Type Inference for Rank-2 Intersection Types using Set Unification

Pedro Ângelo[1,2] and Mário Florido[1,3]

[1] LIACC, Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto, rua do Campo Alegre s/n, 4169 - 007
[2] `pedro.angelo@fc.up.pt`
[3] `amflorid@fc.up.pt`

Abstract. Several type inference approaches for rank-2 idempotent and commutative intersection types have been presented in the literature. Type inference relies on two stages: type constraint generation and solving. Defining constraint generation rules is rather straightforward, with one exception. To infer the type of an application, several derivations of the argument are required, one for each instance of the domain type of the function. The types of these derivations are then constrained against the instances. Noting that these derivations are isomorphic, by renaming of type variables, they can be obtained via a duplication operation on a single derivation of the argument. The application rule then constrains the intersection type resulting from duplication against the domain type of the function, resulting in an equality constraint between intersections. By treating intersections as sets, these constraints can be solved by solving a set unification problem, thus ensuring the types of the argument unify with the domain type of the function. Here we present a new type inference algorithm for rank-2 intersection types, which relies on set unification to solve equality constraints between intersections, and show it is both sound and complete.

Keywords: Intersection types · Type inference · Set unification.

1 Introduction

The benefits (and the costs) of strong static typing in programming languages are now generally recognized. Languages such as ML, Haskell or Java are examples of the use of strong typing. To avoid the extra effort of declaring types for every part of the program, compilers should infer types as much as possible. And to avoid rejecting well-behaved programs as much as possible, type inference should be able to support some form of polymorphism. Two of the main options for polymorphism are universally quantified types (such as the Damas-Milner type system [17] and System $\mathcal{F}$ [29,28,41]), and intersection types [13].

Intersection types originate in the works of Barendregt, Coppo and Dezani [15,13,6], and give us a characterization of the strongly normalizable terms. New attention was given to intersection type systems due to a result of Kfoury and Wells [35,36] which proved that these systems are decidable for restrictions of finite rank, which correspond to a large class of typable terms. Consider the following example: in intersection type systems $\lambda x \, . \, x \; x$ has type $(\alpha \wedge (\alpha \to \beta)) \to \beta$. Note that the two (non-unifiable) types of the variable x belong to the domain type of the abstraction linked by the intersection operator. A more interesting example is the term $(\lambda x \, . \, x \; x) \; I$, where $I = \lambda y \, . \, y$. This term has type $\alpha \to \alpha$ which does not involve intersections, although it is not typable in the simply typed lambda calculus [16,31] because it has a non-typable subterm.

Intersection type systems characterise the set of strongly normalising terms and have huge expressive power, typing more terms than the simply typed lambda calculus or the type system of pure ML or core Haskell. Applications of intersection types in programming language theory cover diverse topics, including the design of programming languages [42,7], program analysis [38], program synthesis [26], and extensions such as refinement, union and gradual types [27,24,23,3,11,12]. But expressive power comes with a price: type theoretic problems such as type inference and inhabitation are undecidable in general [45,5].

In [35] Wells and Kfoury define an intersection type system which types exactly the strongly normalizable terms and shows that every finite-rank restriction of this system, using Leivant's notion of rank [37], has principal typings and also decidable type inference. This system uses expansion variables [36,10], which are subject to substitution as are ordinary variables, and a unification-based type inference algorithm using a new form of unification called β-unification. Due to the complexity of type inference algorithms for higher finite ranks, the most successful decidable fragments of intersection type systems have focused on the rank-2 restriction. Indeed, we rely on the same argument to motivate our rank-2 restriction of the type inference algorithm presented in this paper. The rank of a type is easily determined by its syntactic tree. A type is of rank n if no path from the root of the syntactic tree of the type to an intersection passes to the left of n arrows. Rank 0 and 1 are equivalent to the simple typed lambda-calculus. But starting from rank 2, the systems type more terms than the type system of pure ML or Core Haskell.

Van Bakel presents a unification algorithm as the basis of type inference for a rank-2 system [46]. Later, independent work by Trevor Jim also solves the same problem for practical programming language issues such as recursive definitions and separate compilation [33]. Damiani [18] also studied rank-2 principal typings with intersection types and focus his work on rank-2 typable recursive definitions.

All these previous algorithms rely on extensions to first-order unification [43], either explicit [46,35,36], or implicit [32,33]. In these, a more general form of subtyping type constraints is first generated, and subsequently, in the constraint solving phase, further simplified by rewriting subtyping into equalities.

Several authors have also explored intersection type inference systems using unification theory. Approaches focus on relating β-reduction with unification [8], and more similarly to our work, building type inference algorithms using set unification theory [22,14,44]. However, we relate the properties of intersection types with set unification, which as far as we know, is novel work.

Originally [15,13], intersection types were denoted by finite sets of types:

"The main idea is to define from an arbitrary set of types $\{\tau_1, \cdots, \tau_n\}$ *a "sequence"* $[\tau_1, \cdots, \tau_n]$ *whose underlying set of terms can be interpreted as the intersection of those of* $\tau_1, \cdots, \tau_n$*." [13]*

Picking up on this original motivation, we here define a new type inference algorithm for rank-2 intersection types which relies on *set unification* [20,21] to solve the type constraints generated by function applications. The main contributions of this paper are the following:

- A unification-based type inference algorithm for rank-2 intersection types using set unification. The algorithm is terminating, always returning principal typings. A nice feature of this algorithm is its similarity with type inference for simple types [39] - just replace first-order unification by set unification.
- Proofs of soundness and completeness of the algorithm, meaning that the outputs of type inference are types which are derivable in a rank-2 intersection type system, and more, they are principal typings in the sense that every other type derivable in the type system may be obtained from them using substitution.

It is important to note that the majority of the discussed results can be obtained by the other previously defined rank-2 intersection type inference algorithms. Nonetheless, it is our belief that the work in this paper constitutes further a step towards a better understanding of the role of set unification as the base of type inference algorithms for intersection types and may highlight how intersections at different depth are related to different restrictions of set unification in the type inference mechanism. Complete proofs for theorems introduced in this paper are presented in the technical report [4].

The paper is organized as follows. Section 2 introduces the syntax of the system. A rank-2 intersection type system, where every type declared in the context is used in the type derivation, is presented in section 3. The formalization of the type inference algorithm, along with its components, follows in section 4. The first phase of the algorithm consists of the constraint generation rules, which are detailed in subsection 4.1. In particular, we present an alternative design to the rule for applications: by requiring only one derivation and then duplicating it, there is a derivation of the argument for each instance in the domain type of the function. Set unification, explained in subsection 4.2, will be required to solve equality constraints between intersection types. The general constraint solving rules are presented in subsection 4.3. We finally produce the type inference algorithm, along with important properties, in subsection 4.4. The conclusion follows in section 5.

2 Types and Terms

Our language is an intersection typed lambda calculus à la Curry, which supports term constants, such as integers and booleans, and built-in addition. Other arithmetic operations can be defined similarly. The syntax of our language is given by the following grammar:

Definition 1 (Syntax).

monotypes	τ, ρ	$::=$	$B \mid \alpha \mid \sigma \to \tau$
sequences	σ, υ	$::=$	$\tau_1 \wedge \ldots \wedge \tau_n \mid \phi$
terms	M, N	$::=$	$k \mid x \mid \lambda x\ .\ M \mid M\ M \mid M + M$
typing context	Γ	$::=$	$\emptyset \mid \Gamma, x : \sigma \qquad$ *with* $\sigma \in \mathcal{T}_1$
constraint	C	$::=$	$\sigma \doteq \sigma$
constraints	C^*	$::=$	$\emptyset \mid C^* \cup C$

B ranges over base types such as Int and Bool, α and β range over type variables and ϕ ranges over sequence variables. τ and ρ range over monotypes i.e. the top level constructor is not the intersection type connective, and σ and υ range over sequences. M and N range over terms, x, y and z range over term variables and k ranges over constants, such as integers and booleans. Γ ranges over typing contexts, and $\emptyset$ represents an empty context. C ranges over equality constraints, written as $\sigma \doteq \sigma$, and C^* ranges over multisets of equality constraints C. χ ranges over sets of type, and sequence, variables.

