Dependent Object Types with Implicit Functions

Alex Jeffery
University of Sussex
A.P.Jeffery@sussex.ac.uk

Abstract

DOT (Dependent Object Types) is an object calculus with path-dependent types and abstract type members, developed to serve as a theoretical foundation for the Scala programming language. As yet, DOT does not model all of Scala's features, but a small subset. We present the calculus DIF (DOT with Implicit Functions), which extends the set of features modelled by DOT to include implicit functions, a feature of Scala to aid modularity of programs. We show type safety of DIF, and demonstrate that the generic programming focused use cases for implicit functions in Scala are also expressible in DIF.

Keywords implicits, implicit parameters, implicit functions, type classes, calculus, objects, dependent object types, DOT, Scala, Dotty

1 Introduction

1.1 Implicit functions

Modularity, a core concept in software engineering, is greatly aided by parameterisation of programs. Parameterisation has dual facets: supplying and consuming a parameter. A key tension in large-scale software engineering is between *explicit* (e.g. pure functional programming), and *implicit* parameterisation (e.g. global state). The former enables local reasoning but can lead to repetitive supply of parameters. Here is a simple example of the problem:

```
def compare(x: Int, y: Int)(comparator:
    Int => Int => Boolean): Boolean =
  comparator(x)(y)
...
compare(3, 4)(<=)
...
compare(17, 12)(<=)
...
```

Repeatedly passing functions like <= which are unlikely to change frequently, is tedious, and impedes readability of large code bases. Default parameters are an early proposal for mediating this tension in a type-safe way. The key idea is to annotate function arguments with their default value, to be used whenever an invocation does not supply an argument:

```
def compare(x: Int, y: Int)(comparator:
    Int => Int => Boolean = <=): Boolean =
  comparator(x)(y)
...
compare(3, 4)
...
compare(17, 12)
...
```

The compiler synthesises `compare(3, 4)(<=)` from `compare(3, 4)`, and `compare(17, 12)(<=)` from `compare(17, 12)`. The missing argument indicates to the compiler that the default <= should be used. Default parameters have a key disadvantage: the default value is hard-coded at the callee, and cannot be context dependent. Implicit arguments, a strict generalisation of default parameters, were pioneered in Haskell [8], and popularised as well as refined in Scala [11]: they separate the *callee's declaration* that an argument can be elided, from the *caller's choice* of elided values, allowing the latter to be context dependent.

```
def compare(x: Int, y: Int)(implicit comparator:
    Int => Int => Boolean): Boolean =
  comparator(x)(y)
...
implicit val cmp = <=
compare(3, 4)
...
implicit val cmp = >
compare(17, 12)
...
```

In this example `compare(3, 4)` is rewritten as above, but `compare(17, 12)` becomes `compare(17, 12)(>)`, i.e. a different implicit argument is synthesised. The disambiguation between several providers of implicit arguments happens at compile-time using type and scope information. Programs where elided arguments cannot be disambiguated at compile-time are rejected as ill-formed. Hence type-safety is not compromised.

Scala 2019, July 15–19, London, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Type classes

Type classes [6, 16] are a mechanism to allow ad-hoc polymorphism in languages with parametric polymorphism. Parametrically polymorphic type variables may be *constrained*, such that they can only be instantiated by types over which a user-defined set of functions exist. Further functions can then be defined over all types for which the set of functions are defined. A typical example of a type class is `Show`. Note that for this example we use Haskell-style code, as Haskell is the most well-known language with type classes. In a program, we might want to use a function `print` to print values over many different types, for example we might want to write `print 1`, `print True` and `print [1, 2]` in the same program. To acheive this with type classes, we give `print` the following signature:

```
print :: Show a => a -> IO ()
```

This signature tells us that where `a` is a type in the `Show` type class, `print` takes an `a` and returns `IO ()` (in Haskell, `()` is the unit type and the type constructor `IO` represents a type obtained as a result of performing IO). The definition of the `Show` type class itself is given below:

```
class Show a where
    show :: a -> String
```

We call this a *class definition*. It tells us that a type `a` is in the `Show` type class when there exists a function `show` of type `a -> String`. We can then define a function for a given type, for example `Bool`, as below, in an *instance definition*:

```
instance Show Bool where
    show b = if b then "True" else "False"
```

And for `Int`, assuming a function `intToString :: Int -> String` that converts an `Int` to its `String` representation:

```
instance Show Int where
    show i = intToString i
```

We can also declare type constructors to be in a type class when their argument types are also in the type class. For example, we can declare that tuples of type `(a, b)` for all `a`, `b` are in `Show` when `a` and `b` are in `Show` thusly:

```
instance (Show a, Show b) => Show (a, b) where
    show (a, b) = "(" ++ show a ++ ", " ++
                                show b ++ ")"
```

With these instance definitions at hand, we can give a definition for `print`, which will satisfy the type checker for arguments of any type in the `Show` type class, as above. The definition is as follows:

```
print a = putStrLn (show a)
```

Type classes are usually implemented by via a technique known as *dictionary passing* [7]. Function definitions whose type contains a type class constrained type variable are adapted to take an additional parameter for each such type variable. That additional parameter is known as a *dictionary*, and is a tuple containing the type class function implementations for the type that instantiates the type class constrained type variable. When a type class declares just a single function, such as with `Show`, the dictionary is just a single function rather than a tuple of functions. Call sites for those function definitions are augmented to pass the dictionary of appropriate type - the compiler decides what the appropriate type is based on the parameter passed at the call site.

In the case of the `Show` type class above, we would expect the `Bool` instance definition to be translated into the following dictionary:

```
showBoolDict :: Bool -> String
showBoolDict = show where
    show b = if b then "True" else "False"
```

The `print` function is then translated to the following:

```
print :: (a -> String) -> a -> IO ()
print dict a = putStrLn (dict a)
```

Call sites `print b` where `b` is of type `Bool` are then rewritten as `print showBoolDict b`.

1.3 Dependent Object Types (DOT)

DOT [1–3, 9, 15] is a foundational calculus intended as a step towards a theoretical foundation for the programming language Scala and its type system. DOT models a restricted subset of Scala – the base-calculus includes Scala's key features from a type theory perspective, such as path-dependency, abstract type members and a subtyping hierarchy with maximal type $\top$ and minimal type $\bot$. Omitted are features of Scala such as traits, classes and inheritance. We now look briefly at two important features of DOT.

Abstract Type Members Abstract type members are a feature of Scala that allow for generic programming. In Scala, a trait, class or object may declare an abstract type member. The trait, class or object may then declare or define methods over that type. Below is a trait `A` with an abstract type member `B`. The trait also contains a method member `consumeB` that consumes a value of type `B`.

```
trait A {
    type B
    def consumeB(b: B): String
}
```

Any object may then declare itself a subtype of `A`, and provide a concrete type in place of the abstract type member. Such an object may implement method members over that type:

```
object C extends A {
    type B = Int
    def consumeB(b: Int): String =
      b.toString
}
```

Like Scala, DOT also includes abstract type members, and thus the above pattern is possible. We write $\{A : S..S'\}$ to denote an object with a type member A, whose lower bound

is S and upper bound is S'. Scala's syntax `type T = U` is then equivalent to DOT's $\{T : U..U\}$ - we let `U` be the upper and lower bound. In DOT we can define a completely abstract type member (equivalent to just `type T` in Scala) by using $\bot$ and $\top$ as lower and upper bounds respectively. Conversely we can define a fully specified type member by using a single specified type as both the lower and upper bound.

