A tale of intersection types

Conference Paper · July 2020

DOI: 10.1145/3373718.3394733

2 authors:

Viviana Bono
Università degli Studi di Torino
85 PUBLICATIONS **1,215** CITATIONS
SEE PROFILE

Mariangiola Dezani-ciancaglini
Università degli Studi di Torino
261 PUBLICATIONS **4,810** CITATIONS
SEE PROFILE

A tale of intersection types

Viviana Bono and Mariangiola Dezani-Ciancaglini
Computer Science Department Università di Torino, Italy
bono, dezani@di.unito.it

July 25, 2020

Abstract

Intersection types have come a long way since their introduction in the Seventies. They have been exploited for characterising behaviours of λ-terms and π-calculus processes, building λ-models, verifying properties of higher-order programs, synthesising code, and enriching the expressivity of programming languages. This paper is a light overview of intersection types and some of their applications.

1 Introduction

Type polymorphism is a desirable feature for programming languages and is indeed a fascinating subject, as it implies a form of infinity. Universal types, namely System F and its extension F_ω [44], are a direct design of polymorphism, given that universality contains a notion of infinity. Intersection types are polymorphic types, too, because they permit us to list explicitly all possible (interesting) types of a program. They are then a formalism to describe portions of infinity, the ones that matter in a certain context. As a finitary description of infinity, intersection types have come a long way since their introduction in the Seventies. They have been exploited for characterising behaviours of λ-terms and π-calculus processes, building λ-models, verifying properties of higher-order programs, synthesising code, and enriching the expressivity of programming languages.

In this work, we would like to take the reader along a path of discovery into the origins and applications of intersection types. This path cannot be complete, given the amplitude of the subject. We avoided technical details as much as possible in favour of intuition, sometimes also sacrificing precision.

It was difficult to choose the subjects of this paper: the main criterion was to cover as many different contexts and periods of time as possible, in the given space. We hope that the reader will enjoy this guided walk through intersection types and will forgive us for our omissions.

The paper is organised as follows. The first three sections introduce intersection types for the λ-calculus, discuss their syntactic properties, and describe

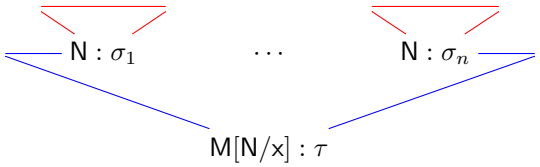


Figure 1: Derivation of $\mathsf{M}[\mathsf{N}/\mathsf{x}] : \tau$.

their use for computing the complexity of λ-terms. The next three sections illustrate the role of intersection types in defining the semantics of the λ-calculus and of the π-calculus. Section 8 presents one of the more unexpected and interesting applications of intersection types, i.e., the ability of model checking higher-order functions. The remaining sections are oriented to programming languages from different points of view: we start from automatic software composition and type decoration, and then we present some aspects of the languages $\mathbb{C}$Duce, Forsythe and Java.

The sections are self-contained, with the exception of Section 6 which requires Section 5, and can be read independently. We only assume some familiarity with the λ-calculus and types (a good introduction to the subjects can be found, for instance, in the keystone books [18, 68]).

2 The birth of intersection types

In the Seventies, looking at the world through the λ-calculus, a natural question to be asked was how to obtain types that were preserved not only by subject reduction (a mandatory request!), but also by subject expansion [29,30,69]. We take the standard syntax for pure λ-terms [7] (Definition 2.1.1):

$$\mathsf{M} ::= \mathsf{x} \mid \lambda\mathsf{x}.\mathsf{M} \mid \mathsf{M}\mathsf{M},$$

with the usual notational conventions, and define reduction $\longrightarrow$ as the reflexive, transitive and contextual closure of the β-rule:

$$(\lambda\mathsf{x}.\mathsf{M})\mathsf{N} \longrightarrow \mathsf{M}[\mathsf{N}/\mathsf{x}],$$

where $\mathsf{M}[\mathsf{N}/\mathsf{x}]$ denotes the λ-term obtained by the (capture free) substitution of x by N in M [32] (page 94).