Remark 1. The indexes i, j, m, n, p and q range over the set $\mathbb{Z}_0^+$.

Remark 2. We distinguish meta-variables with different subscript natural numbers, and also with superscript apostrophe.

As in the original system [13,40], we consider the intersection connective $\wedge$ as commutative, e.g. $\tau \wedge \rho = \rho \wedge \tau$, and idempotent, e.g. $\tau \wedge \tau = \tau$. We do not consider associativity, since we are not dealing with a binary operator. Therefore, an intersection type $\tau_1 \wedge \ldots \wedge \tau_n$ is seen as the set of types $\tau_1, \ldots, \tau_n$. Given a sequence $\tau_1 \wedge \ldots \wedge \tau_n$, each τ_i is called an *instance* of the intersection. We allow sequences of size one, so σ and υ also range over monotypes. Sequences can only appear in the left-hand side of the arrow type constructor, therefore the shape of a (valid) arrow type is $\sigma \to \tau$. The intersection type connective $\wedge$ has a higher precedence than the arrow type constructor $\to$, and $\to$ associates to the right.

Definition 2 (Type Variables). *The function $tvars(.)$, which returns the set of type, and sequence, variables occurring in a given type, is defined as follows: $tvars(\sigma) \overset{def}{=} \{\alpha \mid \alpha$ occurs in $\sigma\} \cup \{\phi \mid \phi$ occurs in $\sigma\}$.*

Definition 3 (Free Variables). *The function $fvars(.)$, which returns the set of free term variables occurring in a given term, is defined as follows: $fvars(M) \overset{def}{=} \{x \mid x \text{ occurs free in } M\}$.*

Definition 4 (Atomic Type Sets). *Atomic types in our language are categorized according to the following sets:*

$$\begin{array}{lcl} \mathcal{T}_{base} & = & \{B \mid B \textit{ is a base type}\} \\ \mathcal{T}_{tvar} & = & \{\alpha \mid \alpha \textit{ is a type variable}\} \\ \mathcal{T}_{svar} & = & \{\phi \mid \phi \textit{ is a sequence variable}\} \end{array}$$

According to the definition of rank restriction [37,33], a *rank n intersection type* can have no intersection type connective $\wedge$ to the left of n or more arrow type constructors $\rightarrow$:

Definition 5 (Rank). *Types of our language are categorized according to rank:*

$$\begin{array}{llcl} \textit{simple types} & \mathcal{T}_0 & = & \mathcal{T}_{base} \cup \mathcal{T}_{tvar} \cup \{\tau \rightarrow \rho \mid \tau,\ \rho \in \mathcal{T}_0\} \\ \textit{rank 1 types} & \mathcal{T}_1 & = & \mathcal{T}_0 \cup \mathcal{T}_{svar} \cup \{\tau_1 \wedge \ldots \wedge \tau_n \mid \tau_1,\ \ldots,\ \tau_n \in \mathcal{T}_0\} \\ \textit{rank 2 types} & \mathcal{T}_2 & = & \mathcal{T}_0 \cup \{\sigma \rightarrow \tau \mid \sigma \in \mathcal{T}_1,\ \tau \in \mathcal{T}_2\} \end{array}$$

We restrict types in our system to be only of up to rank 2, so the only possible types are those belonging to $\mathcal{T}_1 \cup \mathcal{T}_2$, e.g. $(((\tau \rightarrow \rho) \wedge \tau) \rightarrow \rho) \rightarrow \tau$ is not a valid type.

Remark 3. We denote the singleton context, which contains only one type binding, as $x : \sigma$. We write Γ_1, Γ_2 for the union of contexts Γ_1 and Γ_2, assuming Γ_1 and Γ_2 are disjoint.

Definition 6 (Joining Typing Contexts). *Let Γ_1 and Γ_2 be two typing contexts. $\Gamma_1 \wedge \Gamma_2$ is a typing context, where $x : \sigma \in \Gamma_1 \wedge \Gamma_2$ if and only if σ is defined as follows:*

$$\sigma = \begin{cases} \sigma_1 \wedge \sigma_2, & \textit{if } x : \sigma_1 \in \Gamma_1 \textit{ and } x : \sigma_2 \in \Gamma_2 \\ \sigma_1, & \textit{if } x : \sigma_1 \in \Gamma_1 \textit{ and } \neg\exists \sigma_2 \ .\ x : \sigma_2 \in \Gamma_2 \\ \sigma_2, & \textit{if } \neg\exists \sigma_1 \ .\ x : \sigma_1 \in \Gamma_1 \textit{ and } x : \sigma_2 \in \Gamma_2 \end{cases}$$

3 Type System

In figure 1 we define an intersection type system where every type declared in the context is used in the type derivation, a property which is going to be quite useful in subsequent results.

The two rules for abstractions, [T-AbsI] and [T-AbsK], are necessary because if there is a derivation of $\Gamma \vdash_\wedge M : \sigma$ and x does not occur free in M, then there is not a type declaration for x in Γ. The set of types for a given term M in this system is strictly included in the set of types for M in the original

$$[\text{T-C}\textsc{on}]\ \frac{\text{k is a constant of base type B}}{\emptyset \vdash_\wedge k : B} \qquad [\text{T-V}\textsc{ar}]\ \frac{}{x : \tau \vdash_\wedge x : \tau}$$

$$[\text{T-A}\textsc{bs}\text{I}]\ \frac{\Gamma, x : \sigma \vdash_\wedge M : \tau}{\Gamma \vdash_\wedge \lambda x\ .\ M : \sigma \to \tau} \qquad [\text{T-A}\textsc{bs}\text{K}]\ \frac{\Gamma \vdash_\wedge M : \tau}{\Gamma \vdash_\wedge \lambda x\ .\ M : \sigma \to \tau}\ x \notin fvars(M)$$

$$[\text{T-A}\textsc{pp}]\ \frac{\begin{array}{c}\Gamma \vdash_\wedge M : \tau_1 \wedge \ldots \wedge \tau_n \to \tau \\ \forall i \in 1..n\ .\ \Gamma_i \vdash_\wedge N : \tau_i\end{array}}{\Gamma \wedge \Gamma_1 \wedge \ldots \wedge \Gamma_n \vdash_\wedge M\ N : \tau}$$

$$[\text{T-A}\textsc{dd}]\ \frac{\Gamma_1 \vdash_\wedge M : Int \qquad \Gamma_2 \vdash_\wedge N : Int}{\Gamma_1 \wedge \Gamma_2 \vdash_\wedge M + N : Int}$$

Fig. 1. Intersection Type System ($\Gamma \vdash_\wedge M : \sigma$)

intersection type system of Coppo and Dezani [15,13]. For example, the type $(\alpha_1 \wedge \alpha_2) \to \alpha_1$ types $\lambda x\ .\ x$ in the Coppo-Dezani type system but not in our system. The reason for this is that types for free variables, which are introduced by rule [T-Var], can only be included in an intersection via rules [T-App] or [T-Add]. Thus each element of the intersection corresponds to a type that is actually used in the type derivation. However, the set of terms typable in both systems is the same and corresponds to the strongly normalizable terms (a proof of this for a similar type system can be found in [25]).