Path-dependent Types Path-dependent types are a restricted form of dependent types [9]. Instead of allowing arbitrary computations over values in types, objects with type members are the only values allowed, and selection of type members is the only permitted operation on those objects.

Path-dependent types with function arrows can be used to recover Hindley-Damas-Milner style polymorphism despite the absence of type variables, via the passing of an object with a type member [9]. Consider the function `id` in a Haskell-like language:

```
id :: a → a
id x = x
```

We can rewrite this in DOT, with an additional parameter used to pass the type of the parameter x. In DOT, $\forall(y : T)T'$ is the type of a path-dependent function, and $\lambda(y : T)t$ is a function literal, where y is a variable, T, T' are types and t is a term. The encoding of a polymorphic `id` in DOT is given below:

```
id : ∀(u: {A: S}) ∀(v: u.A) u.A
id = λ(u: {A: S}) λ(x: u.A) x
```

1.4 Dependent object types with Implicit Functions (DIF)

Implicits are widely used in Scala. They are valuable to programmers as a mechanism for passing context without excessive verbosity. The *Dotty* Scala compiler [10], written in Scala, contains at more than 5000 occurrences of the `implicit` keyword. Akka [4], a popular Scala library for Actor-based concurrency, uses implicits at the core of its API. Therefore it is of importance to the Scala community to have a solid theoretical foundation for the correctness of Scala's implicit language constructs. Indeed the safety of implicit functions in Scala has been evidenced by the type-safe integration of implicit functions into lambda calculus [11]. This evidence could be further strengthened by their type-safe integration into DOT, the calculus on which the Dotty compiler is based. In the remainder of this paper, we present DIF, a type-safe integration of implicit functions into DOT. DIF is shown to be type-safe by translation into the DOT calculus presented in [9]. We demonstrate that the type class pattern [12], a typical use case of implicit functions in Scala, can be translated typably into DIF.

Example The type class pattern [11] in DIF maps closely to the type class pattern as is used in Scala. The example leverages DOT's path-dependency, which exhibits parametric polymorphism by passing objects with abstract type members. The dictionary passing of languages with type classes can then be implemented with implicit functions. Example 1 shows the type class pattern in Scala, and example 2 shows the type class pattern adapted for DIF. In example 1, lines 1–3 represent the class definition - we declare a type class `Ord` with a single definition, `compare`, which compares two values of type `A`. Line 5 represents a type class constrained polymorphic function definition, like `print` in our earlier example. Lines 7-9 are is instance definition for `Int`, and line 11 shows an example call to a type class constrained polymorphic function. Example 2 encodes the class declaration across lines 1-5. In place of the type variable in the trait declaration, the DIF encoding includes an abstract type member `A`. The encoding of the `comp` function includes an additional argument over the Scala version. This additional argument is an object with an abstract type member, which the subsequent arguments can use as their type, since DIF does not have type variables. This additional argument is not to be confused with the dictionary argument `ev`, which both encodings require. Example 2 encodes the instance declaration across lines 9-13. We assign the instance to the implicit variable so that it can be passed implicitly to calls to `comp`, and leverage intersection types to make it a subtype of `Ord`. We specify that `A` is restricted to `Int`, and provide a definition of `compare`. Line 16 shows an example call.

2 The Language DIF

Figure 1 gives the syntax of DIF. The syntax is similar to that of DOT in [9], with the addition of constructs related to implicit functions. We add a new type $\forall\wr(u : S)S'$ which represents an implicit (path-dependent) function from S to S'. We also add the *implicit query* $\wr$, the analogue to Scala's `implicitly[T]`. Occurrences of $\wr$ are *resolved* into variables by our typing and translation rules. The variable that replaces a given occurrence of $\wr$ is chosen from candidate implicit variables, which are bound by let constructs, where $\wr$ is the variable name, i.e. **let** $\wr = t$ **in** t'.

Note that we make a distinction between two sets of variable names. The first, ranged over by x, y is the set of variable names with the implicit query. The second set, ranged over by u, v is the set of variables where $\wr$ is not allowed. At the term level, anywhere we can write a variable name, we could also write an implicit query, so we use x in the grammar of terms. At the type level, however, implicit queries are not allowed in names, and we therefore use u for names in the grammar of types.

2.1 Abbreviations

We employ the following abbreviations in examples:
– We encode multiple argument function (types) as multiple single function (types):

$$\forall(y : T, z : U) \equiv \forall(y : T)\forall(z : U)$$

```
trait Ord[A] {
    def compare(x: A, y: A): Boolean
}

def comp[A](x: A, y: A)(implicit ev: Ord[A]): Boolean = ev.compare(x, y)

implicit def intOrd: Ord[Int] = new Ord[Int] {
    def compare(x: Int, y: Int): Boolean = a < b
}
...
comp(1, 2)
...
```

Example 1. The type class pattern in Scala

```
let ord_package = ν(ord_p) {
    Ord = μ(self: {
        A
        compare: ∀(x: self.A, y: self.A)Boolean
    })
    comp: ∀(ty: {A}, x: ty.A, y: ty.A)∀ι(ev: ord_p.Ord∧{A})Boolean =
        λ(ty: {A}, x: ty.A, y: ty.A)ι.compare(x, y)
} in
let ι: ord_package.Ord∧{A = Int} = ν(self: {
        A = Int
        compare: ∀(x: self.A, y: self.A)Boolean =
            λ(x: self.A, y: self.A) x < y
    })
in
...
ord_package.comp({A}, 1, 2)
...
```

Example 2. The type class pattern in DIF

$$\lambda(y : T, z : U) \equiv \lambda(y : T)\lambda(z : U)$$

– We group intersection contents together inside { curly brackets }, separating definitions with a newline, additional whitespace or semicolon:

$$\{d \quad d'\} \equiv \{d\} \wedge \{d'\}$$
$$\{d\ ;\ d'\} \equiv \{d\} \wedge \{d'\}$$

– We allow terms in application and selection by encoding them as let-bindings:

$$t\ t' \equiv \textbf{let}\ x = t\ \textbf{in}\ x\ t'$$
$$x\ t \equiv \textbf{let}\ y = t\ \textbf{in}\ x\ y$$
$$t.a \equiv \textbf{let}\ x = t\ \textbf{in}\ x.a$$

– We abbreviate type bounds thusly:

$$A <: T \equiv A : \bot..T \qquad A = T \equiv A : T..T$$
$$A >: T \equiv A : T..\top \qquad A \equiv A : \bot..\top$$

– We encode type ascription as application:

$$t : T \equiv (\lambda(x : T)x)t$$
$$\textbf{let}\ x : T = t\ \textbf{in}\ t' \equiv \textbf{let}\ x = t : T\ \textbf{in}\ t'$$

– Finally we omit types in new-bindings if the type of the definition is explicit, writing $\nu(x : T)d$ as $\nu(x)d$.