Type preservation under subject expansion means that if $\mathsf{M}[\mathsf{N}/\mathsf{x}]$ has a type τ, written $\mathsf{M}[\mathsf{N}/\mathsf{x}] : \tau$, then also $(\lambda\mathsf{x}.\mathsf{M})\mathsf{N}$ has the same type, i.e., $(\lambda\mathsf{x}.\mathsf{M})\mathsf{N} : \tau$. This is enough for compositional typings, a common feature. A tree representation of a type derivation for $\mathsf{M}[\mathsf{N}/\mathsf{x}] : \tau$ is given in Figure 1, where n is the number of occurrences of x in M. If $n = 1$ the drawing simplifies as in Figure 2, and we can derive the same type for $(\lambda\mathsf{x}.\mathsf{M})\mathsf{N}$ as shown in Figure 3, just using the standard rules of arrow introduction and elimination. If $n = 0$, we get the result in Figure 4, where N has no type. To tackle this case, it is useful to have

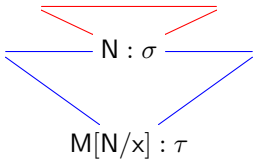


Figure 2: Derivation of $\mathsf{M}[\mathsf{N}/\mathsf{x}] : \tau$ when x occurs exactly once in M.

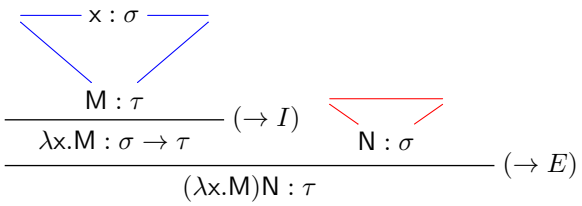


Figure 3: Derivation of $(\lambda\mathsf{x}.\mathsf{M})\mathsf{N} : \tau$ when x occurs exactly once in M.

a universal type, which we name ω, and the typing rule (ω) assigning ω to any arbitrary λ-term. Therefore, we type the expansion by the derivation shown in Figure 5.

Lastly, if $n > 1$, we need to assign the types $\sigma_1, \ldots, \sigma_n$ to the variable x. We add then the intersection type constructor $\wedge$ with the expected introduction and elimination rules:

$$\frac{\mathsf{M} : \tau \quad \mathsf{M} : \sigma}{\mathsf{M} : \tau \wedge \sigma}\,(\wedge I) \qquad \frac{\mathsf{M} : \tau \wedge \sigma}{\mathsf{M} : \tau}\,(\wedge E) \qquad \frac{\mathsf{M} : \tau \wedge \sigma}{\mathsf{M} : \sigma}\,(\wedge E)$$

Figure 6 shows the typing of the expansion for $n = 2$, the generalisation to an arbitrary $n > 2$ being straightforward.

To sum up, intersection types are generated by the grammar:

$$\tau, \sigma ::= \phi \mid \omega \mid \tau \to \tau \mid \tau \wedge \tau,$$

where ϕ ranges over an enumerable set of type variables. We adopt the convention that $\wedge$ has precedence over $\to$. Unless otherwise stated, we assume $\wedge$ to be idempotent, commutative and associative with neutral element ω.

We write typing judgments as $\Gamma \vdash \mathsf{M} : \tau$, where a *basis* Γ is a finite mapping from term variables to types:

$$\Gamma ::= \emptyset \mid \Gamma, \mathsf{x} : \tau.$$

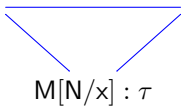


Figure 4: Derivation of $\mathsf{M}[\mathsf{N}/\mathsf{x}] : \tau$ when x does not occur in M.

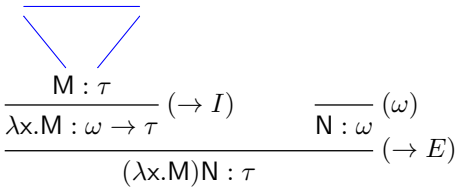


Figure 5: Derivation of $(\lambda \mathsf{x}.\mathsf{M})\mathsf{N} : \tau$ when x does not occur in M.

