Path Dependent Types with Path-Equality

Jaemin Hong
KAIST
Republic of Korea

Jihyeok Park
KAIST
Republic of Korea

Sukyoung Ryu
KAIST
Republic of Korea

Abstract

While the Scala type system provides expressive features like objects with type members, the lack of equality checking between path-dependent types prohibits some programming idioms. One such an example is abstract domain combinators in implementing static analyzers. In this paper, we propose to extend the Scala type system with path-equality, and formalize it as a DOT variant, πDOT, which supports records with type members and fields. We show that πDOT has the normalization property and prove its type soundness.

CCS Concepts **• Software and its engineering → General programming languages**;

Keywords Scala, DOT, path equality

ACM Reference Format:
Jaemin Hong, Jihyeok Park, and Sukyoung Ryu. 2018. Path Dependent Types with Path-Equality. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3241653.3241657

1 Introduction

We have developed static analyzers [Park et al. 2017; Ryu et al. 2018] based on abstract interpretation [Cousot and Cousot 1977] by using various Scala features [Odersky et al. 2016]. In Scala, one can represent an abstract domain as trait `AbsDom` with one type parameter `V` for its concrete domain:

```
trait AbsDom[V] { type Elem <: ElemTrait
                  trait ElemTrait { this: Elem => } }
```

The abstract type `Elem` denotes abstract values. Since `Elem` is a subtype of the inner trait `ElemTrait` and `ElemTrait` has `Elem` as its self-type, methods of `ElemTrait` denote methods of abstract values like a greatest lower bound (`+`). `AbsDom` also has methods like an abstraction function (`alpha`).

Abstract values of different abstract domains may be distinguished by path-dependent types. Consider the following:

```
object A extends AbsDom[Int] { ... }
object B extends AbsDom[Int] { ... }
val a0: A.Elem = A.alpha(0)
val a1: A.Elem = A.alpha(1)
val b0: B.Elem = B.alpha(42)
a0 + a1 // type check
a0 + b0 // type error: A.Elem != B.Elem
```

For two different abstract domains for integers, `A` and `B`, we cannot use elements of `A` as elements of `B`, because they are different types even though they both extend `AbsDom[Int]`. This feature ensures the type safety while static analysis performs diverse operations on abstract values.

Let us consider an abstract domain combinator that combines two or more abstract domains and generates a new abstract domain. A typical example is an abstract pair domain combinator `PairDom` with two type parameters `L` and `R`, respectively denoting the component types of a tuple:

```
trait PairDom[L, R] (val domL: AbsDom[L],
                     val domR: AbsDom[R]) { this: AbsDom[(L, R)] =>
    type Elem <: ElemTrait with PairTrait
    trait PairTrait { this: Elem =>
       def left: domL.Elem = ...
       def right: domR.Elem = ... } }
```

To combine abstract domains, it gets two abstract domains as arguments. `PairTrait` has the `left` and `right` methods, which project the first and second components of a tuple, respectively. Then, by extending `PairDom`, we can define an abstract pair domain of `A` and `B` as follows:

```
object P extends PairDom[Int, Int](A, B) { ... }
val p: P.Elem = P.alpha((0, 1))
val left: P.domL.Elem = p.left
a0 + left // type error: A.Elem != P.domL.Elem
```

When `p` is an abstract value of the domain, we can get the left element of `p` using `left`. However, we cannot use the left element `p.left` as an element of its original domain `A`, because their types `P.domL.Elem` and `A.Elem` are different. The main reason of this problem is because the Scala type system does not support path-equality between path-dependent types. The type checker cannot infer that `P.domL` equals to `A`.

In this paper, we define πDOT, which provides records with type members and fields, and prove its type soundness.

2 Formalization of πDOT

We design πDOT based on μDOT [Amin et al. 2014] rather than DOT [Rompf and Amin 2016] to focus on path-equality.