One peculiarity of this type system is that it does not satisfy the property of *subject reduction* as it is shown by the following example:

Example 1. In this system, the following two statements hold:

$$z : \alpha_2 \to \beta \vdash_\wedge \lambda x\ .\ (\lambda y\ .\ z)\ x\ x : \alpha_1 \wedge \alpha_2 \to \beta$$

$$\lambda x\ .\ (\lambda y\ .\ z)\ x\ x \underset{\beta}{\to} \lambda x\ .\ z\ x$$

But we also have that

$$z : \alpha_2 \to \beta \not\vdash_\wedge \lambda x\ .\ z\ x : \alpha_1 \wedge \alpha_2 \to \beta$$

because type $\alpha_1 \wedge \alpha_2$ can't be assigned to x, since only one occurrence of x (typed with α_2) exists.

The lack of subject reduction also happens in other restrictions of intersection type systems where every type in the environment has to be used in the type derivation [19,35]. The reason for the lack of subject reduction is that there is no weakening introducing unneeded type assumptions. Note that the lack of subject reduction is not a problem, because derivations in this system can be easily translated into derivations on more standard systems of intersection types which have subject reduction. Defining the system without a weakening mechanism makes the later analysis about type inference much easier.

Consider the following example of a type derivation for $(\lambda x\ .\ x\ x)\ (\lambda y\ .\ y)$:

Example 2. We abbreviate $\tau = \rho \to \rho$. We have the following derivations by applying the last rule as follows:

$$\text{[T-AbsI]} \quad \emptyset \vdash_\wedge \lambda x\ .\ x\ x : (\tau \to \tau) \wedge \tau \to \tau \tag{1}$$
$$\text{[T-AbsI]} \quad \emptyset \vdash_\wedge \lambda y\ .\ y : \tau \to \tau \tag{2}$$
$$\text{[T-AbsI]} \quad \emptyset \vdash_\wedge \lambda y\ .\ y : \tau \tag{3}$$

By rule [T-App] on derivations (1), (2) and (3), we have:

$$\text{[T-App]} \quad \emptyset \vdash_\wedge (\lambda x\ .\ x\ x)\ (\lambda y\ .\ y) : \tau$$

4 Type Inference

We follow a conventional approach to type inference [47]: a constraint generation phase generates type constraints from the term, and a constraint solving phase solves these constraints to generate type substitutions.

Substitution on types is defined in the standard way [39], extended to allow intersections.

Definition 7 (Substitution). *Let S range over standard type substitutions [39]. We write $[\alpha \mapsto \tau]$ for a type substitution on monotypes that maps a type variable α into a monotype τ; and $[\phi \mapsto \sigma]$ for a type substitution on sequences that maps a sequence variable ϕ into a sequence σ.*

For each type system rule in figure 1, an analogous constraint generation rule is required. Deriving these from the type system is rather straightforward: convert judgments in the premises to constraint generation judgments, making the type opaque; then convert the judgment in the conclusion, adding constraints that reflect how types relate to each other in the type system.

Deriving a constraint generation rule from [T-App] is not as straightforward. In the type system rule for applications, the function is assumed to be typed with an arrow type. However, the same assumption cannot be made for the constraint generation rules. Therefore, two constraint generation rules for applications are required: one where this assumption holds, and another where it does not, leading to an opaque type being inferred for the function. In standard systems [32], the application rule which assumes the type of the function is an arrow type behaves similarly to rule [T-App]. The rule ensures there are distinct type derivations of the argument, exactly one for each instance of the domain type of the function. By having distinct type derivations, the rule ensures the argument fits into each occurrence of the bound variable in the body of the lambda abstraction.

We follow a different approach: the application rule features a single type derivation of the argument. Then, the type obtained from this derivation is duplicated, and each copy is constrained to each instance in the domain type of the function. The duplication operation is defined as in [46]:

Definition 8 (Duplication). *Let $\chi = \{\alpha_1, \ldots, \alpha_j\} \cup \{\phi_{j+1}, \ldots, \phi_m\}$ be a set of type and sequence variables; let $\beta_{11}, \ldots, \beta_{1n}, \ldots, \beta_{m1}, \ldots, \beta_{mn}$ be fresh type variables; and let $S_i = [\alpha_1 \mapsto \beta_{1i}, \ldots, \alpha_j \mapsto \beta_{ji}, \phi_{j+1} \mapsto \beta_{(j+1)i}, \ldots, \phi_m \mapsto \beta_{mi}]$, for $1 \leq i \leq n$. The duplication function $duplicate^n_\chi(\tau)$ is defined as follows: $duplicate^n_\chi(\tau) \overset{def}{=} S_1(\tau) \wedge \ldots \wedge S_n(\tau)$.*

The argument χ represents the set of variables that will be duplicated, and the argument n represents the number of duplications. Therefore, n fresh variables β are required for each type variable in χ, to ensure new duplications. Only simple types ($\tau \in \mathcal{T}_0$) are duplicated, so sequence variables ϕ that might appear in the type are treated as simple types and replaced by type variables β. Note that if duplication is applied to a type without type variables, due to idempotence, duplication will return the same type, e.g. $duplicate^2_\chi(Int \rightarrow Int) = (Int \rightarrow Int) \wedge (Int \rightarrow Int)$, which is the same as $Int \rightarrow Int$. On the other hand, if type variables are considered, duplication will generate n many copies of the type, e.g. $duplicate^2_{\{\alpha_1,\alpha_2\}}(\alpha_1 \rightarrow \alpha_2) = (\beta_{11} \rightarrow \beta_{21}) \wedge (\beta_{12} \rightarrow \beta_{22})$.

We give meaning to constraints through a satisfaction relation $\models$. A substitution S satisfies a constraint $\sigma \doteq \upsilon$ if and only if applying the substitution to both types in the constraint yields an equality. Taking into account that intersection types are idempotent and commutative, two sequences are equal if both share the same set of instances. Since sequences of size one are allowed, the equality constraint between monotypes $\tau \doteq \rho$ is an instance of $\sigma \doteq \upsilon$, i.e. $S \models \tau \doteq \rho \iff S(\tau) = S(\rho)$.

Definition 9 (Constraint Satisfaction).

1. $S \models \emptyset$
2. $S \models \sigma \doteq \upsilon \iff S(\sigma) = S(\upsilon)$
3. $S \models C^* \iff S \models C$ *for all* $C \in C^*$

Definition 10 (Lifting Type Variables). *We lift function $tvars(.)$, from definition 2, to typing contexts Γ and equality constraints C^* in the obvious way.*

Definition 11 (Lifting Substitution). *We lift substitutions, from definition 7, to:*

- *typing contexts Γ in the obvious way;*
- *constraints in the following way: $S(\sigma \doteq \upsilon) \overset{def}{=} S(\sigma) \doteq S(\upsilon)$. Also, $S(C^* \cup C) \overset{def}{=} S(C^*) \cup S(C)$ and $S(\emptyset) \overset{def}{=} \emptyset$.*

Definition 12 (Lifting Duplication). *Assuming $S_1, \ldots, S_n$ are type substitutions generated from χ according to definition 8, we lift function $duplicate^n_\chi(.)$, from definition 8, to:*

