ICFP: G: Type Inference for Disjoint Intersection Types

Birthe van den Berg, KU Leuven, Belgium[*]
birthe.vandenberg@kuleuven.be

1 Problem and Motivation

Intersection types [14, 34] were first introduced in the early 1980s in a theoretical pursuit to characterize all strongly normalizing λ-calculus terms. Yet, in recent years they have risen to practical prominence as part of industrial programming languages like Scala and its DOT-calculus [2], Microsoft's TypeScript, Facebook's Flow and Red Hat's Ceylon.

The intersection of types A and B, denoted $A\&B$, is the type of all values that have at the same time type A and type B. They are a natural fit for languages that feature both subtyping and advanced forms of inheritance, like multiple inheritance, traits or mixins. Consider the following Scala example where openAndShow invokes the open and show methods on its parameter x:

```
def openAndShow(x: Openable & Showable)
  = { x.open() ; x.show() }
```

The two invoked methods are defined in separate traits

```
trait Openable { def open(): Unit }
trait Showable { def show(): Unit }
```

Hence, x should have both types Openable and Showable. These two type requirements are combined in the intersection type Openable & Showable. The class Door inherits both traits and implements both methods:

```
class Door extends Openable with Showable {
  def open() { println("door opens") }
  def show() { println("door shows") }
}
```

This makes the call openAndShow(new Door) valid.

The merge operator [17] $x\,,,\,y$ provides a primitive notation for introducing intersection types. For instance, (true, , 5) has type Bool & Int and can behave as a Bool value or an Intvalue to suit the context it is used in:

```
(true ,, 5) OR false = true
(true ,, 5) + 3 = 8
```

If not treated carefully, the merge operator can merge terms of overlapping types, making the semantics ambiguous, for example:

```
(1 ,, 5) + 3 = 6
(1 ,, 5) + 3 = 8
```

The Scala and Typescript counterparts of the merge operator deal with this ambiguity by picking the first component of the intersection in case of overlap. This automatic choice can be somewhat arbitrary and unexpected. Hence, a more explicit approach prevents incoherence by rejecting programs with multiple meanings. This is the approach taken by *disjoint* intersection types [30], which we build on in this work. They require the types A and B of the two terms in a merge to be disjoint, denoted $A * B$. If the disjointness requirement is violated, as in the example above where Int $*$ Int does not hold, the program is rejected, explicitly asking the programmer to manually resolve the issue.

Disjoint intersection types are also compatible with powerful BCD-style subtyping [4] and parametric polymorphism [1]. This makes them able to express advanced subtyping features—like nested composition [15, 19], which is a key aspect of family polymorphism [18] and scalable extensibility—as well as a concise solution to the expression problem. Unfortunately using disjoint intersection types for these and other applications can be daunting as programmers have to add ample type and disjointness annotations to their programs in order to help the type checker.

This work aims to unburden the programmer and make disjoint intersection types more accessible by providing a type inference algorithm that automatically derives the necessary type and disjointness information. Our algorithm should do for disjoint intersection types what the famous Algorithm W does for the Hindley-Milner type system [21, 26, 27]. What makes our setting much more challenging is the combination of intersection types, disjointness constraints, BCD subtyping and parametric polymorphism. As far as we know, no prior work considers the combination of these features and can unlock type inference for modern programming languages with advanced inheritance features. In short, the problem statement of this paper is:

How to do type inference for disjoint intersection types?

This work is based on my master's thesis.

2 Background and Related Work

2.1 History of the F_i^+-calculus

Figure 1 shows an historical overview of how intersection types evolved into the F_i^+-calculus.