2.2 Semantics

We define the semantics of DIF by type-based translation from DIF programs to DOT programs. The meaning of a DIF program is therefore given by its typed translation into a DOT program. This translation is the topic of section 3.

3 Typing for DIF

In this section we introduce the typing system for DIF programs. Typing judgements in DIF are 4-place: $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$. This judgement can be read: under environment Γ, the DIF term t has type S, and translated to the DOT term $\widehat{t}$.

Terms

t	::=	$x.a$	*Selection*
	|	x	*Variable*
	|	$x\ y$	*Application*
	|	$\textbf{let}\ x = t\ \textbf{in}\ t'$	*Let-binding*
	|	$\nu(x : S)d$	*Object*
	|	$\lambda(x : S)t$	*Abstraction*

Types

S	::=	$\{a : S\}$	*Field Declaration*
	|	$S \wedge S'$	*Intersection*
	|	$\mu(u : S)$	*Recursive type*
	|	$\top$	*Top*
	|	$\bot$	*Bottom*
	|	$\{A : S..S'\}$	*Type Declaration*
	|	$u.A$	*Type Projection*
	|	$\forall(u : S)S'$	*Dependent Function*
	|	$\forall\wr(u : S)S'$	*Implicit Function*

Definitions

d	::=	$\{A = S\}$	*Type Definition*
	|	$\{a = t\}$	*Field Definition*
	|	$d \wedge d'$	*Aggregate Definition*

Variables (with implicit query)

$x, y\ ::=\ p \quad | \quad i \quad | \quad \wr$

Variables

$u, v\ ::=\ p \quad | \quad i$

Explicit Variables $p, q, \ldots$

Implicit Variables $i, j, \ldots$

Implicit Query $\wr$

Term Members $a, b, \ldots$

Type Members $A, B, \ldots$

Figure 1. Grammar of DIF

3.1 Translation of Types

We define the function $(\bullet)^*$, which translates DIF types into DOT types. DIF types differ from DOT types only in the inclusion of the type for implicit functions $\forall\wr(u : S)S'$. The translation simply erases occurrences of $\wr$, translating implicit functions into explicit functions.

Definition 1 (Translation of types)**.** Figure 2 defines translation from DIF types to DOT types.

$$
\begin{aligned}
x^* &= x & \bot^* &= \bot \\
\{a : S\}^* &= \{a : S^*\} & \{A : S..S'\}^* &= \{A : S^*..S'^*\} \\
(S \wedge S')^* &= S^* \wedge S'^* & (u.A)^* &= u.A \\
\mu(u : S)^* &= \mu(u : S^*) & (\forall(u : S)S')^* &= \forall(u : S^*)S'^* \\
\top^* &= \top & (\forall\wr(u : S)S')^* &= \forall(u : S^*)S'^*
\end{aligned}
$$

Figure 2. Translation from DIF types to DOT types

We extend the definition of $(\bullet)^*$ pointwise to environments Γ.

3.2 Type substitution

Definition 2 (Name substitution on types)**.** We define capture-avoiding substitution of names within types, written $[u := v]S$, in the usual way.

3.3 The functions *depth* and *spec*

The function $\textbf{\textit{depth}}(\Gamma, i, j)$ decides which variable is more deeply nested in the environment Γ, for purposes of disambiguation of implicit variable selection. It returns -1 if i is more deeply nested, and 1 if j is more deeply nested. Its formal definition is given in figure 3.

$$
\textbf{\textit{depth}}(\Gamma, i, j) = \begin{cases}
-1 & \textbf{if}\ \Gamma = \Gamma_1, i : S_1, \Gamma_2, j : S_2, \Gamma_3 \\
 & \wedge \{i, j\} \cap \textbf{dom}(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3) \neq \emptyset \\
1 & \textbf{if}\ \Gamma = \Gamma_1, j : S_1, \Gamma_2, i : S_2, \Gamma_3 \\
 & \wedge \{i, j\} \cap \textbf{dom}(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3) \neq \emptyset \\
\bot & \textbf{otherwise}
\end{cases}
$$

Figure 3. The ***depth*** function

The function $\textbf{\textit{spec}}(\Gamma, i, j)$ decides which variable's type is more specific in the environment Γ, i.e. which type can be instantiated to the other by widening or polymorphic parameter instantiation. It returns -1 if i is more specific, 1 if j is more specific, and if they are equal (i.e. both a subtype and a supertype of each other), 0 is returned. Its formal definition is given in figure 4.

$$
\textbf{\textit{spec}}(\Gamma, i, j) = \begin{cases}
-1 & \textbf{if}\ \Gamma \vdash i <: j \wedge \neg(\Gamma \vdash j <: i) \\
1 & \textbf{if}\ \neg(\Gamma \vdash i <: j) \wedge \Gamma \vdash j <: i \\
0 & \textbf{if}\ \Gamma \vdash i <: j \wedge \Gamma \vdash j <: i \\
\bot & \textbf{otherwise}
\end{cases}
$$

Figure 4. The ***spec*** function

3.4 Typing rules

Figure 5 gives the binding rules for DIF binders. Binding rules are two-place judgements of the form $x \rightsquigarrow u$. Every time a binder is encountered in the typing system, a binding rule will decide if that binder should be replaced with a fresh variable name (in the case that the binder is the implicit query $\wr$) or left unchanged. We allow use of $\wr$ as a binder in several places, for example in binding an implicit variable: **let** $\wr$ = ... **in** .... The rule (Bind-Im) generates a fresh name in such a case, and occurrences of the implicit query to be resolved to the variable at the binder are replaced with the variable generated by the binding rule. The rule (Bind-Ex) leaves explicit variable bindings unchanged.

$$\text{(Bind-Ex)} \quad p \rightsquigarrow p \qquad \text{(Bind-Im)} \quad \frac{i \text{ fresh}}{\wr \rightsquigarrow i}$$

Figure 5. Binding rules for DIF terms

Figure 6 gives the binding rules for DIF terms. Most rules are identical to their corresponding rules in DOT (specifically in [9]), when ignoring the translation part of typing judgements ($\rightsquigarrow \widehat{t}$). The typing rules differ slightly from their DOT counterparts in that they have binding judgements in their premises. The rules (Var-Ex), (All-Ex-I) and (All-Ex-E) are analogous to the rules $(\text{Var})_{DOT}$, $(\text{All-I})_{DOT}$ and $(\text{All-E})_{DOT}$ respectively. The rule (All-Im-I) types introduction of implicit functions, introducing an (explicit) abstraction to the translation. The rule (All-Im-E) types implicit function elimination, inserting an implicit variable of suitable type as an argument to the implicit function. Finally (Var-Im) types implicit query, replacing $\wr$ with a chosen implicit variable in the translation. This rule also performs *disambiguation*, whenever more than one implicit variable is of suitable type. Disambiguation follows the same process as Scala (as specified in [11]). We prefer more deeply nested implicit variables over less deeply nested ones, and implicit variables with more specific types over ones with more general types. Where we have a choice between two variables, one of which is more deeply nested and the other a more specific type, we reject the program as ambiguous.