- *typing contexts in the following way: $duplicate^n_\chi(\Gamma) \overset{def}{=} S_1(\Gamma) \wedge \ldots \wedge S_n(\Gamma)$;*
- *constraints in the following way: $duplicate^n_\chi(C^*) \overset{def}{=} S_1(C^*) \cup \ldots \cup S_n(C^*)$.*

$$[\text{G-C}\textsc{on}]\ \frac{\text{k is a constant of base type B}}{\emptyset \vdash_\wedge k : B \mid \emptyset} \qquad [\text{G-V}\textsc{ar}]\ \frac{\alpha \text{ fresh}}{x : \alpha \vdash_\wedge x : \alpha \mid \emptyset}$$

$$[\text{G-A}\textsc{bs}\text{I}]\ \frac{\Gamma, x : \sigma \vdash_\wedge M : \tau \mid C^*}{\Gamma \vdash_\wedge \lambda x\ .\ M : \sigma \to \tau \mid C^*}$$

$$[\text{G-A}\textsc{bs}\text{K}]\ \frac{\Gamma \vdash_\wedge M : \tau \mid C^* \qquad \phi \text{ fresh}}{\Gamma \vdash_\wedge \lambda x\ .\ M : \phi \to \tau \mid C^*}\ x \notin fvars(M)$$

$$[\text{G-A}\textsc{pp}\wedge]\ \frac{\begin{array}{c}\Gamma_1 \vdash_\wedge M : \tau_1 \wedge \ldots \wedge \tau_n \to \tau \mid C_1^* \qquad \Gamma_2 \vdash_\wedge N : \rho \mid C_2^* \\ \mathit{duplicate}^n(\langle \Gamma_2, \rho, C_2^* \rangle) = \langle [\Gamma_{21}, \ldots, \Gamma_{2n}], [\rho_1, \ldots, \rho_n], [C_{21}^*, \ldots, C_{2n}^*] \rangle \\ C = \tau_1 \wedge \ldots \wedge \tau_n \doteq \rho_1 \wedge \ldots \wedge \rho_n \end{array}}{\Gamma_1 \wedge \Gamma_{21} \wedge \ldots \wedge \Gamma_{2n} \vdash_\wedge M\ N : \tau \mid C_1^* \cup C_{21}^* \cup \ldots \cup C_{2n}^* \cup C}$$

$$[\text{G-A}\textsc{pp}]\ \frac{\Gamma_1 \vdash_\wedge M : \tau \mid C_1^* \qquad \Gamma_2 \vdash_\wedge N : \rho \mid C_2^* \qquad \alpha \text{ fresh}}{\Gamma_1 \wedge \Gamma_2 \vdash_\wedge M\ N : \alpha \mid C_1^* \cup C_2^* \cup \{\tau \doteq \rho \to \alpha\}}$$

$$[\text{G-A}\textsc{dd}]\ \frac{\Gamma_1 \vdash_\wedge M : \tau \mid C_1^* \qquad \Gamma_2 \vdash_\wedge N : \rho \mid C_2^*}{\Gamma_1 \wedge \Gamma_2 \vdash_\wedge M + N : \mathit{Int} \mid C_1^* \cup \{\tau \doteq \mathit{Int}\} \cup C_2^* \cup \{\rho \doteq \mathit{Int}\}}$$

Fig. 2. Constraint Generation ($\Gamma \vdash_\wedge M : \tau \mid C^*$)

Besides duplicating the type of argument derivations in the application rule, the typing context and constraints must also be duplicated, thus simulating several derivations of the same term. These derivations are just renamings of type variables of the original derivation.

Definition 13 (Duplication). *Let* $\langle \Gamma, \tau, C^* \rangle$ *be a triple composed of a typing context* Γ*, a type* τ *and constraints* C^**. The* duplication *function is defined as* $\mathit{duplicate}^n(\langle \Gamma, \tau, C^* \rangle) = \langle [\Gamma_1, \ldots, \Gamma_n], [\tau_1, \ldots, \tau_n], [C_1^*, \ldots, C_n^*] \rangle$ *where:*

– $\chi = tvars(\Gamma) \cup tvars(\tau) \cup tvars(C^*)$;
– $\mathit{duplicate}_\chi^n(\Gamma) \equiv \Gamma_1 \wedge \ldots \wedge \Gamma_n$;
– $\mathit{duplicate}_\chi^n(\tau) \equiv \tau_1 \wedge \ldots \wedge \tau_n$;
– $\mathit{duplicate}_\chi^n(C^*) \equiv C_1^* \cup \ldots \cup C_n^*$.

4.1 Constraint Generation

We define the constraint generation rules in figure 2. The constraint generation judgment is written as $\Gamma \vdash_\wedge M : \tau \mid C^*$, where given a term M, the rules generate a typing context Γ, type τ and constraints C^*. We follow [32], assigning fresh type variables to variables in [G-VAR]. No assumptions are made for the type of the term variable, allowing it to be constrained to the correct type according to the context. Similarly to the type system, there are two constraint generation rules for lambda abstractions: [G-ABSI], when the bound variable occurs free in the body, and [G-AbsK], when it does not. When the bound variable occurs free in the body, rule [G-Var] will gather type assumptions in the context. Then, rules containing several premises, [G-App], [G-App$\wedge$] and [G-Add], join the contexts under an intersection (definition 6). Due to this, the domain of the function type in the conclusion of rule [G-AbsI] corresponds to the intersection of the types of all ocurrences of the bound variable, which is given by the context in the premise of the rule. When the bound variable does not occur free in the body, there is no information regarding the type for the domain. Rule [G-AbsK] then returns an arrow type whose domain is a fresh sequence variable.

Whereas in the type system, there's a single application rule, two constraint generation rules are required: [G-App$\wedge$] and [G-App]. In [G-App$\wedge$], the type of the function term is an arrow and its domain is an intersection. Then, the type of the function term, particularly the domain $\tau_1 \wedge \ldots \wedge \tau_n$, constrains how many derivations are needed of the argument term. For each instance in the domain type of the function, a derivation of the argument is required. Furthermore, each instance must unify with its corresponding argument's type.

However, instead of following the standard approach [32] of ensuring multiple derivations of the argument, we explore a different approach. In fact, generating multiple derivations of the argument amounts to duplicating type variables found in the context, type and constraints. We made this explicit in rule [G-App$\wedge$].

If the type of the function term is not an arrow, then there is no information on the number of derivations required of the argument term, so only one is needed. Furthermore, the type of the function is constrained to be an arrow type, and its domain to match the argument's type, as specified in [G-App].

Taking the previous example in section 3, constraints are now generated for the expression:

Example 3. We have the following derivations by applying the rule:

$$\text{[G-AbsI]} \quad \emptyset \vdash_\wedge \lambda x \,.\, x\ x : \alpha_1 \wedge \alpha_2 \to \alpha_3 \mid \{\alpha_1 \doteq \alpha_2 \to \alpha_3\} \tag{4}$$

$$\text{[G-AbsI]} \quad \emptyset \vdash_\wedge \lambda y \,.\, y : \alpha_4 \to \alpha_4 \mid \emptyset \tag{5}$$

By rule [G-App$\wedge$] on derivations (4), (5) and premises (6) and (7) we have:

$$\mathit{duplicate}^2(\langle \emptyset, \alpha_4 \to \alpha_4, \emptyset \rangle) = \langle [\emptyset, \emptyset], [\alpha_5 \to \alpha_5, \alpha_6 \to \alpha_6], [\emptyset, \emptyset] \rangle \tag{6}$$

$$C = \alpha_1 \wedge \alpha_2 \doteq \alpha_5 \to \alpha_5 \wedge \alpha_6 \to \alpha_6 \tag{7}$$

$$\text{[G-App}\wedge\text{]} \quad \emptyset \vdash_\wedge (\lambda x \,.\, x\ x)\ (\lambda y \,.\, y) : \alpha_3 \mid \{\alpha_1 \doteq \alpha_2 \to \alpha_3\} \cup C$$

We show the following properties of our constraint generation algorithm:

Lemma 1 (Soundness of Constraint Generation). *If $\Gamma \vdash_\wedge M : \tau \mid C^*$ and $S \models C^*$ then $S(\Gamma) \vdash_\wedge M : S(\tau)$.*

Proof. Proof by induction on the length of the derivation tree of $\Gamma \vdash_\wedge M : \tau \mid C^*$.