2.1 Syntax

Figure 1 shows πDOT, which removes self-references and methods in μDOT but adds fields and let-binding as discussed

term	s, t, u	$::=$	$x \mid t.f \mid \mathsf{new}\ (\overline{I}) \mid \mathsf{let}\ x : T = t\ \mathsf{in}\ t$
member init.	I	$::=$	$\tau \mid \mathsf{val}\ f = t$
type member decl.	τ	$::=$	$\mathsf{type}\ L = T\ ..\ T \mid \mathsf{type}\ L <: T$
type	S, T, U	$::=$	$\{\overline{D}\} \mid p.L$
member decl.	D	$::=$	$\tau \mid \mathsf{val}\ f : T$
path	p, q	$::=$	$x \mid p.f$

Figure 1. Syntax of πDOT

$$\frac{\Gamma; \Psi \vdash t : T \quad t\ \mathsf{implies}\ \langle\overline{\rho \equiv \pi}\rangle \qquad \mathsf{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi' \quad \Gamma, x : T; \Psi' \vdash s : U \quad \Gamma; \Psi \vdash U}{\Gamma; \Psi \vdash \mathsf{let}\ x : T = t\ \mathsf{in}\ s : U}$$

Figure 2. Static semantics of let-binding

in Appendix A. Metavariables x and y range over let-bound variables, f over fields, and L over type members.

A path is a let-bound variable or one or more field accesses starting from a let-bound variable. For brevity, we assume that variables are uniquely named. πDOT has two kinds of types: record types $\{\overline{D}\}$ and type selections $p.L$. A record type has zero or more member declarations that consist of type member declarations τ and field declarations. A type member L is declared with both lower and upper bounds as in type $L = T\ ..\ T$ or only with an upper bound as in type $L <: T$. A field declaration specifies the name and type of a field. A type selection $p.L$ is an access to the type member L in a record, which is bound to the given path p. Note that fields enable general paths in type selection.

A term is one of a variable, field access, record initialization, and let-binding. A record initialization consists of zero or more member initializations. A member initialization is either a type member declaration, or a field initialization.

2.2 Dynamic Semantics

Similarly for μDOT, we define the dynamic semantics of πDOT in a big-step style. We show the total evaluation rules of form $\Sigma \vdash t \Downarrow r$ in Figure 7 in Appendix A.2. As usual, a run-time environment Σ maps variables to their values. Since only a record initialization generates a value in πDOT, a value is a record, which maps fields to their values $\langle\overline{f : v}\rangle$. A result r of evaluating a term t is either a value v or stuck.

Because πDOT has the normalization property, we do not specify numbers of evaluation steps and timeout cases.

Theorem 2.1 (Normalization). *For any Σ and t, $\Sigma \vdash t \Downarrow r$, that is, evaluating t never diverges.*

We can prove the normalization theorem by defining sizes of terms and using induction on sizes of terms.

2.3 Static Semantics

The static semantics of πDOT terms of the form $\Gamma; \Psi \vdash t : T$ describes that under a type environment Γ and a path environment Ψ, the type of the term t is T. A type environment maps variables to their types, and a path environment is a set of sets of paths, which represents a partition of paths. A path environment contains all the paths available under the current type environment, and it partitions the paths using the equality between the values denoted by the paths as an equivalence relation. Thus, two paths denoting the same value belong to the same path set in a path environment.

While constructing path environments, the static semantics uses pseudo paths and nullable paths:

pseudo path	ρ	$::=$	$\cdot \mid f, \rho$
nullable path	π	$::=$	$p \mid \mathsf{null}$

A pseudo path ρ is a sequence of zero or more fields, which denotes a postfix of a path; it does not begin with a let-bound variable. The static semantics uses pseudo paths when it collects path-equality information; because it starts collecting the information from the end of a path, it extends a pseudo path by prepending a field in front of the pseudo path. For example, f_1, f_2 is a pseudo path, which represents $.f_1.f_2$, a postfix of an ordinary path without showing the prefix of the path. It represents a postfix of a path like $x.f_1.f_2$ or $x.f_0.f_1.f_2$. A nullable path π is either a valid path p or null, which is not a path. The static semantics uses nullable paths to indicate that a path is equal to some path or is not equal to any path.

Figure 2 shows the typing rule of πDOT for the let-binding term, which utilizes pseudo paths and nullable paths. Since the typing rules of the other terms are conventional, we include them in Appendix A.3. In order to check the type of let $x : T = t$ in s, the typing rule first checks whether the type of t is indeed T as annotated in the syntax. Then, it extends the given path environment Ψ with newly available paths by using the implies and expand rules discussed below. It then checks the type of s under the extended type environment $\Gamma, x : T$ and the extended path environment. Because the type of s, U, should not contain any paths defined in s to prohibit any leakage of local paths, it checks the well-formedness of U under the originally given environments Γ and Ψ. Finally, the type of s becomes the type of the let-binding term.