Figure 7 gives the typing rules for DIF definitions. These are again very similar to the typing rules for DOT defintions, and in each case the translations are homomorphic.

Figure 8 gives the subtyping rules for DIF terms and definitions. These are yet again very similar to the subtyping rules in DOT. The exception is the rule (All-<:-All-Im), which makes explicit functions a subtype of implicit functions. This allows the passing of arguments explicitly to an implicit function, which is an important aspect of implicit functions in Scala, as it allows overriding as necessary any implicit variable the compiler would otherwise insert.

4 Type safety of DIF

We show the type safety of DIF by translation into DOT. Given an environment Γ, a DIF term t, a type S and a DOT term $\widehat{t}$, we show that the judgement $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$ implies the judgement $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$. In other words, if a DOT term is typable under Γ with type S, and translates to $\widehat{t}$, then $\widehat{t}$ is typable in DOT under the translation of Γ with the translation of the type S. The same property also holds for definitions, i.e. $\Gamma \vdash d : S \rightsquigarrow \widehat{d} \Rightarrow \Gamma^* \vdash_{DOT} \widehat{d} : S^*$. We call this property *type preserving translation*. The type safety of DIF depends upon the type safety of DOT, via type preserving translation.

Theorems 1 and 2 show type preserving translation for terms and definitions respectively. We also include some auxiliary lemmas, such as lemma 1, which shows that subtyping is preserved by the translation from DIF types to DOT types. We state our main results below, and full proofs are given in the appendix.

Lemma 1 (Preservation of subtyping under type translation).
If $\Gamma \vdash S <: U$ then $\Gamma^* \vdash_{DOT} S^* <: U^*$.

Theorem 1 (Type-preserving translation of DIF terms).
If $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$ then $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$.

Theorem 2 (Type and domain-preserving translation of DIF definitions).
If $\Gamma \vdash d : S \rightsquigarrow \widehat{d}$ then $\Gamma^* \vdash_{DOT} \widehat{d} : S^*$ and $\mathbf{dom}(d) = \mathbf{dom}(\widehat{d})$.

5 Conclusion

The key aim of DOT is to provide a theoretical foundation for Scala. We extend the existing foundation to include implicits, a feature widely used in Scala programming.

Scala's lack of type classes mean that implicits must be leveraged to achieve such behaviour. We have demonstrated the robustness of this approach with a typable example in a type-sound calculus.

6 Related and Further Work

It is known that type assignment and subtyping are undecidable in DOT [15] since DOT can encode system $F_{<:}$. As DIF is a superset of DOT, these properties hold for DIF transitively. DOT does, however, have a local type inference procedure [14]. It follows that such a procedure should also exist for DIF, especially since local type inference is used to great effect in Scala, a language with implicits. DOT has no principal types, and it is therefore likely that DIF also lacks principal types, although this remains to be shown.

It is well-known that there is a correspondence between implicits and type classes - it is easily seen that the dictionary passing of type classes is simulated with an implicit parameter. This has been observed in particular by [12] and [13].

$$\text{(Var-Im)}\quad \frac{i \in \mathbf{dom}(\Gamma) \quad \forall j \in \mathbf{dom}(\Gamma), j \neq i.\Gamma \vdash \wr : S \rightsquigarrow j \Rightarrow \boldsymbol{depth}(\Gamma, i, j) + \boldsymbol{spec}(\Gamma, i, j) < 0}{\Gamma \vdash \wr : S \rightsquigarrow i}$$

$$\text{(Var-Ex)}\quad \Gamma, p : S, \Gamma' \vdash p : S \rightsquigarrow p \qquad \text{(Sub)}\quad \frac{\Gamma \vdash t : S \rightsquigarrow \widehat{t} \quad \Gamma \vdash S <: S'}{\Gamma \vdash t : S' \rightsquigarrow \widehat{t}} \qquad \text{(And-I)}\quad \frac{\Gamma \vdash x : S \rightsquigarrow \widehat{x} \quad \Gamma \vdash x : U \rightsquigarrow \widehat{x}}{\Gamma \vdash x : S \wedge U \rightsquigarrow \widehat{x}}$$

$$\text{(All-Ex-I)}\quad \frac{x \rightsquigarrow u \quad \Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t} \quad \{x, u\} \cap \mathrm{fv}(S) = \emptyset}{\Gamma \vdash \lambda(x : S)t : \forall(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}} \qquad \text{(All-Ex-E)}\quad \frac{\Gamma \vdash x : \forall(u : S)U \rightsquigarrow \widehat{x} \quad \Gamma \vdash y : S \rightsquigarrow \widehat{y}}{\Gamma \vdash x\ y : [u := y]U \rightsquigarrow \widehat{x}\ \widehat{y}}$$

$$\text{(Let)}\quad \frac{x \rightsquigarrow u \quad \Gamma \vdash t : S \rightsquigarrow \widehat{t} \quad \Gamma, u : S \vdash t' : U \rightsquigarrow \widehat{t'} \quad x \notin \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ t' : U \rightsquigarrow \mathbf{let}\ u = \widehat{t}\ \mathbf{in}\ \widehat{t'}} \qquad \text{(Rec-I)}\quad \frac{\Gamma \vdash x : S \rightsquigarrow \widehat{x}}{\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}}$$

$$\text{({}-I)}\quad \frac{x \rightsquigarrow u \quad \Gamma, u : S \vdash d : S \rightsquigarrow \widehat{d}}{\Gamma \vdash \nu(x : S)d : \mu(u : S) \rightsquigarrow \nu(u : S^*)\widehat{d}} \qquad \text{(Fld-E)}\quad \frac{\Gamma \vdash x : \{a : S\} \rightsquigarrow \widehat{x}}{\Gamma \vdash x.a : S \rightsquigarrow \widehat{x}.a} \qquad \text{(Rec-E)}\quad \frac{\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}}{\Gamma \vdash x : S \rightsquigarrow \widehat{x}}$$

$$\text{(All-Im-I)}\quad \frac{u \text{ fresh} \quad \Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t} \quad u \notin \mathrm{fv}(S)}{\Gamma \vdash t : \forall\wr(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}} \qquad \text{(All-Im-E)}\quad \frac{\Gamma \vdash x : \forall\wr(u : S)U \rightsquigarrow \widehat{x} \quad \Gamma \vdash \wr : S \rightsquigarrow v}{\Gamma \vdash x : [u := v]U \rightsquigarrow \widehat{x}\ v}$$

Figure 6. Typing and translation rules for DIF terms

$$\text{(Typ-I)}\quad \Gamma \vdash \{A = S\} : \{A : S..S\} \rightsquigarrow \{A = S^*\} \qquad \text{(Fld-I)}\quad \frac{\Gamma \vdash t : S \rightsquigarrow \widehat{t}}{\Gamma \vdash \{a = t\} : \{a : S\} \rightsquigarrow \{a = \widehat{t}\}}$$

$$\text{(AndDef-I)}\quad \frac{\Gamma \vdash d_1 : S_1 \rightsquigarrow \widehat{d_1} \quad \Gamma \vdash d_2 : S_2 \rightsquigarrow \widehat{d_2} \quad \mathbf{dom}(d_1) \cap \mathbf{dom}(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : S_1 \wedge S_2 \rightsquigarrow \widehat{d_1} \wedge \widehat{d_2}}$$