Lemma 2 (Completeness of Constraint Generation). *If $S_1(\Gamma) \vdash_\wedge M : \tau$ then $\Gamma \vdash_\wedge M : \rho \mid C^*$ s.t. the domain of S_1 is disjoint from χ, and $\exists S_2$ s.t. S_2 agrees with S_1 except at χ, $S_2 \models C^*$ and $S_2(\rho) = \tau$, where χ are the fresh variables introduced in the derivation of $\Gamma \vdash_\wedge M : \rho \mid C^*$.*

Proof. Proof by induction on the length of the derivation tree of $S_1(\Gamma) \vdash_\wedge M : \tau$.

4.2 Set Unification

Type inference for simple types relies on first-order unification. However, equality constraints between idempotent and commutative intersection types are not so easy to solve. Solving such constraints involves finding the correct association between instances in both sequences. If we consider sequences as sets, this problem is equivalent to solving a set unification problem [21,20].

According to [21], a set is an arbitrary, unordered collection of elements, i.e. the order and repetition of elements do not matter. Since we consider the intersection type operator $\wedge$ as idempotent and commutative, a sequence $\tau_1 \wedge \ldots \wedge \tau_n$ can be interpreted as a set $\{\tau_1, \ldots, \tau_n\}$, whose elements are the instances of the sequence. By definition 1, a sequence can have as instances base types B, type variables α, and arrows $\sigma \to \tau$. These are the building blocks of sequences, so we define their counterparts for sets:

Definition 14 (Individuals). *The set of individuals $\mathcal{U}$ is defined as follows:*

- *if $B \in \mathcal{T}_{base}$ then $B \in \mathcal{U}$;*
- *if s, t are abstract set terms, then $\to (s, t) \in \mathcal{U}$.*

Individuals are essentially ground terms that make up our sets. Besides base types B, the arrow type is also considered an individual, however, one with two arguments.

Now we can define sets, that will act as a counterpart for sequences. According to [21,20], the full class of sets is defined as follows. For $m, n, p, q \geq 0$, the class $set(m, n, p, q)$ represents the collection of all abstract set terms $\{X_1, \ldots, X_{m'}, a_1, \ldots, a_{n'}, s_1, \ldots, s_{p'}\} \cup Y_1 \cup \ldots \cup Y_{q'}$ such that $0 \geq m' \geq m, 0 \geq n' \geq n, 0 \geq p' \geq p, 0 \geq q' \geq q$, where X_i, Y_i are variables, a_i are individuals and s_i, t_i are abstract set terms (distinct from variables).

However, the full class of sets has more expressive power than what we need to encode sequences. The language of types, as well the rank restriction (definition 5), restricts the expressive power of sequences to be less than that of sets. Only rank 1 sequences are allowed, therefore sequences cannot contain other sequences as elements. This restriction means that abstract set terms s_i inside sets are not permitted. Furthermore, extra variables Y_i have no counterpart in our sequences. Therefore, we only need a restricted fragment of the class $set(m, n, p, q)$: the class $flat(0) = \bigcup_{m \geq 0, n \geq 0} set(m, n, 0, 0)$. We then define our sets under this class:

Definition 15 (Abstract Set Terms). *An abstract set term is a term of the form: $\{X_1, \ldots, X_m, a_1, \ldots, a_n\}$, with $m, n \geq 0$.*

Therefore, rank 1 sequence solving is equivalent to the Set Unification Decision [21] problem between two flat(0) sets.

We now define the translation, allowing sequences to be encoded as abstract set terms, which can be then passed onto the unification algorithm:

Definition 16 (Types as Abstract Set Terms). *The translation function* $(\!|.|\!)$ *is defined according to the following rules:*

$$\frac{B \in \mathcal{T}_{base}}{(\!|B|\!) = B} \qquad \frac{}{(\!|\alpha|\!) = X} \qquad \frac{(\!|\sigma|\!) = s \qquad (\!|\upsilon|\!) = t}{(\!|\sigma \to \tau|\!) = \to (s, t)}$$

$$\frac{(\!|\tau_1|\!) = t_1 \ \ldots \ (\!|\tau_n|\!) = t_n}{(\!|\tau_1 \wedge \ldots \wedge \tau_n|\!) = \{t_1, \ldots, t_n\}}$$

The translation function is bijective, and its inverse is defined as follows: assuming $(\!|\sigma|\!) = s$, *then* $(\!|s|\!)^{inv} = \sigma$.

With an encoding of sequences as sets, we can unify two sets with algorithm `AbCl_unify` [21,20]. Generally, algorithm `AbCl_unify` takes a system of equations as input and returns either fail or a collection of systems in solved form. However, since the constraint solving algorithm only needs to solve one equality constraint between sequences at a time, `AbCl_unify` is only ever called with a single equation as input. The algorithm then essentially tries to find a match between the elements of the two sets, non-deterministically checking different permutations. As two sets can be unified in several ways, this algorithm is non-deterministic, i.e. provides various solutions, albeit all correct. Therefore, due to relying on `AbCl_unify`, the constraint solving algorithm is also non-deterministic. We encapsulate the unification algorithm as well as the necessary translation, and define the sequence solving procedure $C \overset{s}{\Rightarrow} S$:

Definition 17 (Sequence Solving). *Let* $\sigma \doteq \upsilon$ *be an equality constraint between two rank 1 sequences* σ *and* υ. *The sequence solving procedure* $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S_i$, *that non-deterministically returns a set of substitutions* $S_1, \ldots, S_n$, *is defined by the following steps.*

Let $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S_i$, *such that:*

1. *let* t, s *be abstract set terms such that* $(\!|\sigma|\!) = t$ *and* $(\!|\upsilon|\!) = s$;
2. *choose an arbitrary solution* $\mathcal{E}_i$ *returned by* `AbCl_unify`$(\{t = s\})$:
   (a) *for every solved form equation* $X = t' \in \mathcal{E}_i$, *if* $(\!|X|\!)^{inv} = \alpha$ *and* $(\!|t'|\!)^{inv} = \sigma'$, *then* $[\alpha \mapsto \sigma'] \in S_i$

We transcribe the soundness and completeness result from [20], from which we can then derive our own:

Theorem 1 (Soundness and Completeness of `AbCl_unify` [20]). *Given a system* $\mathcal{E}$, *let* $\mathcal{E}_1, \ldots, \mathcal{E}_n$ *be all the systems in solved form produced by the unification algorithm. Then* $Soln(\mathcal{E}) = Soln(\mathcal{E}_1)|_{vars(\mathcal{E})} \cup \ldots \cup Soln(\mathcal{E}_n)|_{vars(\mathcal{E})}$ *where* $Soln(X)$ *is the set of all ground set-unifiers of* X *and* $Soln(\mathcal{E}_i)|_{vars(\mathcal{E})}$ *is* $Soln(\mathcal{E}_i)$ *restricted to the variables of* $\mathcal{E}$.

Lemma 3 (Soundness of Sequence Solving). *If* $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S$ *then* $S \models \sigma \doteq \upsilon$.