Coppo [14] and Pottinger [34] were the first to introduce intersection types to characterize all strongly normalizing λ-terms. Intersection types increase the expressiveness of types but in many languages, such as Ceylon [28], intersections of disjoint types are uninhabited. In order to also increase the expressiveness of terms, Reynolds [36] introduced

[*] Advisor: Tom Schrijvers

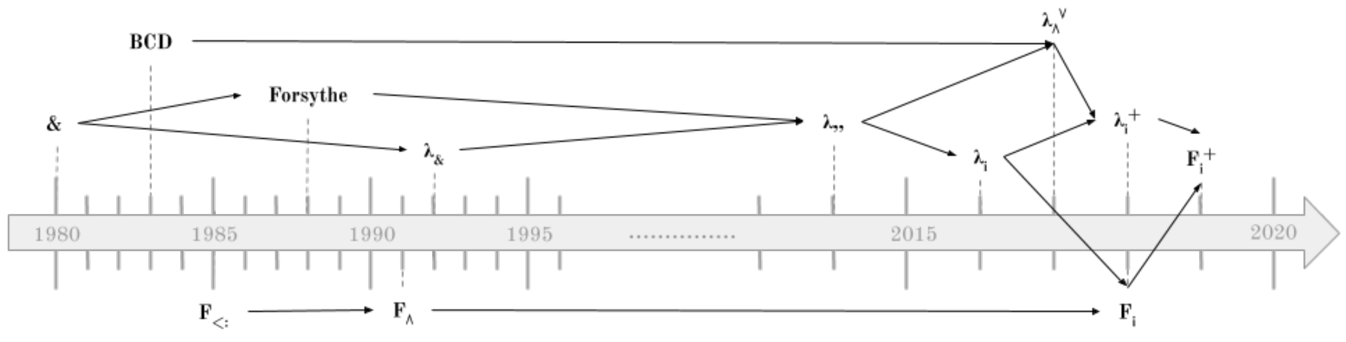

Figure 1. Timeline of the evolution of calculi with intersection types.

a restricted version of a merge operator for the Forsythe language, which for instance forbids intersections of two function types. Castagna et al. [11] also studied a merge-like operator for functions in their language $\lambda_\&$.

In 2014, Dunfield [17] introduced a calculus $\lambda_{,,}$ with unrestricted intersections, unions and merges. Oliveira et al. [30] transformed this calculus into a coherent equivalent λ_i by introducing disjoint intersection types. In 1983, Barendregt et al. introduced BCD subtyping [4] in their BCD type system with distributivity of the intersection operator over function types. Blaauwbroek [9] extended $\lambda_{,,}$ with BCD subtyping in his calculus $\lambda_\wedge^\vee$. In λ_i^+ [7] these ideas were combined into a coherent calculus with disjoint and unrestricted intersections featuring BCD subtyping.

When combining subtyping with polymorphism, the most common procedure is to use bounded quantification, e.g. $\forall(\alpha <: \text{Int}).\alpha \to \alpha$, of which $F_{<:}$ [10] in 1985 defines a model. Pierce [32] claimed that bounded quantification can be encoded in terms of intersection types and polymorphism and proposed the $F_\wedge$ calculus, in which the above type is equivalent to $\forall\alpha.(\alpha \wedge \text{Int}) \to (\alpha \wedge \text{Int})$.

Disjoint intersection types deserve special attention when used in combination with parametric polymorphism. This was apparent to Alpuim et al. [1] who extended λ_i with polymorphism to their calculus F_i, featuring disjoint polymorphism. Disjoint quantification includes disjointness requirements at the term and type level in the introduction of type variables, generalizing System F univeral quantification, e.g. $\forall(\alpha * \text{Int}).\alpha \,\&\, \text{Int}$. The empty constraint is denoted by disjointness with $\top$.

Combining the ideas of λ_i^+ and F_i led to the design of F_i^+ [8], a calculus with disjoint and unrestricted intersections, disjoint polymorphism and BCD subtyping; and SEDEL [6], a source language built on top of F_i^+.

2.2 Row polymorphism

Row types were first introduced by Wand [40] to model inheritance, presenting the concept of row variables to allow an incremental construction of record types. Consider polymorphic records: records that have many types, depending on which fields are shown or hidden. Therefore, whenever a value of a certain record type is expected, a record containing more information than necessary can be supplied [33]. Polymorphic records are supported by row polymorphism and by structural subtyping. The extra information of polymorphic records, which is implicitly included in the records, can take three forms: `absent`, `present` A, or row variable ρ, indicating that the field information is unknown. Row variables cannot contain fields with a label that is already present in the record. Similarly, in type $\{l : A\} \,\&\, B$ [1], type B can be considered a row variable. The disjointness constraint ensures that values of type B do not contain a record with type $\{l : A\}$.