The implies function collects paths that equal to newly generated paths. When checking the type of let $x : T = t$ in s, the newly generated paths from t are collected by the implies function: t implies $\langle\overline{\rho \equiv \pi}\rangle$ where $\langle\overline{\rho \equiv \pi}\rangle$ denotes a list of pairs of pseudo paths and nullable paths. Since the implies function does not consider the name of a binding variable x, which is the first component of the newly generated paths, the collected pseudo paths $\overline{\rho}$ represent the newly generated paths without the starting let-bound variable. The collected nullable paths $\overline{\pi}$ denote existing paths that equal to their corresponding newly generated paths.

Figure 3 defines the implies function mutually inductively on both terms and member initializations. We classify terms into four kinds: paths, field accesses on non-path terms, record initializations, and let-binding. Even though πDOT is a small language, which allows to fully determine path information at compile time, we decided not to collect path-equality information from field accesses on non-path terms

$$\boxed{t \text{ implies } \langle\overline{\rho \equiv \pi}\rangle}$$

$$p \text{ implies } \langle \cdot \equiv p\rangle \qquad \text{[PE-PATH]}$$

$$\frac{t \neq p}{t.f \text{ implies } \langle \cdot \equiv \text{null}\rangle} \qquad \text{[PE-FIELD]}$$

$$\frac{\overline{I \text{ implies } \langle\overline{\rho \equiv \pi}\rangle}}{\text{new } (\overline{I}) \text{ implies } \langle\overline{\overline{\rho \equiv \pi}}, \cdot \equiv \text{null}\rangle} \qquad \text{[PE-NEW]}$$

$$\text{let } x : T = t \text{ in } s \text{ implies } \langle \cdot \equiv \text{null}\rangle \qquad \text{[PE-LET]}$$

$$\boxed{I \text{ implies } \langle\overline{\rho \equiv \pi}\rangle}$$

$$\text{type } L = S \,..\, U \text{ implies } \langle\rangle \qquad \text{[PEI-TYPE-LU]}$$

$$\text{type } L <: U \text{ implies } \langle\rangle \qquad \text{[PEI-TYPE-U]}$$

$$\frac{t \text{ implies } \langle\overline{\rho \equiv \pi}\rangle \quad \forall\, 1 \leq i \leq |\overline{\rho}|.\ \rho'_i = f, \rho_i}{\text{val } f = t \text{ implies } \langle\overline{\rho' \equiv \pi}\rangle} \qquad \text{[PEI-FIELD]}$$

Figure 3. Collection of equal paths

$$\boxed{\text{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi}$$

$$\text{expand}(\Psi, x, \langle\rangle) = \Psi \qquad \text{[EXP-NIL]}$$

$$\frac{\text{make}(x; \rho') = p \quad \psi = \{p\} \quad \Psi' = \Psi \cup \{\psi\} \quad \text{expand}(\Psi', x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi''}{\text{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}, \rho' \equiv \text{null}\rangle) = \Psi''} \qquad \text{[EXP-EQ-NULL]}$$

$$\frac{\begin{array}{c}\text{make}(x; \rho') = p \quad \Psi = \{\overline{\psi}\} \\ \forall\, 1 \leq i \leq |\overline{\psi}|.\ \psi'_i = \psi_i \cup \{p' \mid \text{make}(q; \rho'') = q' \ \wedge\ q' \in \psi_i \\ \wedge\ \text{make}(p; \rho'') = p'\} \\ \Psi' = \{\overline{\psi'}\} \quad \text{expand}(\Psi', x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi''\end{array}}{\text{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}, \rho' \equiv q\rangle) = \Psi''} \qquad \text{[EXP-EQ-PATH]}$$

Figure 4. Expansion of path environment

and let-binding terms. In other words, when a binding term is a field access, we collect path-equality information only when the term is a field access on a path. We made this decision to show that one can tune how much path-equality information to collect. Since we cannot infer all the path-equality information for languages that contain complex features like user inputs and side effects anyway, we decided to collect path-equality information from only trivial cases. For example, we do not collect path-equality information from the following expressions:

$$\text{let } x : T = \text{new (val } f = z).f \text{ in \_} \quad \text{and}$$
$$\text{let } x : T = (\text{let } y : S = z \text{ in } y) \text{ in \_}$$

even though x equals to z in both. On the other hand, we collect path-equality information from the following:

$$\text{let } x : T = y \text{ in \_} \qquad \text{and}$$
$$\text{let } x : T = y.f \text{ in \_}$$

to find that x equals to y and $y.f$, respectively.