Proof. If $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S_i$, for all $i \in 1..n$, then by definition 17: $(\!|\sigma|\!) = t$ and $(\!|\upsilon|\!) = s$; `AbCl_unify`$(\{t = s\})$ returns solutions $\mathcal{E}_1, \ldots, \mathcal{E}_n$; and for every solved form equation $X = t' \in \mathcal{E}_i$, if $(\!|X|\!)^{inv} = \alpha$ and $(\!|t'|\!)^{inv} = \sigma'$, then $[\alpha \mapsto \sigma'] \in S_i$. By theorem 1, $Soln(\{t = s\}) = Soln(\mathcal{E}_1)|_{vars(\{t=s\})} \cup \ldots \cup Soln(\mathcal{E}_n)|_{vars(\{t=s\})}$. We then have that $\mathcal{E}_i$ is a solution for $\{t = s\}$. By definition 16, $(\!|t|\!)^{inv} = \sigma$ and $(\!|s|\!)^{inv} = \upsilon$. Therefore, S_i is a solution to $\sigma \doteq \upsilon$, or rather, $S_i(\sigma) = S_i(\upsilon)$. By definition 9, $S_i \models \sigma \doteq \upsilon$.

Lemma 4 (Completeness of Sequence Solving). *If $S_1 \models \sigma \doteq \upsilon$ then $\exists S, S_2$ s.t. $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S_2$ and $S_1 = S \circ S_2$.*

Proof. If $S_1 \models \sigma \doteq \upsilon$, then by definition 16, (1) $(\!|\sigma|\!) = t$ and $(\!|\upsilon|\!) = s$. We then have that (2) `AbCl_unify`$(\{t = s\})$ returns solutions $\mathcal{E}_1, \ldots, \mathcal{E}_n$, with $i \in 1..n$. By theorem 1, we have that $Soln(\{t = s\}) = Soln(\mathcal{E}_1)|_{vars(\mathcal{E})} \cup \ldots \cup Soln(\mathcal{E}_n)|_{vars(\mathcal{E})}$. Therefore, the set of solved form equations of $\mathcal{E}_i$, for all $i \in 1..n$, represents all possible solutions of $\{t = s\}$, and each solution $\mathcal{E}_i$ is a minimal solution. (2a) For every solved form equation $X = t' \in \mathcal{E}_i$, if $(\!|X|\!)^{inv} = \alpha$ and $(\!|t'|\!)^{inv} = \sigma'$, then $[\alpha \mapsto \sigma'] \in S'_i$. By definition 17, since we have (1), (2), and (2a), then $(\sigma \doteq \upsilon) \overset{s}{\Rightarrow} S'_i$, non-deterministically for all $i \in 1..n$. One of these solutions S'_i agrees with S_1, and is a most general solution to $\sigma \doteq \upsilon$. Therefore, $\exists S, S'_i$ s.t. $S_1 = S \circ S'_i$.

4.3 Constraint Solving

The constraint solving rules are defined in figure 2. The constraint solving judgment is written as $C^* \Rightarrow S$, where given constraints C^* the rules generate substitutions S. Most rules are straightforward, following standard formulations for type inference. Rule [S-Empty] allows constraint solving to terminate: when no constraints are left, the algorithm returns the substitutions. Rule [S-Same] discards equality constraints between the same types. Rule [S-Arrow] deconstructs an equality constraint between two arrows, by constraining both the domains to each other, and both the codomains to each other.

Rule [S-Seq] solves equality constraints between two sequences by calling the sequence solving algorithm $C \overset{s}{\Rightarrow} S'$, which in turn calls the solving algorithm `AbCl_unify` from [21,20]. Resulting substitutions are then applied to the remaining constraints, and solving proceeds as usual. Due to non-determinism of `AbCl_unify`, and consequently, $C \overset{s}{\Rightarrow} S'$, this rule introduces non-determinism in the constraint solving algorithm. However, every parallel solution is either correct, or constraint solving fails.

The remaining rules are standard rules to deal with type variables. Rules [S-TVarR] and [S-SVarR] apply when the type (and sequence) variables appear on the right side, swapping the positions of the constrained types. Rules [S-TVarL] and [S-SVarL] then produce a substitution between the type (and sequence) variable and the type on the right of the constraint.

Continuing the example from section 4.1, constraints are solved:

$$[\text{S-Empty}]\ \frac{}{\emptyset \Rightarrow \emptyset} \qquad [\text{S-Same}]\ \frac{C^* \Rightarrow S}{\{\tau \doteq \tau\} \cup C^* \Rightarrow S}\ \tau \in \mathcal{T}_{base} \cup \mathcal{T}_{tvar}$$

$$[\text{S-Arrow}]\ \frac{\{\sigma \doteq \upsilon, \tau \doteq \rho\} \cup C^* \Rightarrow S}{\{\sigma \to \tau \doteq \upsilon \to \rho\} \cup C^* \Rightarrow S}$$

$$[\text{S-Seq}]\ \frac{(\tau_1 \wedge \ldots \wedge \tau_n \doteq \rho_1 \wedge \ldots \wedge \rho_m) \overset{s}{\Rightarrow} S' \qquad S'(C^*) \Rightarrow S}{\{\tau_1 \wedge \ldots \wedge \tau_n \doteq \rho_1 \wedge \ldots \wedge \rho_m\} \cup C^* \Rightarrow S \circ S'}$$

$$[\text{S-TVarR}]\ \frac{\{\alpha \doteq \tau\} \cup C^* \Rightarrow S}{\{\tau \doteq \alpha\} \cup C^* \Rightarrow S}\ \tau \notin \mathcal{T}_{tvar}$$

$$[\text{S-TVarL}]\ \frac{[\alpha \mapsto \tau]C^* \Rightarrow S}{\{\alpha \doteq \tau\} \cup C^* \Rightarrow S \circ [\alpha \mapsto \tau]}\ \tau \in \mathcal{T}_0 \textit{ and } \alpha \notin tvars(\tau)$$

$$[\text{S-SVarR}]\ \frac{\{\phi \doteq \sigma\} \cup C^* \Rightarrow S}{\{\sigma \doteq \phi\} \cup C^* \Rightarrow S}\ \sigma \notin \mathcal{T}_{svar}$$

$$[\text{S-SVarL}]\ \frac{[\phi \mapsto \sigma]C^* \Rightarrow S}{\{\phi \doteq \sigma\} \cup C^* \Rightarrow S \circ [\phi \mapsto \sigma]}\ \sigma \in \mathcal{T}_1 \textit{ and } \phi \notin tvars(\sigma)$$

Fig. 3. Constraint Solving ($C^* \Rightarrow S$)

Example 4. We now have the following constraints to solve:

$$\begin{array}{rl} & \{\alpha_1 \doteq \alpha_2 \to \alpha_3, \alpha_1 \wedge \alpha_2 \doteq \alpha_5 \to \alpha_5 \wedge \alpha_6 \to \alpha_6\} \Rightarrow \emptyset \\ [\text{S-TVarL}] & \{\alpha_2 \to \alpha_3 \wedge \alpha_2 \doteq \alpha_5 \to \alpha_5 \wedge \alpha_6 \to \alpha_6\} \Rightarrow [\alpha_1 \mapsto \alpha_2 \to \alpha_3] \end{array}$$

Due to non-determinism of $C \overset{s}{\Rightarrow} S$, there are two solutions:

$$\begin{array}{rl} [\text{S-Seq}] & \emptyset \Rightarrow [\alpha_5 \mapsto \alpha_6 \to \alpha_6] \circ [\alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto \alpha_5] \circ [\alpha_1 \mapsto \alpha_2 \to \alpha_3] \\ [\text{S-Seq}] & \emptyset \Rightarrow [\alpha_6 \mapsto \alpha_5 \to \alpha_5] \circ [\alpha_2 \mapsto \alpha_6, \alpha_3 \mapsto \alpha_6] \circ [\alpha_1 \mapsto \alpha_2 \to \alpha_3] \end{array}$$

