-language programs are type safe. *Object algebras* [d. S. Oliveira and Cook 2012] are based on a similar idea, but for object oriented meta-languages. Bahr and Hvitved [2012] extend the final tag-less approach to intrinsically-typed definitional interpreters in Haskell. Their intrinsically-typed interpreter does not treat value types as open, but does support open object language types, by using Haskell types as object language types. A similar result was achieved by Parreaux et al. [2019] in Scala. It is unclear how the final tag-less approach can be used to define type safe semantics of languages with effects that are not built into the meta-language.

Other Approaches to Modular Semantics and their Proofs. Cimini et al. [2020] extrinsically verify type safety of language specifications by developing a meta type-system for type-safe language specifications. A (meta) type-correct specification yields a type-safe semantics. Language definitions are monolithic, and cannot be constructed from separately checked fragments. Their approach uses reduction semantics, which has some modularity issues. For example, if we add support for exception handlers, we need to copy-paste the current reduction context definition to express a context up-to the closest handler. On the other hand, their approach make it easy to verify type safety of new extensions, since that is done automatically by their meta-language type checker.

Schwaab and Siek [2013] present an Agda formalization of a modular extrinsic type safety proof for a small-step operational semantics defined using a variant [Norell 2008] of the universe encoding due to Chapman et al. [2010]; Dagand [2013] discussed above. The approach is closely related to the extrinsic modular proofs found in MTC [Delaware et al. 2013b], but is based on modular

progress/preservation lemmas about a small-step transition relation. A direct comparison with MTC is difficult because of the difference in meta- and object language (Schwaab and Siek [2013] encode a simple object language with just numbers and lists), but we expect that progress/preservation style requires more LOC per feature. On the other hand, progress/preservation lemmas are a time tested paradigm for type safety proofs. As shown by Wadler et al. [2020], it is possible to marry the intrinsically-typed approach with the small-step style. We expect that it is possible to modularize intrinsically-typed small-step semantics using similar techniques as we use in our framework, but leave verification of this expectation for future research.

Modular Structural Operational Semantics (MSOS) [Mosses 2004] is a framework for modularly specifying small-step operational semantics. Madlener et al. [2011] show how to implement this framework in the Coq proof assistant, and how to do modular proofs about the small-step transition relation. In a related line of work, Torrini and Schrijvers [2015] describe a different method for modular proofs in Coq, and provide, as case study, a modular extrinsic type safety proof about a modularly-specified small-step transition relation. Churchill et al. [2015] use an extension of MSOS [Churchill and Mosses 2013] to modularly specify the static and dynamic semantics of a collection of individual fundamental programming constructs, but do not establish type safety.

Interaction Trees [Xia et al. 2020] are a general purpose coinductive data structure for representing effectful and non-terminating computations based on the *freer monad* [Kiselyov and Ishii 2015; Kiselyov et al. 2013]. As such they support modular reasoning by defining the "visible events" (i.e., effects) of a computation as a functor signature, with the type of interaction trees being closed under the co-product of these signatures. In later work, Zakowski et al. [2021] used interaction trees to give a formal semantics for LLVM IR. The flavor of effect handlers that we used to define a semantics for MiniML in Section 5.5 bears some semblance to interaction trees, but to explore their relation in more detail is a subject of further study.

In his popular textbook on *Types and Programming Languages*, Pierce [2002] makes use of *canonical forms* lemmas to make type safety proofs *robust* as new language fragments are gradually introduced. He attributes the idea of using canonical forms lemmas to Bob Harper (no citation given). We have shown how to define interpreters and language fragments in a way that guarantees that the subset of values that an interpreter- or language fragment knows about is guaranteed to be canonical, essentially making language fragments *robust by construction*.

7 CONCLUSION

In this paper, we presented a framework for defining composable and safe by construction language fragments that can be checked in isolation and safely reused to build type safe languages. This makes it easier to develop and reuse interpreters that do not go wrong, and reduces the overhead traditionally associated with modular verification of type safety. This makes mechanized meta-theory available to a wider audience of DSL developers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their extensive comments and feedback, which helped to improve the paper enormously. Furthermore, we thank Benedikt Ahrens for helpful discussion about the generalization of language fragments to domain descriptions. This research was partially funded by the NWO VENI Composable and Safe-by-Construction Programming Language Definitions project (VI.Veni.192.259).

REFERENCES

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. USA. AAI3136691.