For a variable or a field access on a path p, [PE-PATH] generates a singleton list, which describes that p is an equal path. For a field access on a non-path term and a let-binding term, [PE-FIELD] and [PE-LET], respectively, generate a singleton list, which describes that no existing paths are equal. For a record initialization, the last [PE-NEW] rule, inductively collects equal paths from the member initializations of the record using the implies function for member initialization. It concatenates all the resulting lists from the inductive implies functions, and appends the information that the binding variable is not equal to any existing path. Appending the empty pseudo path $\cdot \equiv \text{null}$ is necessary because every newly constructed path should be added to a path environment.

The implies function for member initialization collects equal paths only from field initializations; type member declarations do not provide any information. The [PEI-FIELD] rule describes that a path may be assigned to a field of a newly initialized record f. Thus, the rule inductively collects path-equality information for the field, and prepends the field to the inductively collected pseudo paths: $\forall 1 \leq i \leq |\overline{\rho}|.\ \rho'_i = f, \rho_i$. For instance, from the following expression:

$$\text{let } x : T = \text{new (val } f = y) \text{ in \_}$$

the rule infers that $.f$ equals to y and also $x.f$ equals to y.

The $\text{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}\rangle)$ function in Figure 4 expands a given path environment Ψ with paths constructed by the make function defined in Appendix A.3 by prepending the binding variable x to pseudo paths $\overline{\rho}$ generated by the implies function. If a path p is not equal to any existing path, the [EXP-EQ-NULL] rule adds a singleton set $\{p\}$ to Ψ. Otherwise, because p equals to some existing paths $\overline{q}$, the [EXP-EQ-PATH] rule adds p to the path sets that contain $\overline{q}$. In addition, it also propagates the path-equality information to longer paths that contain p. For example, suppose that x and y are equal, and the value y refers to is $\langle f : \langle\rangle\rangle$. Then, not only x and y are equal, but also $x.f$ and $y.f$ are equal. Therefore, the expand function searches path sets to check whether they contain p or a path can be obtained by extending p.

Subtyping rules of πDOT are shown in Figure 5. Since the subtyping relation between two record types is conventional, we present only the subtyping rules that involve at least one type selection. The [SUB-PATH] rule specifies that two type selections have a subtype relation if their paths are equal and the type members are equal as well. More specifically, the rule declares that if two paths p and q are in the same path set, then a type selection of p is a subtype of the same type selection of q regardless of their lower and upper bounds.

The other three rules compare a type selection and an arbitrary type. They use not only the lower or upper bound of a given path p but also the lower or upper bound of a path that equals to p, which is critical to maintain the subtyping transitivity. If the rules utilize only the bound of a given path, it may break the subtyping transitivity. For example, suppose that x and y denote the same value and the lower bounds of

$\boxed{\Gamma; \Psi \vdash T <: T}$

$$\frac{\psi \in \Psi \quad p, q \in \psi}{\Gamma; \Psi \vdash p.L <: q.L} \quad \text{[Sub-Path]}$$

$$\frac{\psi \in \Psi \quad p, q \in \psi \qquad \Gamma; \Psi \vdash q : \{\text{type } L = S \,..\, U\} \quad \Gamma; \Psi \vdash U <: T}{\Gamma; \Psi \vdash p.L <: T} \quad \text{[Sub-Type-LU-L]}$$

$$\frac{\psi \in \Psi \quad p, q \in \psi \qquad \Gamma; \Psi \vdash q : \{\text{type } L = S \,..\, U\} \quad \Gamma; \Psi \vdash T <: S}{\Gamma; \Psi \vdash T <: p.L} \quad \text{[Sub-Type-LU-R]}$$

$$\frac{\psi \in \Psi \quad p, q \in \psi \qquad \Gamma; \Psi \vdash q : \{\text{type } L <: U\} \quad \Gamma; \Psi \vdash U <: T}{\Gamma; \Psi \vdash p.L <: T} \quad \text{[Sub-Type-U]}$$