Choosing the first solution, our expression is typed as follows:

$$\begin{array}{rl} [\text{T-AbsI}] & \emptyset \vdash_\wedge \lambda x\ .\ x\ x : ((\alpha_6 \to \alpha_6) \to \alpha_6 \to \alpha_6) \wedge (\alpha_6 \to \alpha_6) \to \alpha_6 \to \alpha_6 \\ [\text{T-AbsI}] & \emptyset \vdash_\wedge \lambda y\ .\ y : (\alpha_6 \to \alpha_6) \to \alpha_6 \to \alpha_6 \\ [\text{T-AbsI}] & \emptyset \vdash_\wedge \lambda y\ .\ y : \alpha_6 \to \alpha_6 \\ [\text{T-App}] & \emptyset \vdash_\wedge (\lambda x\ .\ x\ x)\ (\lambda y\ .\ y) : \alpha_6 \to \alpha_6 \end{array}$$

We show our constraint solving algorithm is both sound and complete:

Lemma 5 (Soundness of Constraint Solving). *If $C^* \Rightarrow S$ then $S \models C^*$.*

Proof. Proof by induction on the length of the derivation tree of $C^* \Rightarrow S$.

Lemma 6 (Completeness of Constraint Solving). *If* $S_1 \models C^*$ *then* $\exists S, S_2$ *s.t.* $C^* \Rightarrow S_2$ *and* $S_1 = S \circ S_2$.

Proof. Proof by induction on the breakdown of constraint sets C^* by the solving rules.

4.4 Algorithm

Having defined both a constraint generation and solving algorithm, we now include both in the main type inference algorithm. We also show our type inference is sound and complete.

Definition 18 (Type Inference). *The type inference procedure infer*$(M) \overset{def}{=} (\Gamma, \tau, S)$*, that given an expression* M*, non-deterministically returns a triple* (Γ, τ, S) *composed of a typing context* Γ*, type* τ *and substitutions* S*, is defined by the following steps:*

Let infer$(M) \overset{def}{=} (\Gamma, \tau, S)$*, such that:*

1. *let* Γ*,* τ *and* C^* *such that* $\Gamma \vdash_\wedge M : \tau \mid C^*$*;*
2. *let* S *such that* $C^* \Rightarrow S$*;*

Theorem 2 (Soundness). *If infer*$(M) = (\Gamma, \tau, S)$ *then* $S(\Gamma) \vdash_\wedge M : S(\tau)$.

Proof. By definition 18, we have Γ, τ and C^* such that $\Gamma \vdash_\wedge M : \tau \mid C^*$, and S such that $C^* \Rightarrow S$. By lemma 5, since $C^* \Rightarrow S$ then $S \models C^*$. By lemma 1, since $\Gamma \vdash_\wedge M : \tau \mid C^*$ and $S \models C^*$ then $S(\Gamma) \vdash_\wedge M : S(\tau)$.

Theorem 3 (Completeness). *If* $S_1(\Gamma) \vdash_\wedge M : \tau$ *then* $\exists S_2, \rho, S$ *s.t. infer*$(M) = (\Gamma, \rho, S_2)$ *and* $\tau = S \circ S_2(\rho)$.

Proof. If $S_1(\Gamma) \vdash_\wedge M : \tau$ then by lemma 2, $\Gamma \vdash_\wedge M : \rho \mid C^*$ and $\exists S_2$ s.t. S_2 agrees with S_1 except at χ, $S_2 \models C^*$ and $S_2(\rho) = \tau$, where χ are the fresh variables introduced in the derivation of $\Gamma \vdash_\wedge M : \rho \mid C^*$. By lemma 6, $\exists S, S_3$ s.t $C^* \Rightarrow S_3$ and $S_2 = S \circ S_3$. By definition 18, infer$(M) = (\Gamma, \rho, S_3)$. Then, we have that $\tau = S \circ S_3(\rho)$.

5 Conclusion and Future Work

In this paper we present a sound and complete unification-based type inference algorithm for rank-2 intersection types using set unification. One nice feature of this algorithm is its similarity with type inference for simple types, it is basically the same algorithm, replacing first-order unification by set unification.

5.1 Future Work

Using Set-Unification based Type Inference in Practice This work is carried out in the context of a larger research project, focused in the use of intersection types and gradual types for programming language design and implementation. This larger project assumes the implementation and evaluation of intersection gradual types in a functional programming language compiler. Several points need to be further developed to enable the use of the algorithm presented here in the overall project goals. Some important points to address are:

1. Extension of the term language with recursive definitions. This will enable to apply our algorithm to a more realistic language and will address the known problems related with decidability for recursive definitions [34,30].
2. Add support to let expressions and conditional expressions. Most likely, in the case of conditional expressions, this will mean extending the type language with union types.

Theoretical Issues The work presented here inspires the following possible future work:

1. Types here use associative, commutative and idempotent intersections. In the last years non-idempotent intersections have been successfully used to obtain quantitative information of program behaviour [9,1,2]. We believe it is rather promising to use multiset unification (usually based on solving diophantine equations) in the same way we use set unification, to infer types in this particular setting.
2. Investigate the complexity of our type inference algorithm. Being exponential for sure, because this is the complexity of the type inference problem for rank-2 intersection types, we want to study the exact complexity of our type inference algorithm and investigate if using set-unification may have some impact on the overall efficiency of type inference.
3. Extension to higher rank intersection types. Here we use a simple form of set unification where there cannot be sets inside sets. We conjecture that using those nested sets limited to a fixed level of nesting will result in type inference algorithms for higher (but finite) rank intersection types.
4. Study the relation of our approach with β-unification [35] and other forms of unification. Unification theory is a wide research field and studying in detail the relations between different unification algorithms, which, in this case, are used for the same purpose may shed some light on their relations and also contribute to the area of unification theory.

Acknowledgements This work was partially financially supported by the portuguese Fundação para a Ciência e a Tecnologia, under the PhD grant number SFRH/BD/145183/2019 and by Base Funding - UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC - funded by national funds through the FCT/MCTES (PIDDAC).