Figure 7. Typing and translation rules for DIF definitions

$$\text{(Top)}\quad \Gamma \vdash S <: \top \qquad \text{(Bot)}\quad \Gamma \vdash \bot <: S \qquad \text{(Refl)}\quad \Gamma \vdash S <: S \qquad \text{(Trans)}\quad \frac{\Gamma \vdash S <: U \quad \Gamma \vdash U <: R}{\Gamma \vdash S <: R}$$

$$\text{(<:-And)}\quad \frac{\Gamma \vdash S <: U \quad \Gamma \vdash S <: R}{\Gamma \vdash S <: U \wedge R} \qquad \text{(<:-Sel)}\quad \frac{\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u}{\Gamma \vdash S <: u.A} \qquad \text{(Sel-<:)}\quad \frac{\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u}{\Gamma \vdash u.A <: U}$$

$$\text{(All-<:-All-Ex)}\quad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall(x : S_1)U_1 <: \forall(x : S_2)U_2} \qquad \text{(Typ-<:-Typ)}\quad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{A : S_1..U_1\} <: \{A : S_2..U_2\}}$$

$$\text{(And}_1\text{-<:)}\quad \Gamma \vdash S \wedge U <: S \qquad \text{(And}_2\text{-<:)}\quad \Gamma \vdash S \wedge U <: U \qquad \text{(Fld-<:-Fld)}\quad \frac{\Gamma \vdash S <: U}{\Gamma \vdash \{a : S\} <: \{a : U\}}$$

$$\text{(All-<:-All-Im)}\quad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall\wr(x : S_1)U_1 <: \forall\wr(x : S_2)U_2}$$

Figure 8. Subtyping rules for DIF

It would be interesting to establish this correspondence formally – one can envision a translation $t_{c\to i}$ from type classes to implicits, and given the established results of type classes to lambda calculus $t_{c\to l}$ [16] and implicits to lambda calculus $t_{i\to l}$ [11], it seems that it should be possible to establish $t_{c\to i} \cdot t_{i\to l} \equiv t_{c\to l}$ for some notion of $\equiv$.

It has not been investigated whether implicits can be used to encode type classes. One can envison another translation $t_{i\to c}$ from implicits to type classes, and it should then be expected that $t_{i\to c} \cdot t_{c\to i} \equiv t_{c\to i} \cdot t_{i\to c} \equiv \mathtt{id}$, again for some $\equiv$.

Implicit program constructs for session-typed concurrency have been studied in [5]. While Scala concurrency libraries leverage implicits, it is unclear whether their usage is orthogonal to concurrency, or whether there are deeper connections between the two concepts that can be further studied in theory. This is a topic for further study.

References

[1] Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent Object Types. In *19th International Workshop on Foundations of Object-Oriented Languages.*

[2] Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. *ACM SIGPLAN Notices* 52, 1 (2017), 666–679.

[3] Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *Acm Sigplan Notices*, Vol. 49. ACM, 233–249.

[4] Philipp Haller. 2012. On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM, 1–6.

[5] Alex Jeffery and Martin Berger. 2018. Asynchronous Sessions with Implicit Functions and Messages. International Symposium on Theoretical Aspects of Software Engineering (TASE), 9–16. https://doi.org/10.1109/TASE.2018.00010

[6] Stefan Kaes. 1988. Parametric Overloading in Polymorphic Programming Languages. In *ESOP '88*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–144.

[7] Oleg Kiselyov. 2014. Implementing, and Understanding Type Classes. https://web.archive.org/web/20180910165920/http://okmij.org/ftp/Computation/typeclass.html.

[8] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proc. POPL*.

[9] Sam Lindley, Conor McBride, Phil Trinder, Don Sannella, Nada Amin, Karl Samuel GrÃijtter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. 9600 (03 2016).

[10] Martin Odersky et al. 2017. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/.

[11] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. POPL* (2018).

[12] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. *ACM Sigplan Notices* 45, 10 (2010), 341–360.

[13] Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi, and Philip Wadler. 2012. The Implicit Calculus: A New Foundation for Generic programming. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 35–44.

[14] Benjamin C Pierce and David N Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

[15] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Acm Sigplan Notices*, Vol. 51. ACM, 624–641.

[16] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76.

A Appendix

Full Soundness Proof

Lemma 1 (Preservation of subtyping under type translation)**.** If $\Gamma \vdash S <: U$ then $\Gamma^* \vdash_{DOT} S^* <: U^*$.

Proof. By induction on subtyping derivations.

- **Case** (Top): $\Gamma \vdash S <: \top$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: \top^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} S^* <: \top$
  - The goal follows from $(\text{Top})_{DOT}$.
- **Case** (Bot): $\Gamma \vdash \bot <: S$
  - To show: $\Gamma^* \vdash_{DOT} \bot^* <: S^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \bot <: S^*$
  - The goal follows from $(\text{Bot})_{DOT}$.
- **Case** (Refl): $\Gamma \vdash S <: S$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: S^*$
  - The goal is immediate from $(\text{Refl})_{DOT}$.
- **Case** (Trans): $\Gamma \vdash S <: R$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: R^*$
  - By inversion of (Trans):
    - $\Gamma \vdash S <: U$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
    - $\Gamma \vdash U <: R$
      - Then by induction: $\Gamma^* \vdash_{DOT} U^* <: R^*$
  - The goal then follows from $(\text{Trans})_{DOT}$.
- **Case** (<:-And): $\Gamma \vdash S <: U \wedge R$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: (U \wedge R)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} S^* <: U^* \wedge R^*$
  - By inversion of (<:-And):
    - $\Gamma \vdash S <: U$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
    - $\Gamma \vdash S <: R$
      - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: R^*$
  - The goal then holds by $(\text{<:-And})_{DOT}$.
- **Case** (<:-Sel): $\Gamma \vdash S <: u.A$
  - To show: $\Gamma^* \vdash_{DOT} S^* <: (u.A)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} S^* <: u.A^*$
  - By inversion of (<:-Sel): $\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u$
    - Then by theorem 1: $\Gamma^* \vdash_{DOT} u : \{A : S..U\}^*$
    - Then by definition 1: $\Gamma^* \vdash_{DOT} u : \{A : S^*..U^*\}$
  - The goal then follows from $(\text{<:-Sel})_{DOT}$.
- **Case** (Sel-<:): $\Gamma \vdash u.A <: U$
  - To show: $\Gamma^* \vdash_{DOT} (u.A)^* <: U^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} u.A^* <: U^*$
  - By inversion of (Sel-<:): $\Gamma \vdash u : \{A : S..U\} \rightsquigarrow u$
    - Then by theorem 1: $\Gamma^* \vdash_{DOT} u : \{A : S..U\}^*$
    - Then by definition 1: $\Gamma^* \vdash_{DOT} u : \{A : S^*..U^*\}$
  - The goal then follows from $(\text{Sel-<:})_{DOT}$.
- **Case** (All-<:-All-Ex): $\Gamma \vdash \forall(x : S_1)U_1 <: \forall(x : S_2)U_2$
  - To show: $\Gamma^* \vdash_{DOT} (\forall(x : S_1)U_1)^* <: (\forall(x : S_2)U_2)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \forall(x : S_1^*)U_1^* <: \forall(x : S_2^*)U_2^*$
  - By inversion of (All-<:-All-Ex):
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash_{DOT} S_2^* <: S_1^*$
    - $\Gamma, x : S_2 \vdash U_1 <: U_2$
      - Then by induction: $(\Gamma, x : S_2)^* \vdash_{DOT} U_1^* <: U_2^*$
      - And by definition 1: $\Gamma^*, x : S_2^* \vdash_{DOT} U_1^* <: U_2^*$
  - The goal then follows from $(\text{All-<:-All})_{DOT}$.
- **Case** (Typ-<:-Typ): $\Gamma \vdash \{A : S_1..U_1\} <: \{A : S_2..U_2\}$
  - To show: $\Gamma^* \vdash_{DOT} \{A : S_1..U_1\}^* <: \{A : S_2..U_2\}^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \{A : S_1^*..U_1^*\} <: \{A : S_2^*..U_2^*\}$
  - By inversion of (Typ-<:-Typ):
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash_{DOT} S_2^* <: S_1^*$
    - $\Gamma \vdash U_1 <: U_2$