Figure 5. Subtype relation

```
let  $dom$ : {type $E$ = $T$ .. $T$} = new (type $E$ = $T$ .. $T$) in
    let  $pair$ : {val $domL$ : {type $E$ <: $T$}} =
              new (val $domL$ = $dom$) in
        let  $elem$ :  $dom.E$ = new() in
            let $elemL$ :  $pair.domL.E$ = $elem$ in $elemL$
where   $T$ = {}
```

Figure 6. Example πDOT code

$x.L$ and $y.L$ are S and U, respectively, where $\Gamma; \Psi \vdash S <: U$ does not hold. Then, while $\Gamma; \Psi \vdash S <: x.L$ and $\Gamma; \Psi \vdash x.L <: y.L$ are true, $\Gamma; \Psi \vdash S <: y.L$ is not, since S is not a subtype of U, the lower bound of $y.L$, which breaks the subtyping transitivity. In πDOT, we can show that $\Gamma; \Psi \vdash S <: y.L$ holds by replacing y with x, which belongs to the same path set with y, so that we can have the subtyping transitivity.

2.4 Expressiveness

The code example in Figure 6 shows the expressivity of πDOT. If πDOT does not support path-equality, then type checking the code fails because the value of *elem* of type *dom.E* cannot be assigned to *elemL* of type *pair.domL.E*. Although *dom* and *pair.domL* refer to the same record, if πDOT does not collect the path information, and the lower bounds of *dom.E* and *pair.domL.E* which are T and Bottom, respectively, it cannot show $\Gamma; \Psi \vdash dom.E <: pair.domL.E$.

However, since the typing rules collect path-equality information, when we check the type of the innermost let-binding, the path environment contains a path set that contains both *dom.E* and *pair.domL.E*. Thus, by [Sub-Path], $\Gamma; \Psi \vdash dom.E <: pair.domL.E$ holds and type checking of the entire code succeeds.

2.5 Type Soundness

Now, we formally state the type soundness theorem of πDOT.

Theorem 2.2 (Type Soundness). *Let a run-time environment Σ, a type environment Γ, and a path environment Ψ are all consistent: $\Sigma : \Gamma; \Psi$. For any term t, if $\Gamma; \Psi \vdash t : T$ and $\Sigma \vdash t \Downarrow r$, then $r = v$ for some v and $\Gamma; \Psi \vdash v : T$.*

The theorem implies that evaluating a well-typed term does not go wrong and the type of the evaluation result equals to the type of the term. Appendix A.4 proves the theorem.

2.6 Discussion

We can encode abstract domain combinators using singleton types of Scala without extending Scala with path-equality:

```
trait PairDom[L, R, DL <: AbsDom[L] with Singleton,
                    DR <: AbsDom[R] with Singleton]
              (val domL: DL, val domR: DR)
              { this: AbsDom[(L, R)] =>
  type Elem <: ElemTrait with PairTrait
  trait PairTrait { this: Elem =>
    val left: domL.Elem
    val right: domR.Elem } }
object P extends PairDom(A, B) { ... }
(a: A.Elem) + (p.left: P.domL.Elem)
```

where `DL` and `DR` are inferred as `A.type` and `B.type` respectively. Where `a` and `p` are elements of `A` and `P` respectively, `P.domL.Elem` is equal to `A.type#Elem`, which is `A.Elem`. Therefore, the code passes type checking.

3 Related Work

In object-oriented languages, objects with type members can implement modules. In traditional module systems, a module may be opaque or transparent. For an opaque module, the type of the module does not reveal the module implementation; it allows modules as first-class values but restricts uses of higher-order modules. On the contrary, the type of a transparent module exposes the implementation of the module; compilers can decide path-equality at compile time, which enables effective uses of higher-order modules while sacrificing modules as first-class values. Manifest types [Leroy 1994] and translucent sums [Lillibridge 1996] propose to mix both opaque and transparent approaches: programmers can choose whether a module exposes its implementation via its type. In manifest types, modules are second-class entities, but modules as translucent sums are first-class values.

In DOT, objects are first-class by default, and users can choose the degree of abstraction as in manifest type and translucent sums. πDOT additionally offers path-equality like transparent module systems, and fields allow encoding of submodules. Path-equality also makes higher-order features like functions and binding objects as fields more useful.