Many features of row polymorphism can be simulated with disjoint polymorphism. Harper and Pierce have designed $\lambda^{||}$ [20], which is essentially System F with row polymorphism and a merge operator $||$. It uses constrained quantification to avoid concatenating records with a common field label and allows merging records with statically unknown fields, for example:

$$\text{mergeRcd} = \Lambda\alpha_1\#\text{Empty}.\Lambda\alpha_2\#\alpha_1.\lambda x_1 : \alpha_1.\lambda x_2 : \alpha_2.x_1||x_2$$

where the type variables lack type information, but the compatibility (i.e. no common record labels) is guaranteed by the constraints in the quantification. This can easily be translated to a calculus with disjoint quantification:

$$\text{mergeRcd'} = \Lambda(\alpha_1 * \top).\Lambda(\alpha_2 * \alpha_1).\lambda x_1 : \alpha_1.\lambda x_2 : \alpha_2.x_1,, x_2$$

The ideas of row polymorphism were later adopted into type systems for *extensible records* [20, 24]. Type inference is supported for various row type systems such as those of Wand [39, 40], Leijen [24] or Rémy [35].

2.3 Intersection Types and Type Inference

An intersection of types A and B can be understood from three perspectives [5]:

- *Set-theory*: containing all elements of A that are also elements of B,
- *Type-theory*: being a subtype of A as well as of B,
- *Order-theory*: greatest lower bound of A and B.

In this sense, intersection types can also be considered an option for type polymorphism. Every finite-rank [25] restriction of calculi with intersection types has principal typings and decidable type inference [23]. Rank-N intersection types, for a fixed value of N, refer to types in which an intersection does not appear in the domain of N or more function types [31]. For instance, ((Int&Bool) $\to$ Int) $\to$ Bool is a rank-3 intersection type. Furthermore, parametric polymorphism should be restricted to prenex predicative polymorphism in order to have decidable type inference. This means that type variables can only be instantiated with monomorphic types and polymorphic type constructors are only allowed at the top level and may not be nested in types, as in the Hindley-Milner type system. The latter only has to solve equality constraints but this paper combines disjointness constraints and subtyping constraints.

As Pierce presented encoding of subtyping constraints in bounded polymorphism with intersection types [32], Castagna and Xu [12] showed that we can encode a condition $\alpha <: \tau$ by replacing every occurrence of α with $\beta \sqcap \tau$ (β is a fresh variable). Dolan and Mycroft [16] use a similar recipe for their biunification algorithm for MLsub: they make bisubsitutions for constraints $\alpha <: \tau$ of the form $[\alpha^- \mapsto \alpha \sqcap \tau]$ and similarly for $\tau <: \alpha$ of the form $[\alpha^+ \mapsto \alpha \sqcup \tau]$. The symbols $\sqcap$ and $\sqcup$ represent greatest lower bounds and least upper bounds respectively. Pottier [33] describes an algorithm for System F extended with let-bindings and subtyping. He works with constrained types [29], including predicates π on types to express constraints [1]. For instance, a type scheme $\forall t.\pi(t) \Rightarrow f(t)$ is a type scheme representing the types $\{f(\tau) \mid \tau \text{ is a type such that } \pi(\tau) \text{ holds}\}$. Constrained types encode the same types as polar types with $\sqcup$ and $\sqcap$ operators. For instance, a constrained type $\forall \alpha.(\alpha <: \texttt{Int}) \Rightarrow \alpha \to \alpha$ uses a variation of bounded quantification and is thus equivalent to $\alpha \sqcap \texttt{Int} \to \alpha$ in MLsub.

But there is more! Jim [22] introduces a type system that features parametric polymorphism and recursion and poses a rank-2 restriction for intersection types. This type system has decidable type inference and is able to type some polymorphic, recursive terms. Angelo et al. [3] describe a type inference algorithm for a rank-2 fragment of gradual intersection types. Gradual typing is a type system that combines type checking at compile time (static typing) and at runtime (dynamic typing) whereby some expressions may statically be left untyped. However, as far as we know, none of the existing type inference algorithm deal with disjointness- and subtyping constraints.