  - Then by induction: $\Gamma^* \vdash_{DOT} U_1^* <: U_2^*$
- The goal then follows from $(\text{Typ-<:-Typ})_{DOT}$.

- **Case** $(\text{And}_1\text{-<:})$: $\Gamma \vdash S \wedge U <: S$
  - To show: $\Gamma^* \vdash_{DOT} (S \wedge U)^* <: S^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} S^* \wedge U^* <: S^*$
  - The goal follows from $(\text{And}_1\text{-<:})_{DOT}$.
- **Case** $(\text{And}_2\text{-<:})$: $\Gamma \vdash S \wedge U <: U$
  - To show: $\Gamma^* \vdash_{DOT} (S \wedge U)^* <: U^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} S^* \wedge U^* <: U^*$
  - The goal follows from $(\text{And}_2\text{-<:})_{DOT}$.
- **Case** (Fld-<:-Fld): $\Gamma \vdash \{a : S\} <: \{a : U\}$
  - To show: $\Gamma^* \vdash_{DOT} \{a : S\}^* <: \{a : U\}^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \{a : S^*\} <: \{a : U^*\}$
  - By inversion of (Fld-<:-Fld): $\Gamma \vdash S <: U$
    - Then by induction: $\Gamma^* \vdash_{DOT} S^* <: U^*$
  - The goal follows from $(\text{Fld-<:-Fld})_{DOT}$.
- **Case** (All-<:-All-Im): $\Gamma \vdash \forall\wr(x : S_1)U_1 <: \forall\wr(x : S_2)U_2$
  - To show: $\Gamma^* \vdash_{DOT} (\forall\wr(x : S_1)U_1)^* <: (\forall\wr(x : S_2)U_2)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \forall(x : S_1^*)U_1^* <: \forall(x : S_2^*)U_2^*$
  - By inversion of (All-<:-All-Im):
    - $\Gamma \vdash S_2 <: S_1$
      - Then by induction: $\Gamma^* \vdash S_2^* <: S_1^*$
    - $\Gamma, x : S_2 \vdash U_1 <: U_2$
      - Then by induction: $(\Gamma, x : S_2)^* \vdash U_1^* <: U_2^*$
      - And by definition 1: $\Gamma^*, x : S_2^* \vdash U_1^* <: U_2^*$
  - The goal then follows from $(\text{All-<:-All})_{DOT}$.

□

Lemma 2 (Preservation of free variables under type translation)**.** For all S, $\text{fv}(S) = \text{fv}(S^*)$.

Proof. Immediate from definition 1. □

Lemma 3 (Commutativity of type translation and name substitution)**.** For all S, u, v, $[u := v](S^*) = ([u := v]S)^*$.

Proof. By induction on S.

- **Case** $\top$
  - To show: $[u := v](\top^*) = ([u := v]\top)^*$
  - By definition 1 on LHS: $[u := v]\top = ([u := v]\top)^*$
  - By definition 2 on LHS and RHS: $\top = \top^*$
  - By definition 1 on RHS: $\top = \top$
- **Case** $\bot$
  - To show: $[u := v](\bot^*) = ([u := v]\bot)^*$
  - By definition 1 on LHS: $[u := v]\bot = ([u := v]\bot)^*$
  - By definition 2 on LHS and RHS: $\bot = \bot^*$
  - By definition 1 on RHS: $\bot = \bot$
- **Case** w
  - To show: $[u := v](w^*) = ([u := v]w)^*$
  - By definition 1 on LHS: $[u := v]w = ([u := v]w)^*$
  - **Case** $u = w$
    - By definition 2 on LHS and RHS: $v = v^*$
    - By definition 1 on RHS: $v = v$
  - **Case** $u \neq w$
    - By definition 2 on LHS and RHS: $w = w^*$
    - By definition 1 on RHS: $w = w$
- **Case** $\mu(u : S')$
  - To show: $[u := v](\mu(w : S')^*) = ([u := v]\mu(w : S'))^*$
  - By definition 1 on LHS: $[u := v]\mu(w : S'^*) = ([u := v]\mu(w : S'))^*$
  - **Case** $u = w$
    - By definition 2 on LHS and RHS: $\mu(w : S'^*) = \mu(w : S')^*$
    - By definition 1 on RHS: $\mu(w : S'^*) = \mu(w : S'^*)$
  - **Case** $u \neq w$
    - By definition 2 on LHS and RHS: $\mu(w : [u := v](S'^*)) = (\mu(w : [u := v]S'))^*$
    - By definition 1 on RHS: $\mu(w : [u := v](S'^*)) = \mu(w : ([u := v]S')^*)$
    - The goal then follows by induction.
- **Case** $\{A : S'..S''\}$
  - To show: $[u := v](\{A : S'..S''\}^*) = ([u := v]\{A : S'..S''\})^*$
  - By definition 1 on LHS: $[u := v](\{A : S'^*..S''^*\}) = ([u := v]\{A : S'..S''\})^*$
  - By definition 2 on LHS and RHS: $\{A : ([u := v](S'^*))..([u := v](S''^*))\} = \{A : [u := v]S'..[u := v]S''\}^*$
  - By definition 1 on RHS: $\{A : ([u := v](S'^*))..([u := v](S''^*))\} = \{A : ([u := v]S')^*..([u := v]S'')^*\}$
  - The goal then follows by induction.
- **Case** $\{a : S'\}$
  - To show: $[u := v](\{a : S'\}^*) = ([u := v]\{a : S'\})^*$
  - By definition 1 on LHS: $[u := v]\{a : S'^*\} = ([u := v]\{a : S'\})^*$
  - By definition 2 on LHS and RHS: $\{a : [u := v](S'^*)\} = (\{a : [u := v]S'\})^*$
  - By definition 1 on RHS: $\{a : [u := v](S'^*)\} = \{a : ([u := v]S')^*\}$
  - The goal then follows by induction.
- **Case** $u.A$
  - To show: $[u := v]((w.A)^*) = ([u := v](w.A))^*$
  - By definition 1 on LHS: $[u := v](w.A) = ([u := v](w.A))^*$
  - **Case** $u = w$
    - By definition 2 on LHS and RHS: $v.A = (v.A)^*$
    - By definition 1 on RHS: $v.A = v.A$
  - **Case** $u \neq w$
    - By definition 2 on LHS and RHS: $w.A = (w.A)^*$
    - By definition 1 on RHS: $w.A = w.A$
- **Case** $S' \wedge S''$
  - To show: $[u := v]((S' \wedge S'')^*) = ([u := v](S' \wedge S''))^*$
  - By definition 1 on LHS: $[u := v](S'^* \wedge S''^*) = ([u := v](S' \wedge S''))^*$
  - By definition 2 on LHS and RHS: $([u := v](S'^*)) \wedge ([u := v](S''^*)) = (([u := v]S') \wedge ([u := v]S''))^*$