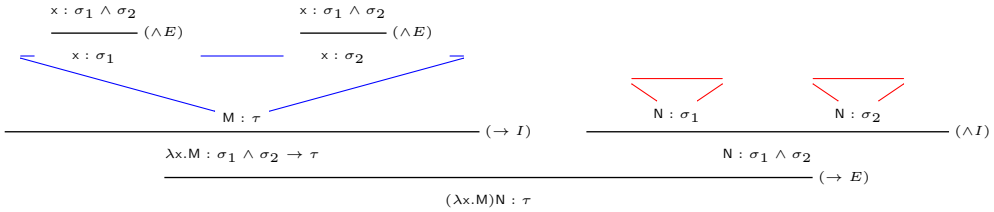


Figure 6: Derivation of $(\lambda \mathsf{x}.\mathsf{M})\mathsf{N} : \tau$ when x occurs twice in M.

The typing rules are given in Figure 7. A nice property of intersection types is the possibility of typing all *normal forms*, i.e., λ-terms that cannot be reduced, with types capturing completely the normal forms. In fact, from these typings, a simple algorithm can extract the corresponding normal forms [30]. Note that some normal forms, for example the auto-application $\lambda \mathsf{x}.\mathsf{x}\mathsf{x}$, cannot be typed using the simple types à la Curry [49] (Definition 15.6). The type derivation of Figure 8 shows how intersection can deal instead with auto-application, by considering the same variable both as a function and as an argument for that function.

The notion of approximation formalises the idea that the result of a computation is gradually built, step by step, by the computation itself (in contrast to the set theoretic static notion of function as graph). A program may fail to terminate, but still go on building some information, as, for example, a program building an infinite stream. In the λ-calculus, recursion is simulated through the fixed point combinator $\mathbf{Y}$, which does not have a normal form, but repeatedly unfolds its recursive definition. In the intersection type system $\mathbf{Y}$ has an

$$\frac{}{\Gamma, \mathsf{x} : \tau \vdash \mathsf{x} : \tau}\ (Ax) \qquad \frac{}{\Gamma \vdash \mathsf{M} : \omega}\ (\omega)$$

$$\frac{\Gamma, \mathsf{x} : \sigma \vdash \mathsf{M} : \tau}{\Gamma \vdash \lambda \mathsf{x}.\mathsf{M} : \sigma \to \tau}\ (\to I) \qquad \frac{\Gamma \vdash \mathsf{M} : \sigma \to \tau \quad \Gamma \vdash \mathsf{N} : \sigma}{\Gamma \vdash \mathsf{M}\mathsf{N} : \tau}\ (\to E)$$

$$\frac{\Gamma \vdash \mathsf{M} : \sigma \quad \Gamma \vdash \mathsf{M} : \tau}{\Gamma \vdash \mathsf{M} : \sigma \wedge \tau}\ (\wedge I) \qquad \frac{\Gamma \vdash \mathsf{M} : \sigma \wedge \tau}{\Gamma \vdash \mathsf{M} : \tau}\ (\wedge E)$$

Figure 7: Typing rules for intersection types.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{x} : (\sigma \to \tau) \wedge \sigma \vdash \mathsf{x} : (\sigma \to \tau) \wedge \sigma}\,(Ax)}{\mathsf{x} : (\sigma \to \tau) \wedge \sigma \vdash \mathsf{x} : \sigma \to \tau}\,(\wedge E) \qquad \dfrac{\dfrac{}{\mathsf{x} : (\sigma \to \tau) \wedge \sigma \vdash \mathsf{x} : (\sigma \to \tau) \wedge \sigma}\,(Ax)}{\mathsf{x} : (\sigma \to \tau) \wedge \sigma \vdash \mathsf{x} : \sigma}\,(\wedge E)}{\dfrac{\mathsf{x} : (\sigma \to \tau) \wedge \sigma \vdash \mathsf{xx} : \tau}{\vdash \lambda \mathsf{x.xx} : (\sigma \to \tau) \wedge \sigma \to \tau}\,(\to I)}\,(\to E)$$

Figure 8: Typing auto-application.

infinite number of types, which correspond exactly to its unfoldings, i.e., its approximants:

$$\begin{array}{ccccc} \omega & (\omega \to \tau_1) \to \tau_1 & (\omega \to \tau_1) \wedge (\tau_1 \to \tau_2) \to \tau_2 & \ldots \\ \ldots & (\omega \to \tau_1) \wedge \ldots \wedge (\tau_n \to \tau_{n+1}) \to \tau_{n+1} & \ldots & \end{array}$$

This sequence of types represents $\mathbf{Y}$'s infinite normal form.