3 Uniqueness of the Approach

3.1 Challenges

The uniqueness of my work lies in the combination of three elements: a **merge operator**, **disjointness constraints** and **subtyping constraints**, of which the most challenging are those of the form $A \,\&\, B <: C$.

The merge operator allows to explicitly introduce values of types with different type constructors, e.g., (Int $\to$ Int) & Bool. Most calculi lack support for this operator because it is not well-studied and complicates getting coherent semantics [17]. However, it enables a powerful type system with function overloading, e.g., $((\lambda x \to x + 1),, (\lambda x \to x > 0))\ 3$ which has as result 4,, true of type Int & Bool.

Also distributivity is seldomly used in calculi with a merge operator or languages with intersection types aimed at programming, since it adds considerable complexity to the system [7]. The merge operator in combination with BCD subtyping enables nice features such as partial application (e.g., $((\lambda x \to x + 1),, \text{true})\ 3$ with result 4) and parallel application (e.g., $((\lambda x \to x + 1),, (\lambda x \to x || \text{false}))\ (3,, \text{true})$ with result 4,, true).

Including disjointness requirements in the typing of terms allows detecting violations against them at compile time. Requiring disjoint intersection types restricts the expressive power of the merge operator but also enables interesting features [1, 6, 7, 30] such as multi-field extensible records, traits [37], mixins [13] and family polymorphism [18].

Solving subtyping constraints of the form $A \,\&\, B <: C$ is a necessary prerequisite to support first-class intersections, but is nontrivial since they cannot be solved without losing information or backtracking. Our approach tackles this constraint with three measures: (1) BCD-subtyping decomposes intersections into more simple types, (2) we add a lower/upper bound if a unification variable appears on the right/left of the constraint (3) explore both $A <: C$ and $B <: C$, and if both are solvable, then add the intersection of their common unification variables to the bounds. This way, we only need to backtrack with a constraint such as $u_1 \,\&\, u_2 <:$ Int, that is when we have no information yet about the types (unification variables) in the intersection.

3.2 High-level Approach

In order to tackle type inference for disjoint intersection types, we designed a novel calculus, PHiDI, that borrows ideas from F_i^+ and from Hindley-Milner. The challenge is to design a coherent language, together with a type inference algorithm that can infer principal types and is correct with respect to the specification of the underlying calculus.

The Hindley-Milner type system offers an approach to type inference by means of constraints and a unification algorithm for a prenex predicative calculus.

The features of PHiDI are inspired by the F_i^+-calculus, described by X. Bi et al. [8]. F_i^+ is an extension of the simply typed lambda calculus, supporting disjoint intersection types with a merge operator, BCD subtyping that enables nested composition, and disjoint parametric polymorphism. The F_i^+-calculus is currently presented by means of a bidirectional type system: it has both a checking mode and a synthesis mode. The programmer is required to explicitly annotate several terms with a type. Furthermore, it is necessary to make type abstraction and type application explicit, as well as to provide disjointness annotations in abstractions.

We designed a calculus with a type inference algorithm that elaborates to F_i^+, similar to Algorithm W, which elaborates the Hindley-Milner type system to System F:

$$\begin{tikzcd}
\text{Hindley-Milner} \arrow[r, Rightarrow, "\text{Algorithm W}"'] \arrow[d] & \text{System F} \arrow[d] \\
??? \arrow[r, Rightarrow, "???"] & F_i^+
\end{tikzcd}$$

4 Results and Contributions

This work presents Hindley-Milner style type inference for PHiDI. The main contributions of this paper are:

- **The PHiDI-calculus**: A type system with decidable type inference and let-polymorphism, disjoint intersection types with a merge operator, parametric disjoint polymorphism and BCD subtyping
- **Elaboration**: The semantics of the PHiDI type system are defined by elaboration to F_i^+.
- **Implementation**: PHiDI is implemented and all code presented in the paper is available. The implementation is available at https://github.com/birthevdb/Type_Inference_PHiDI. Moreover, the implementation features an extended version of the syntax presented in this paper, including booleans, `ifthenelse`-constructs and arithmetic expressions.