Acknowledgment

This research has received funding from National Research Foundation of Korea (Grants 2017R1A2B3012020 and 2017M3C4A7068177).

References

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 233–249. https://doi.org/10.1145/2660193.2660216

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.*

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

Mark Lillibridge. 1996. *Translucent Sums: A Foundation for Higher Module Systems.* dissertation. Carnegie Mellon University.

Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala: Updated for Scala 2.12* (3rd ed.). Artima Incorporation, USA.

Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion.* IEEE Press.

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 624–641. https://doi.org/10.1145/2983990.2984008

Sukyoung Ryu, Jihyeok Park, and Joonyoung Park. 2018. Towards Analysis and Bug Finding of JavaScript Web Applications in the Wild. *IEEE Software* (2018).

A Appendix

A.1 Self References

We designed πDOT to exclude self-references, which decreases the expressivity of the language but lets us focus on the main problem, path-equality. Supporting self-references in the presence of fields introduces an initialization problem. Unlike methods, fields require evaluation of their initialization terms when their enclosing record is initialized. Thus, if the language supports self-references, an initialization term of a field may refer to other field of the same record, which requires some specific order between initialization of fields.

We discuss two possible ways to address this problem. First, the language can provide a default value when a term refers to an uninitialized field, which is the current approach of Scala. For example, Scala supports 0 and null as the default values for integers and strings, respectively. Then, we would also need to add null to πDOT to provide it as the default record value, which would require more changes. Second, we may want to type check record initialization incrementally: an initialization term can refer to only initialized fields. Then, the typing rule for record initialization should check the property additionally, which may affect the proof of subtyping transitivity as well.

A.2 Total Evaluation

Figure 7 shows the total evaluation rules of πDOT terms. If a term refers to absent variables or fields, the evaluation

$$\boxed{\Sigma \vdash t \Downarrow r}$$

$$\frac{\Sigma = \_, x : v, \_}{\Sigma \vdash x \Downarrow v} \qquad [\text{TE-Var}]$$

$$\frac{\Sigma \vdash t \Downarrow \langle \overline{f : v} \rangle}{\Sigma \vdash t.f_i \Downarrow v_i} \qquad [\text{TE-Field}]$$

$$\frac{\forall i \text{ such that } I_i = \mathsf{val}\ f_i = t_i.\ \Sigma \vdash t_i \Downarrow v_i}{\Sigma \vdash \mathsf{new}\ (\bar{I}) \Downarrow \langle \overline{f : v} \rangle} \qquad [\text{TE-New}]$$

$$\frac{\Sigma \vdash t \Downarrow v \quad \Sigma, x : v \vdash s \Downarrow v'}{\Sigma \vdash \mathsf{let}\ x : T = t\ \mathsf{in}\ s \Downarrow v'} \qquad [\text{TE-Let}]$$

$$\frac{\Sigma = \overline{y : v} \quad \forall\, 1 \le i \le |\overline{y}|.\ x \ne y_i}{\Sigma \vdash x \Downarrow \mathsf{stuck}} \qquad [\text{TE-Var-Stuck}]$$

$$\frac{\Sigma \vdash t \Downarrow \langle \overline{f' : v} \rangle \quad \forall\, 1 \le i \le |\overline{f'}|.\ f \ne f'_i}{\Sigma \vdash t.f \Downarrow \mathsf{stuck}} \qquad [\text{TE-Field-Stuck}]$$

Figure 7. Total evaluation

$$\boxed{\Gamma; \Psi \vdash t : T}$$

$$\frac{\Gamma = \_, x : T, \_}{\Gamma; \Psi \vdash x : T} \qquad [\text{T-Var}]$$

$$\frac{\Gamma; \Psi \vdash t : \{\mathsf{val}\ f : T\}}{\Gamma; \Psi \vdash t.f : T} \qquad [\text{T-Field}]$$

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash t : T \qquad t \text{ implies } \langle \overline{\rho \equiv \pi} \rangle \\ \text{expand}(\Psi, x, \langle \overline{\rho \equiv \pi} \rangle) = \Psi' \\ \Gamma, x : T; \Psi' \vdash s : U \qquad \Gamma; \Psi \vdash U\end{array}}{\Gamma; \Psi \vdash \mathsf{let}\ x : T = t\ \mathsf{in}\ s : U} \qquad [\text{T-New}]$$

$$\frac{\Gamma; \Psi \vdash t : T \quad \Gamma; \Psi \vdash T <: U}{\Gamma; \Psi \vdash t : U} \qquad [\text{T-Sub}]$$