References

1. Accattoli, B., Graham-Lengrand, S., Kesner, D.: Tight typings and split bounds. Proc. ACM Program. Lang. **2**(ICFP), 94:1–94:30 (2018)
2. Alves, S., Kesner, D., Ventura, D.: A quantitative understanding of pattern matching. In: 25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway. LIPIcs, vol. 175, pp. 3:1–3:36. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
3. Ângelo, P., Florido, M.: Type inference for rank 2 gradual intersection types. In: Bowman, W.J., Garcia, R. (eds.) Trends in Functional Programming. pp. 84–120. Springer International Publishing, Cham (2020). `https://doi.org/10.1007/978-3-030-47147-7_5`
4. Ângelo, P., Florido, M.: Type inference for rank-2 intersection types using set unification. Tech. rep., Faculdade de Ciências & LIACC, Universidade do Porto (May 2022), `https://raw.githubusercontent.com/pedroangelo/papers/master/angelo2022type_complete.pdf`
5. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda Calculus with Types. Perspectives in logic, Cambridge University Press (2013)
6. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. Journal of Symbolic Logic **48**(4), 931–940 (1983). `https://doi.org/10.2307/2273659`
7. Bettini, L., Bono, V., Dezani-Ciancaglini, M., Giannini, P., Venneri, B.: Java & lambda: a featherweight story. Log. Methods Comput. Sci. **14**(3) (2018)
8. Boudol, G., Zimmer, P.: On type inference in the intersection type discipline. Electronic Notes in Theoretical Computer Science **136**, 23–42 (2005). `https://doi.org/https://doi.org/10.1016/j.entcs.2005.06.016`, `https://www.sciencedirect.com/science/article/pii/S1571066105050589`, proceedings of the Third International Workshop on Intersection Types and Related Systems (ITRS 2004)
9. Bucciarelli, A., Kesner, D., Ventura, D.: Non-idempotent intersection types for the Lambda-Calculus. Logic Journal of the IGPL **25**(4), 431–464 (07 2017). `https://doi.org/10.1093/jigpal/jzx018`
10. Carlier, S., Wells, J.B.: Type inference with expansion variables and intersection types in system e and an exact correspondence with β-reduction. In: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. p. 132–143. PPDP '04, Association for Computing Machinery, New York, NY, USA (2004). `https://doi.org/10.1145/1013963.1013980`, `https://doi.org/10.1145/1013963.1013980`
11. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. Proc. ACM Program. Lang. **1**(ICFP), 41:1–41:28 (Aug 2017). `https://doi.org/10.1145/3110285`
12. Castagna, G., Lanvin, V., Petrucciani, T., Siek, J.G.: Gradual typing: A new perspective. Proc. ACM Program. Lang. **3**(POPL), 16:1–16:32 (Jan 2019). `https://doi.org/10.1145/3290329`
13. Coppo, M., Dezani-Ciancaglini, M.: An extension of the basic functionality theory for the λ-calculus. Notre Dame Journal of Formal Logic **21**(4), 685–693 (10 1980). `https://doi.org/10.1305/ndjfl/1093883253`
14. Coppo, M., Giannini, P.: Principal types and unification for simple intersection type systems. Information and Computation **122**(1), 70 – 96 (1995). `https://doi.org/https://doi.org/10.1006/inco.1995.1141`

15. Coppo, M.: An extended polymorphic type system for applicative languages. In: Dembiński, P. (ed.) Mathematical Foundations of Computer Science 1980. pp. 194–204. Springer Berlin Heidelberg, Berlin, Heidelberg (1980)
16. Curry, H.B.: Functionality in Combinatory Logic. Proceedings of the National Academy of Science **20**(11), 584–590 (Nov 1934). `https://doi.org/10.1073/pnas.20.11.584`
17. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 207–212. POPL '82, ACM, New York, NY, USA (1982). `https://doi.org/10.1145/582153.582176`
18. Damiani, F.: Rank 2 intersection types for modules. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. p. 67–78. PPDP '03, Association for Computing Machinery, New York, NY, USA (2003). `https://doi.org/10.1145/888251.888259`
19. Damiani, F., Giannini, P.: A decidable intersection type system based on relevance. In: Hagiya, M., Mitchell, J.C. (eds.) Theoretical Aspects of Computer Software. pp. 707–725. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
20. Dovier, A., Omodeo, E., Pontelli, E., Rossi, G.: A language for programming in logic with finite sets. J. Log. Program. **28**, 1–44 (01 1996)
21. Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory and Practice of Logic Programming **6**(6), 645–701 (2006). `https://doi.org/10.1017/S1471068406002730`
22. Dudenhefner, A., Martens, M., Rehof, J.: The algebraic intersection type unification problem. Log. Methods Comput. Sci. **13** (2017)
23. Dunfield, J.: Elaborating intersection and union types. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 17–28. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012). `https://doi.org/10.1145/2364527.2364534`, `https://doi.org/10.1145/2364527.2364534`
24. Dunfield, J., Pfenning, F.: Type assignment for intersections and unions in call-by-value languages. In: Gordon, A.D. (ed.) Foundations of Software Science and Computation Structures. pp. 250–266. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
25. Florido, M., Damas, L.: Linearization of the lambda-calculus and its relation with intersection type systems. Journal of Functional Programming **14**(5), 519–546 (2004). `https://doi.org/10.1017/S0956796803004970`
26. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: A type-theoretic interpretation. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 802–815. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). `https://doi.org/10.1145/2837614.2837629`, `https://doi.org/10.1145/2837614.2837629`
27. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. p. 268–277. PLDI '91, Association for Computing Machinery, New York, NY, USA (1991). `https://doi.org/10.1145/113445.113468`
28. Girard, J.Y.: Une extension de Ľinterpretation de gödel a Ľanalyse, et son application a Ľelimination des coupures dans Ľanalyse et la theorie des types. In: Fenstad, J. (ed.) Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic and the Foundations of Mathematics, vol. 63, pp. 63–92. Elsevier

(1971). https://doi.org/https://doi.org/10.1016/S0049-237X(08)70843-7, https://www.sciencedirect.com/science/article/pii/S0049237X08708437
29. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press, Cambridge (1989)
30. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. **15**(2), 253–289 (1993). https://doi.org/10.1145/169701.169692, https://doi.org/10.1145/169701.169692
31. Hindley, J.R.: Basic Simple Type Theory. Cambridge University Press (1997)
32. Jim, T.: Rank 2 type systems and recursive definitions. Tech. rep., Cambridge, MA, USA (1995)
33. Jim, T.: What are principal typings and what are they good for? In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 42–53. POPL '96, ACM, New York, NY, USA (1996). https://doi.org/10.1145/237721.237728
34. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Trans. Program. Lang. Syst. **15**(2), 290–311 (1993)
35. Kfoury, A.J., Wells, J.B.: Principality and decidable type inference for finite-rank intersection types. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 161–174. POPL '99, ACM, New York, NY, USA (1999). https://doi.org/10.1145/292540.292556
36. Kfoury, A., Wells, J.: Principality and type inference for intersection types using expansion variables. Theoretical Computer Science **311**(1), 1 – 70 (2004). https://doi.org/10.1016/j.tcs.2003.10.032
37. Leivant, D.: Polymorphic type inference. In: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 88–98. POPL '83, Association for Computing Machinery, New York, NY, USA (1983). https://doi.org/10.1145/567067.567077
38. PALSBERG, J., PAVLOPOULOU, C.: From polyvariant flow information to intersection and union types. Journal of Functional Programming **11**(3), 263–317 (2001). https://doi.org/10.1017/S095679680100394X
39. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
40. Pottinger, G.: A type assignment for the strongly normalizable lambda-terms. In: Hindley, J., Seldin, J. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 561–577. Academic Press (1980)
41. Reynolds, J.C.: Towards a theory of type structure. In: Programming Symposium, Proceedings Colloque Sur La Programmation. p. 408–423. Springer-Verlag, Berlin, Heidelberg (1974)
42. Reynolds, J.C.: Design of the Programming Language Forsythe, pp. 173–233. Birkhäuser Boston, Boston, MA (1997). https://doi.org/10.1007/978-1-4612-4118-8_9
43. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (Jan 1965). https://doi.org/10.1145/321250.321253
44. Ronchi Della Rocca, S.: Principal type scheme and unification for intersection type discipline. Theor. Comput. Sci. **59**(1–2), 181–209 (Jul 1988). https://doi.org/10.1016/0304-3975(88)90101-6
45. Urzyczyn, P.: The emptiness problem for intersection types. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 300–309 (1994). https://doi.org/10.1109/LICS.1994.316059
46. Van Bakel, S.J.: Intersection type disciplines in lambda calculus and applicative term rewriting systems. Amsterdam: Mathematisch Centrum (1993)

47. Wand, M.: A simple algorithm and proof for type inference. Fundamenta Informaticae **10**(2), 115–121 (1987)