4.1 PHiDI: Polymorphic Hindley-Milner for Disjoint Intersection types

The PHiDI-calculus thus adopts features like BCD subtyping, disjoint intersection types and disjoint parametric polymorphism from the F_i^+-calculus. Therefore, some interesting terms can be typed with our algorithm, for example $(\{\ell = 3\},, \{\ell = \text{true}\}).\ell$ gets type Int & Bool and shows the BCD subtyping feature, or let $\hat{f} : \text{Int} \to \text{Int} = \lambda x \to x$ in $\hat{f}\ 5$ gets type Int. Other interesting terms are self-application $(\lambda x.x\ x)$, which is typed $\forall(\alpha, \beta).\alpha \,\&\, (\alpha \to \beta) \to \beta$, or $(\lambda x.x\ x)(\lambda x.x\ x)$, which gets type $\bot$ and can not be elaborated.

We can desugar annotated terms to let-expressions, e.g., $1 : \top$ desugars to let $\hat{x} : \top = 1$ in $\hat{x}$. In the same vein, let-expressions without a type-annotation desugar to lambda-abstractions, e.g., (let $x = E_1$ in E_2) desugars to $(\lambda x.E_2)\ E_1$. This way we only infer monomorphic types for let-expressions. Indeed, following the arguments of Vytiniotis et al. [38], we do not infer generalized types for local let-expressions; instead, we require explicit polymorphic type annotations. The only types that are generalized, are those of top-level expressions, in the line of Hindley-Milner type inference

PHiDI has a formalisation of its declarative subtyping relation, of which the most remarkable rules are those representing the BCD subtyping, for instance $(T_1 \to T_2) \,\&\, (T_1 \to T_3) <: T_1 \to T_2 \,\&\, T_3$. The subtyping relation is reflexive and transitive. Similarly, the disjointness relation under a type context is formalised. These rules are mostly straightforward and similar to those of F_i^+ [8].

The declarative typing consists of three judgments: (1) a generalization judgment to abstract over free variables in a monotype, together with their disjointness requirements, (2) a principality judgment that derives the principal monotype for a term, and (3) the specification judgment to derive monotypes for all expressions. The principality rule is used in a merge to disallow terms like $(\lambda x.x)\ 1,, (\lambda x.x)\ 2$, where we want to avoid one of the type being upcasted to $\top$ in order to satisfy the disjointness requirement. Similarly, we want to reject terms like $1,, 2$, where one of the types could be implicitly upcasted to $\top$. This is realized by omitting a general subsumption rule and instead distributing it over the typing rules for function application, field projection and let-expressions without causing ambiguity.

4.2 Elaboration

F_i^+ is somewhat to PHiDI as System F is to the Hindley-Milner system: F_i^+ features explicit term-level type abstraction (with disjoint quantification) and type application. We refer to the work of Bi et al. [8] for its definition, which we have extended with recursive lets. Our invariant is to generate F_i^+-terms that can be typed in synthesis mode. For this reason we add explicit type annotations to the F_i^+ term abstraction in lambda-abstractions and to the uses of subtyping in applications, projections and lets. The type elaboration is type preserving and indicated in this paper in *gray* .

4.3 Type Inference Algorithm

Polar types allow us to make the distinction between inputs and outputs of the algorithm. Indeed, positive types τ^+ denote outputs while negative types τ^- describe inputs. Following standard practice in subtyping, captured in the subtyping rule for functions types, the types of inputs can vary contravariantly, while those of outputs can vary covariantly. Indeed, when an input of type τ is required, any subtype of τ may be provided. Similarly, an output of type τ may be used at any supertype of τ.

To find a principal type, we simultaneously (1) assign unification variables to subterms, (2) traverse the algorithmic inference rules and (3) generate subtyping and disjointness constraints. In a second step we destruct the subtyping and disjointness constraints to assign lower/upper bounds to the unification variables, resulting in substitutions for these variables that can be applied to the final type and to the elaborated term.