Figure 8. Static semantics of πDOT

of the term becomes stuck, and results in stuck. The [TE-Var-Stuck] and [TE-Field-Stuck] rules describe such cases. Evaluation of a term propagates stuck from any of its subterms, so that evaluation of a term gets stuck when evaluation of any of its subterm gets stuck. We omit rules for propagating stuck in this paper. We show that evaluating a well-typed term never gets stuck in Appendix A.4. If evaluation of a term does not get stuck, it produces a value; the corresponding rules are conventional.

A.3 Complete Static Semantics

Figure 8 presents a comple static semantics of πDOT and Figure 9 defines the make function.

A.4 Type Soundness in Detail

Before discussing the type soundness theorem, we define typing rules of values and run-time environments in Figure 10.

$\boxed{\mathsf{make}(p;\rho) = p}$

$$\frac{}{\mathsf{make}(p;\cdot) = p}\ [\text{MP-Empty}] \qquad \frac{\mathsf{make}(p.f;\rho) = q}{\mathsf{make}(p;f,\rho) = q}\ [\text{MP-Seq}]$$

Figure 9. Construction of paths

$\boxed{\Gamma;\Psi \vdash v : T}$

$$\frac{\forall\, 1 \le i \le |\overline{v}|.\ \Gamma;\Psi \vdash v_i : T_i}{\Gamma;\Psi \vdash \langle\overline{f : v}\rangle : \{\overline{f : T}\}}\ [\text{TV-Record}]$$

$$\frac{\Gamma;\Psi \vdash v : S \quad \Gamma;\Psi \vdash S <: T}{\Gamma;\Psi \vdash v : T}\ [\text{TV-Sub}]$$

$\boxed{\Sigma : \Gamma;\Psi}$

$$\cdot : \cdot;\phi \qquad [\text{RTP-Empty}]$$

$$\frac{\begin{array}{c}\psi = \mathsf{flatten}(\Psi) \quad \psi' = \mathsf{flatten}(\Psi') \quad \psi \subset \psi' \\ \psi'' = \psi \setminus \psi' \quad x \in \psi'' \quad \Sigma' = \Sigma, x : v \\ \forall p \in \psi.\ \nexists\rho.\ \mathsf{make}(x;\rho) = p \\ \forall p \in \psi''.\ \exists\rho.\ \mathsf{make}(x;\rho) = p \\ \Sigma : \Gamma;\Psi \quad \Gamma;\Psi \vdash v : T \\ \forall\psi \in \Psi'.\ \forall p, p' \in \psi.\ \exists n.\ \Sigma' \vdash p \Downarrow v\ \wedge\ \Sigma' \vdash p' \Downarrow v\end{array}}{(\Sigma, x : v) : (\Gamma, x : T);\Psi'}$$

[RTP-Nonempty]

Figure 10. Typing rules of values and run-time environments

The [TV-Record] rule states that the type of a value is a record type, the fields of which have the types of their initialization values. The [TV-Sub] rule specifies the subsumption relation for the type of a value.

The $\Sigma : \Gamma;\Psi$ judgment states that Γ and Ψ are consistent with Σ. A type environment Γ is consistent with Σ, if the domain of Γ contains every variable x in the domain of Σ, and the type of $\Sigma(x)$ is $\Gamma(x)$. A path environment Ψ is consistent with Σ, if Ψ is a refinement of a partition, whose equivalence relation is the value equality, of all the paths available under Σ. In other word, for any two paths, if they are in the same path set, evaluating them results in the same value.

To prove the type soundness theorem, we first need to prove the subtyping transitivity.

Lemma A.1 (Subtyping Transitivity). *For any type environment Γ, types S, T, and U, if $\Gamma;\Psi \vdash S <: T$ and $\Gamma;\Psi \vdash T <: U$, then $\Gamma;\Psi \vdash S <: U$.*

Proving subtyping transitivity for πDOT is simpler than for μDOT, because it does not involve environment narrowing since records in πDOT cannot refer to themselves. Thus, we can prove it by straightforward structural induction on the subtyping rules.