  - By definition 1 on RHS: $([u := v](S'^*)) \wedge ([u := v](S''^*)) = ([u := v]S')^* \wedge ([u := v]S'')^*$
  - The goal then follows by induction.
- **Case** $\forall(w : S')S''$
  - To show: $[u := v]((\forall(w : S')S'')^*) = ([u := v](\forall(w : S')S''))^*$
  - By definition 1 on LHS: $[u := v](\forall(w : S'^*)S''^*) = ([u := v](\forall(w : S')S''))^*$
  - **Case** $u = w$
    - By definition 2 on LHS and RHS: $\forall(w : S'^*)[u := v](S''^*) = (\forall(w : S')[u := v]S'')^*$
    - By definition 1 on RHS: $\forall(w : S'^*)[u := v](S''^*) = \forall(w : S'^*)(([u := v]S'')^*)$
    - The goal then follows by induction.
  - **Case** $u \neq w$
    - By definition 2 on LHS and RHS: $\forall(w : [u := v](S'^*))[u := v](S''^*) = (\forall(w : [u := v]S')[u := v]S'')^*$
    - By definition 1 on RHS: $\forall(w : [u := v](S'^*))[u := v](S''^*) = \forall(w : ([u := v]S')^*)(([u := v]S'')^*)$
    - The goal then follows by induction.
- **Case** $\forall\wr(u : S')S''$
  - To show: $[u := v]((\forall\wr(u : S')S'')^*) = ([u := v](\forall\wr(u : S')S''))^*$
  - By definition 1 on LHS: $[u := v](\forall(w : S'^*)S''^*) = ([u := v](\forall\wr(w : S')S''))^*$
  - **Case** $u = w$
    - By definition 2 on LHS and RHS: $\forall(w : S'^*)[u := v](S''^*) = (\forall\wr(w : S')[u := v]S'')^*$
    - By definition 1 on RHS: $\forall(w : S'^*)[u := v](S''^*) = \forall(w : S'^*)(([u := v]S'')^*)$
    - The goal then follows by induction.
  - **Case** $u \neq w$
    - By definition 2 on LHS and RHS: $\forall(w : [u := v](S'^*))[u := v](S''^*) = (\forall\wr(w : [u := v]S')[u := v]S'')^*$
    - By definition 1 on RHS: $\forall(w : [u := v](S'^*))[u := v](S''^*) = \forall(w : ([u := v]S')^*)(([u := v]S'')^*)$
    - The goal then follows by induction.

□

Theorem 1 (Type-preserving translation of DIF terms)**.** If $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$ then $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$.

Proof. By induction on typing derivations.

- **Case** (Var-Ex): $\Gamma, p : S, \Gamma' \vdash p : S \rightsquigarrow p$
  - To show: $(\Gamma, p : S, \Gamma')^* \vdash_{DOT} p : S^*$
    - Or by definition 1, $\Gamma^*, p : S^*, \Gamma'^* \vdash_{DOT} p : S^*$
  - The goal follows immediately from $(\text{Var})_{DOT}$.
- **Case** (Var-Im): $\Gamma, i : S, \Gamma' \vdash \wr : S \rightsquigarrow i$
  - To show: $(\Gamma, i : S, \Gamma')^* \vdash_{DOT} i : S^*$
    - Or by definition 1, $\Gamma^*, i : S^*, \Gamma'^* \vdash_{DOT} i : S^*$
  - The goal follows immediately from $(\text{Var})_{DOT}$.
- **Case** (Sub): $\Gamma \vdash t : S' \rightsquigarrow \widehat{t}$
  - By inversion of (Sub):
    - $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
      - And by induction: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
    - $\Gamma \vdash S <: S'$
      - And by lemma 1: $\Gamma^* \vdash_{DOT} S^* <: S'^*$
  - The goal then follows from $(\text{Sub})_{DOT}$.
- **Case** (All-Ex-I): $\Gamma \vdash \lambda(x : S)t : \forall(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}$
  - To show: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : (\forall(u : S)U)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : \forall(u : S^*)U^*$
  - By inversion of (All-Ex-I):
    - $x \rightsquigarrow u$
      - Then either $x = u$ or u fresh
    - $\Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t}$
      - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t} : U^*$
      - And by definition 1: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t} : U^*$
    - $\{x, u\} \cap \text{fv}(S) = \emptyset$
      - Then by lemma 2, $\{x, u\} \cap \text{fv}(S^*) = \emptyset$, and then $u \notin \text{fv}(S^*)$
  - The goal then follows from $(\text{All-I})_{DOT}$.
- **Case** (All-Ex-E): $\Gamma \vdash x\ y : [u := y]U \rightsquigarrow \widehat{x}\ \widehat{y}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}\ \widehat{y} : ([u := y]U)^*$
    - Or by lemma 3: $\Gamma^* \vdash_{DOT} \widehat{x}\ \widehat{y} : [u := y]U^*$
  - By inversion of (All-Ex-E):
    - $\Gamma \vdash x : \forall(u : S)U \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : (\forall(u : S)U)^*$
      - Then by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : \forall(u : S^*)U^*$
    - $\Gamma \vdash y : S \rightsquigarrow \widehat{y}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{y} : S^*$
  - The goal then follows from $(\text{All-E})_{DOT}$.
- **Case** (Let): $\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ t' : U \rightsquigarrow \textbf{let}\ u = \widehat{t}\ \textbf{in}\ \widehat{t'}$
  - To show: $\Gamma^* \vdash_{DOT} \textbf{let}\ u = \widehat{t}\ \textbf{in}\ \widehat{t'} : U^*$
  - By inversion of (LET):
    - $x \rightsquigarrow u$
    - $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
    - $\Gamma, u : S \vdash t' : U \rightsquigarrow \widehat{t'}$
      - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t'} : U^*$
      - Then by definition 1: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t'} : U^*$
    - $x \notin \text{fv}(U)$
      - Then by lemma 2: $x \notin \text{fv}(U^*)$
  - The goal then follows from $(\text{Let})_{DOT}$.
- **Case** (And-I): $\Gamma \vdash x : S \wedge U \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : (S \wedge U)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : S^* \wedge U^*$
  - By inversion of (And-I):
    - $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$