The best way to show the working of the algorithm is by an example: derive the type and elaborated term for expression $((\lambda x.x),, 3)\ 5$. The derivation tree for this example is the following:

$$\dfrac{\dfrac{\dfrac{x : u_1 \vdash x : u_1 \rightsquigarrow x}{\bullet \vdash \lambda x.x : u_1 \to u_1 \rightsquigarrow \lambda x.x} \qquad \bullet \vdash 3 : \mathrm{Int} \rightsquigarrow 3 \qquad \mathbf{u_1 \to u_1 * Int}}{\bullet \vdash (\lambda x.x),,3 : (u_1 \to u_1)\,\&\,\mathrm{Int} \rightsquigarrow (\lambda x.x : u_1 \to u_1),,3} \qquad \bullet \vdash 5 : \mathrm{Int} \rightsquigarrow 5 \qquad (\mathbf{u_1} \to \mathbf{u_1})\,\&\,\mathbf{Int} <: \mathbf{Int} \to \mathbf{u_2}}{\bullet \vdash ((\lambda x.x),,3)\;5 : u_2 \rightsquigarrow (((\lambda x.x : u_1 \to u_1),,3) : \mathrm{Int} \to u_2)\;5}$$

The generated disjointness constraint $\mathbf{u_1} \to \mathbf{u_1} * \mathbf{Int}$ is trivially satisfied. There is only one subtyping constraint $(\mathbf{u_1} \to \mathbf{u_1})\,\&\,\mathbf{Int} <: \mathbf{Int} \to \mathbf{u_2}$, which is destructed by first simplifying the right type and then the left type. It therefore uses an (initially empty) queue containing types and/or record labels.

$$\bullet; (\mathbf{u_1} \to \mathbf{u_1})\,\&\,\mathbf{Int} <: \mathbf{Int} \to \mathbf{u_2}$$
$$(\mathbf{Int}); (\mathbf{u_1} \to \mathbf{u_1})\,\&\,\mathbf{Int} <: \mathbf{u_2}$$
$$(\mathbf{Int}); \mathbf{u_1} \to \mathbf{u_1} <: \mathbf{u_2} \vee \cancel{(\mathbf{Int}); \mathbf{Int} <: \mathbf{u_2}}$$
$$\bullet; \mathbf{u_1} <: \mathbf{u_2} \wedge \bullet; \mathbf{Int} <: \mathbf{u_1}$$

Whenever a constraint is encountered between a unification variable and a type, the latter is added to the lower/upper bounds in the table on the right.

	Lower bounds	**Upper bounds**	**Disjointness**
$\mathbf{u_1}$	Int	u_2	$\top$
$\mathbf{u_2}$	Int		$\top$

From this table, we derive a substitution:

$$[u_1^+ \mapsto \mathrm{Int}, u_1^- \mapsto \mathrm{Int}, u_2^+ \mapsto \mathrm{Int}]$$

Applying this substitution to the derived type u_2 gives Int, which has no more free variables to abstract over. Similarly, the elaborated term is $(((\lambda x.x : \mathrm{Int} \to \mathrm{Int}),,3) : \mathrm{Int} \to \mathrm{Int})\;5$

5 Conclusion

We have proposed a type inference algorithm for PHiDI, a coherent calculus for disjoint intersection types with a merge operator and parametric disjoint polymorphism with support for BCD-style subtyping. This type inference algorithm infers principal types, elaborates to F_i^+ and is implemented in Haskell. It is my hope that mainstream languages with intersection types will be inspired by and maybe adopt this principled approach to implement type inference.