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References A Stratified Semantics of General References. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 75. https://doi.org/10.1109/LICS.2002.1029818

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.* 2, ICFP (2018), 90:1–90:30. https://doi.org/10.1145/3236785

Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S095679681500009X

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. https://doi.org/10.1145/3009837.3009866

Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL (2018), 16:1–16:34. https://doi.org/10.1145/3158104

Patrick Bahr and Tom Hvitved. 2012. Parametric Compositional Data Types. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012 (EPTCS)*, James Chapman and Paul Blain Levy (Eds.), Vol. 76. 3–24. https://doi.org/10.4204/EPTCS.76.3

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components: Extended Version. *CoRR* abs/2108.11155 (2021). arXiv:2108.11155 https://arxiv.org/abs/2108.11155

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 3–14. https://doi.org/10.1145/1863543.1863547

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

Martin Churchill and Peter D. Mosses. 2013. Modular Bisimulation Theory for Computations and Values. In *FOSSACS 2013 (LNCS)*, Frank Pfenning (Ed.), Vol. 7794. Springer, 97–112. https://doi.org/10.1007/978-3-642-37075-5

Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. *LNCS Trans. Aspect Oriented Softw. Dev.* 12 (2015), 132–179. https://doi.org/10.1007/978-3-662-46734-3_4

Matteo Cimini, Dale Miller, and Jeremy G. Siek. 2020. Extrinsically typed operational semantics for functional languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 108–125. https://doi.org/10.1145/3426425.3426936

Jesper Cockx. 2017. Dependent Pattern Matching and Proof-Relevant Unification. (2017). https://lirias.kuleuven.be/handle/123456789/583556

Thierry Coquand. 1992. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*. Citeseer, 71–83.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

Pierre-Évariste Dagand. 2013. *A cosmology of datatypes : reusability and dependent types.* Ph.D. Dissertation. University of Strathclyde, Glasgow, UK. http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713

Pierre-Évariste Dagand. 2017. The essence of ornaments. *J. Funct. Program.* 27 (2017), e9. https://doi.org/10.1017/S0956796816000356

Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *J. Funct. Program.* 24, 2-3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013a. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 207–218. https://doi.org/10.1145/2429069.2429094

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. 2013b. Modular monadic meta-theory. (2013), 319–330. https://doi.org/10.1145/2500365.2500587

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *ACM SIGPLAN international conference on Functional Programming (ICFP)*. 143–155. https://doi.org/10.1145/2034773.2034796

Robert Harper. 1994. A Simplified Account of Polymorphic References. *Inf. Process. Lett.* 51, 4 (1994), 201–206. https://doi.org/10.1016/0020-0190(94)90120-1

Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*, Jacques Carette and Jeremiah Willcock (Eds.). ACM, 13–24. https://doi.org/10.1145/2502488.2502491

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. https://doi.org/10.1145/2804302.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, Chung-chieh Shan (Ed.). ACM, 59–70. https://doi.org/10.1145/2503778.2503791

Hsiang-Shang Ko and Jeremy Gibbons. 2017. Programming with ornaments. *J. Funct. Program.* 27 (2017), e2. https://doi.org/10.1017/S0956796816000307

Saul A Kripke. 1963. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly* 9, 5-6 (1963), 67–96.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. 2011. Formal Component-Based Semantics. In *Proceedings Eight Workshop on Structural Operational Semantics 2011, SOS 2011, Aachen, Germany, 5th September 2011 (EPTCS)*, Michel A. Reniers and Pawel Sobocinski (Eds.), Vol. 62. 17–29. https://doi.org/10.4204/EPTCS.62.2

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. (2011). Unpublished manuscript.

Lambert G. L. T. Meertens. 1992. Paramorphisms. *Formal Aspects Comput.* 4, 5 (1992), 413–424. https://doi.org/10.1007/BF01211391

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Peter D. Mosses. 2004. Modular structural operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.), Vol. 5832. Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 1–2. https://doi.org/10.1145/1481861.1481862

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. Towards improved GADT reasoning in Scala. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom (Eds.). ACM, 12–16. https://doi.org/10.1145/3337932.3338813

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-Typed Definitional Interpreters à la Carte (artifact). https://doi.org/10.5281/zenodo.7074690

Christopher Schwaab and Jeremy G. Siek. 2013. Modular type-safety proofs in Agda. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard (Eds.). ACM, 3–12. https://doi.org/10.1145/2428116.2428120

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Paolo Torrini and Tom Schrijvers. 2015. Reasoning about modular datatypes with Mendler induction. In *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015 (EPTCS)*, Ralph Matthes and Matteo Mio (Eds.), Vol. 191. 143–157. https://doi.org/10.4204/EPTCS.191.13

Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda.* http://plfa.inf.ed.ac.uk/20.07/

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572