  - $\Gamma \vdash x : U \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : U^*$
  - The goal then follows from (And-I)$_{DOT}$.
- **Case** ({}-I): $\Gamma \vdash \nu(x : S)d : \mu(u : S) \rightsquigarrow \nu(u : S^*)\widehat{d}$
  - To show: $\Gamma^* \vdash_{DOT} \nu(u : S^*)\widehat{d} : (\mu(u : S))^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \nu(u : S^*)\widehat{d} : \mu(u : S^*)$
  - By inversion of ({}-I):
    - $x \rightsquigarrow u$
    - $\Gamma, u : S \vdash d : S \rightsquigarrow \widehat{d}$
      - Then by theorem 2: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{d} : S^*$
      - And by definition 1: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{d} : S^*$
  - The goal then follows from ({}-I)$_{DOT}$.
- **Case** (Fld-E): $\Gamma \vdash x.a : S \rightsquigarrow \widehat{x}.a$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}.a : S^*$
  - By inversion of (Fld-E): $\Gamma \vdash x : \{a : S\} \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : \{a : S\}^*$
    - And by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : \{a : S^*\}$
  - The goal then follows from (Fld-E)$_{DOT}$.
- **Case** (Rec-I): $\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S^*)$
  - By inversion of (Rec-I): $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$
  - The goal then follows from (Rec-I)$_{DOT}$.
- **Case** (Rec-E): $\Gamma \vdash x : S \rightsquigarrow \widehat{x}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x} : S^*$
  - By inversion of (Rec-E): $\Gamma \vdash x : \mu(\widehat{x} : S) \rightsquigarrow \widehat{x}$
    - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S)^*$
    - And by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : \mu(\widehat{x} : S^*)$
  - The goal then follows from (Rec-E)$_{DOT}$.
- **Case** (All-Im-I): $\Gamma \vdash t : \forall \wr(u : S)U \rightsquigarrow \lambda(u : S^*)\widehat{t}$
  - To show: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : (\forall \wr(u : S)U)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \lambda(u : S^*)\widehat{t} : \forall \wr(u : S^*)U^*$
  - By inversion of (All-Im-I):
    - u fresh
    - $\Gamma, u : S \vdash t : U \rightsquigarrow \widehat{t}$
      - Then by induction: $(\Gamma, u : S)^* \vdash_{DOT} \widehat{t} : U^*$
      - And by definition 1: $\Gamma^*, u : S^* \vdash_{DOT} \widehat{t} : U^*$
    - $u \notin \text{fv}(S)$
      - Then by lemma 2: $u \notin \text{fv}(S^*)$
  - The goal then follows from (All-I)$_{DOT}$.
- **Case** (All-Im-E): $\Gamma \vdash x : [u := v]U \rightsquigarrow \widehat{x}\ v$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{x}\ v : ([u := v]U)^*$
    - Or by lemma 3: $\Gamma^* \vdash_{DOT} \widehat{x}\ v : [u := v]U^*$
  - By inversion of (All-Im-E):
    - $\Gamma \vdash x : \forall \wr(u : S)U \rightsquigarrow \widehat{x}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{x} : (\forall \wr(u : S)U)^*$
      - And by definition 1: $\Gamma^* \vdash_{DOT} \widehat{x} : \forall(u : S^*)U^*$
    - $\Gamma \vdash \wr : S \rightsquigarrow v$
      - Then by induction: $\Gamma^* \vdash_{DOT} v : S^*$
  - The goal then follows from (All-E)$_{DOT}$.

□

Theorem 2 (Type and domain-preserving translation of DIF definitions)**.** If $\Gamma \vdash d : S \rightsquigarrow \widehat{d}$ then $\Gamma^* \vdash_{DOT} \widehat{d} : S^*$ and $\mathbf{dom}(d) = \mathbf{dom}(\widehat{d})$.

Proof. By induction on typing derivations.

- **Case** (Typ-I): $\Gamma \vdash \{A = S\} : \{A : S..S\} \rightsquigarrow \{A = S^*\}$
  - To show for type preservation: $\Gamma^* \vdash_{DOT} \{A = S^*\} : (\{A : S..S\})^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \{A = S^*\} : \{A : S^*..S^*\}$
  - The goal for type preservation follows immediately from (Typ-I)$_{DOT}$.
  - Domain preservation is immediate from (Typ-I).
- **Case** (Fld-I): $\Gamma \vdash \{a = t\} : \{a : S\} \rightsquigarrow \{a = \widehat{t}\}$
  - To show for type preservation: $\Gamma^* \vdash_{DOT} \{a = \widehat{t}\} : \{a : S\}^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \{a = \widehat{t}\} : \{a : S^*\}$
  - By inversion of (Fld-I): $\Gamma \vdash t : S \rightsquigarrow \widehat{t}$
    - Then by theorem 1: $\Gamma^* \vdash_{DOT} \widehat{t} : S^*$
  - The goal for type preservation then follows from (Fld-I)$_{DOT}$.
  - Domain preservation is immediate from (Fld-I).
- **Case** (AndDef-I): $\Gamma \vdash d_1 \wedge d_2 : S_1 \wedge S_2 \rightsquigarrow \widehat{d_1} \wedge \widehat{d_2}$
  - To show: $\Gamma^* \vdash_{DOT} \widehat{d_1} \wedge \widehat{d_2} : (S_1 \wedge S_2)^*$
    - Or by definition 1: $\Gamma^* \vdash_{DOT} \widehat{d_1} \wedge \widehat{d_2} : S_1^* \wedge S_2^*$
  - By inversion of (AndDef-I):
    - $\Gamma \vdash d_1 : S_1 \rightsquigarrow \widehat{d_1}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{d_1} : S_1^*$ and $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$
    - $\Gamma \vdash d_2 : S_2 \rightsquigarrow \widehat{d_2}$
      - Then by induction: $\Gamma^* \vdash_{DOT} \widehat{d_2} : S_2^*$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$
    - $\mathbf{dom}(d_1) \cap \mathbf{dom}(d_2) = \emptyset$
      - Then $\mathbf{dom}(\widehat{d_1}) \cap \mathbf{dom}(\widehat{d_2}) = \emptyset$ follows since $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$
  - The goal for type preservation then follows from (AndDef-I).
  - Domain preservation holds by induction: if $\mathbf{dom}(d_1) = \mathbf{dom}(\widehat{d_1})$ and $\mathbf{dom}(d_2) = \mathbf{dom}(\widehat{d_2})$ then it follows that $\mathbf{dom}(d_1 \wedge d_2) = \mathbf{dom}(\widehat{d_1} \wedge \widehat{d_2})$.

□