References

[1] Alpuim, J.a., Oliveira, B., Shi, Z.: Disjoint polymorphism. pp. 1–28 (2017)
[2] Amin, N., Moors, A., Odersky, M.: Dependent object types. In: FOOL. No. CONF (2012)
[3] Angelo, P., Florido, M.: Type inference for rank 2 gradual intersection types (2019), in submission
[4] Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. J. Symb. Log. **48** (1983)
[5] Bi, X.: Disjoint intersection types : theory and practice. Ph.D. thesis (2018)
[6] Bi, X., d. S. Oliveira, B.C.: Typed first-class traits. In: ECOOP (2018)
[7] Bi, X., d. S. Oliveira, B.C., Schrijvers, T.: The essence of nested composition. In: ECOOP (2018)
[8] Bi, X., Xie, N., et al.: Distributive disjoint polymorphism for compositional programming. In: Caires, L. (ed.) Programming Languages and Systems. pp. 381–409. Springer International Publishing (2019)
[9] Blaauwbroek, L., Geuvers, H., Willemse, T.: On the interaction between unrestricted union and intersection types and computational effects. Ph.D. thesis, TU Eindhoven (2017)
[10] Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–523 (Dec 1985)
[11] Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. In: LFP. pp. 18–192 (1992)
[12] Castagna, G., Xu, Z.: Set-theoretic foundation of parametric polymorphism and subtyping. SIGPLAN Not. **46**(9), 94–106 (Sep 2011)
[13] Cook, W., Palsberg, J.: A denotational semantics of inheritance and its correctness. SIGPLAN Not. **24**(10), 433–443 (1989)
[14] Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Math. Log. Q. **27**, 45–58 (1981)
[15] Corradi, A., Servetto, M., Zucca, E.: Deepfjig: Modular composition of nested classes. In: PPPJ. pp. 101–110 (2011)
[16] Dolan, S., Mycroft, A.: Polymorphism, subtyping, and type inference in mlsub. In: POPL. pp. 60–72 (2017)
[17] Dunfield, J.: Elaborating intersection and union types. SIGPLAN Not. **47**(9), 17–28 (2012)
[18] Ernst, E.: Family polymorphism. In: ECOOP. pp. 303–326 (2001)
[19] Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL. pp. 270–282 (2006)
[20] Harper, R., Pierce, B.: A record calculus based on symmetric concatenation. In: POPL. pp. 131–142 (1991)
[21] Hindley, R.: The principal type-scheme of an object in combinatory logic. T. Am. Math. Soc. **146**, 29–60 (1969)
[22] Jim, T.: Rank 2 type systems and recursive definitions (2002)
[23] Kfoury, A.J., Wells, J.B.: Principality and decidable type inference for finite-rank intersection types. In: POPL. pp. 161–174 (1999)
[24] Leijen, D.: Extensible records with scoped labels. pp. 179–194 (2005)
[25] Leivant, D.: Polymorphic type inference. In: POPL. pp. 88–98 (1983)
[26] Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**, 348–375 (1978)
[27] Morris, J.H.: λ-calculus models of programming languages (1969)
[28] Muehlboeck, F., Tate, R.: Empowering union and intersection types with integrated subtyping. PACMPL **2**(OOPSLA) (2018)
[29] Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. TAPOS **5**, 35–55 (1999)
[30] Oliveira, B.C.d.S., Shi, Z., Alpuim, J.a.: Disjoint intersection types. In: ICFP. pp. 364–377 (2016)
[31] Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
[32] Pierce, B.C.: Programming with Intersection Types and Bounded Polymorphism. Ph.D. thesis, USA (1992)
[33] Pottier, F.: Type inference in the presence of subtyping: from theory to practice (1998)
[34] Pottinger, G.: A type assignment for the strongly normalizable λ-terms (1980)
[35] Rémy, D.: Type Inference for Records in Natural Extension of ML, pp. 67–95. MIT Press (1994)
[36] Reynolds, J.C.: Preliminary design of the programming language forsythe (1988)
[37] Shärli, N., Ducasse, S., et al.: Traits: Composable units of behavior. Tech. rep. (2002)
[38] Vytiniotis, D., Peyton Jones, S., Schrijvers, T.: Let should not be generalized. In: TLDI 2010. pp. 39–50 (2010)
[39] Wand, M.: Complete type inference for simple objects. In: LICS (1987)
[40] Wand, M.: Type inference for record concatenation and multiple inheritance. Inf. Comput. **93**(1), 1–15 (1991)