Using subtyping transitivity, we can obtain inversion lemmas for terms and values.

Lemma A.2 (Inversion Lemma for Variables). *For any variable x, a type environment Γ, and a path environment Ψ, if we have $\Gamma;\Psi \vdash x : T$, there exists a type T' such that $\Gamma = \_, x : T'\_$ and $\Gamma;\Psi \vdash T' <: T$.*

Lemma A.3 (Inversion Lemma for Field Accesses). *For any term t, a field f, a type environment Γ, and a path environment Ψ, if we have $\Gamma;\Psi \vdash t.f : T$, then there exists a type T' such that $\Gamma;\Psi \vdash t : \{\mathsf{val}\ f : T'\}$ and $\Gamma;\Psi \vdash T' <: T$.*

Lemma A.4 (Inversion Lemma for Record Initialization). *For any member initializations $\overline{I}$, a type environment Γ, and a path environment Ψ, if we have $\Gamma;\Psi \vdash \mathsf{new}\ (\overline{I}) : T$, then there exist member declarations $\overline{D}$ such that for all $1 \le i \le |\overline{D}|$, $\Gamma;\Psi \vdash I_i : D_i$ and $\Gamma;\Psi \vdash \{\overline{D}\} <: T$.*

Lemma A.5 (Inversion Lemma for Let-Binding). *For any variable x, terms t and s, a type T, a type environment Γ, and a path environment Ψ, if we have $\Gamma;\Psi \vdash \mathsf{let}\ x : T = t\ \mathsf{in}\ S : U$, $\Gamma;\Psi \vdash t : T$ and there exist a type U', pseudo paths $\overline{\rho}$, nullable paths $\overline{\pi}$, and a path environment Ψ' such that t implies $\langle\overline{\rho \equiv \pi}\rangle$, $\mathsf{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi'$, $\Gamma, x : T;\Psi' \vdash s : U'$, $\Gamma;\Psi \vdash U'$, and $\Gamma;\Psi \vdash U' <: U$.*

Lemma A.6 (Inversion Lemma for Values). *For any fields $\overline{f}$, values $\overline{v}$, a type environment Γ, and a path environment Ψ, if we have $\Gamma;\Psi \vdash \langle\overline{f : v}\rangle : T$, then there exist types $\overline{T'}$ such that for all $1 \le i \le |v|$, $\Gamma;\Psi \vdash v_i : T'_i$ and $\Gamma;\Psi \vdash \{\overline{f : T'}\} <: T$.*

Thanks to subtyping transitivity, we do not need to consider multiple subsumptions in the inversion lemmas: multiple subsumptions collapse into a single subsumption because subtyping is transitive. We use the inversion lemmas during the inductive proof of the type soundness theorem.

Finally, we should prove that the implies and expand functions are sound. Soundness of the functions denotes that if Ψ is consistent with Σ, the expanded path environment Ψ' by the functions is still consistent with its correspondingly extended run-time environment Σ'. To prove the soundness of the entire expansion process, we first prove the soundness of the implies function and then prove the soundness of πDOT.

Lemma A.7 (Soundness of the implies Function). *For any term t, a variable x, and a run-time environment Σ, suppose that $\Sigma \vdash t \Downarrow v$, t implies $\langle\overline{\rho \equiv \pi}\rangle$, and $\forall 1 \le i \le |\overline{p}|$. $\mathsf{make}(x;\rho_i) = p_i$. Then, for all $1 \le i \le |\overline{p}|$, there exists v'_i such that $\Sigma, x : v \vdash p_i \Downarrow v'_i$. Moreover, if there exists a path q_i such that $\pi_i = q_i$, then $\Sigma \vdash q_i \Downarrow v'_i$.*

Lemma A.8 (Soundness of Path Environment Expansion). *Let a run-time environment Σ and a path environment Ψ are consistent: $\Sigma : \Psi$. For any term t, pseudo paths $\overline{\rho}$, and nullable paths $\overline{\pi}$, if we have $\Sigma \vdash t \Downarrow v$, t implies $\langle\overline{\rho \equiv \pi}\rangle$, and $\mathsf{expand}(\Psi, x, \langle\overline{\rho \equiv \pi}\rangle) = \Psi'$, then $(\Sigma, x : v) : \Psi'$.*