As should be clear from the above discussion, the presented type system enjoys both subject reduction and subject expansion.

Theorem 2.1 *(Subject Reduction and Expansion)*

1. *If* $\Gamma \vdash \mathsf{M} : \tau$ *and* $\mathsf{M} \longrightarrow \mathsf{N}$*, then* $\Gamma \vdash \mathsf{N} : \tau$*.*
2. *If* $\Gamma \vdash \mathsf{M} : \tau$ *and* $\mathsf{N} \longrightarrow \mathsf{M}$*, then* $\Gamma \vdash \mathsf{N} : \tau$*.*

A feature of intersection types is the ability to characterise computational properties of λ-terms [37, 69], properties which are widely studied in the literature. In particular we recall the following classes of λ-terms:

- a λ-term is *weakly normalising* if it reduces to a λ-term of the shape $\lambda\mathsf{x}.\mathsf{M}$ (called a λ-abstraction) or of the shape $\mathsf{xM}_1 \ldots \mathsf{M}_n$ $(n \geq 0)$ (called a λ-free term);
- a λ-term is *solvable* if it reduces to a λ-term of the shape $\lambda\mathsf{x}_1 \ldots \mathsf{x}_n.\mathsf{yM}_1 \ldots \mathsf{M}_m$ $(n, m \geq 0)$ (called a head normal form);
- a λ-term is *normalising* if it reduces to a normal form;
- a λ-term is *strongly normalising* if all its reductions terminate.

These computational behaviours have a clear correspondence with the typings of λ-terms.

Theorem 2.2 *(Computational Properties)*

1. $\Gamma \vdash \mathsf{M} : \omega \to \omega$ *iff* M *is weakly normalising.*
2. $\Gamma \vdash \mathsf{M} : \tau_1 \to \ldots \to \tau_n \to \phi$ *$(n \geq 0)$ iff* M *is solvable.*
3. $\Gamma \vdash \mathsf{M} : \tau$ *and* Γ, τ *do not contain* ω *iff* M *is normalising.*

4. $\Gamma \vdash \mathsf{M} : \tau$ *without using* ω *in the derivation iff* M *is strongly normalising.*

Other classes of λ-terms are characterised in [37]. Notably intersection types can then be considered the first example of *behavioural types*, i.e., types able to represent the behaviour of terms. Behavioural types are widely used to ensure soundness of communication protocols [53]. On the negative side, the expressivity of the intersection type system implies the undecidability of intersection type inference, i.e., given a λ-term, we are unable to say if we can derive a type for it. For this reason various restrictions have been proposed in the literature. We will discuss one of them in Section 10.

Inhabitation, i.e., the existence of a λ-term that can be typed from a given basis with a given type, is also undecidable. This problem was open for a long time and then brilliantly solved in [79].

The type system with intersection types is a conservative extension of the type system with simple types à la Curry [8, 47]. System F and its extension F_ω [44] are very expressive thanks to the use of universal quantifiers. Because all λ-terms typable in F_ω are strongly normalising [44], they can also be typed with intersection types. On the contrary, [78] gives a λ-term which has an intersection type but no type in F_ω.

3 Expansion

Can we add the power of intersection types to a calculus or to a programming language? The answer is yes, and *expansion* is one of the key ingredients. Let us see what it is.

Rule $(\wedge I)$ duplicates the typings of the same λ-terms. For example we can derive:

$$\dfrac{\mathsf{y} : \sigma \vdash \mathsf{y} : \sigma \qquad \dfrac{\mathsf{y} : \sigma, \mathsf{x} : \phi \vdash \mathsf{x} : \phi}{\mathsf{y} : \sigma \vdash \lambda\mathsf{x}.\mathsf{x} : \phi \to \phi}}{\mathsf{y} : \sigma \vdash \mathsf{y}(\lambda\mathsf{x}.\mathsf{x}) : \psi}$$

where $\sigma = (\phi \to \phi) \to \psi$, but also

$$\dfrac{\mathsf{y} : \tau \vdash \mathsf{y} : \tau \qquad \dfrac{\dfrac{\mathsf{y} : \tau, \mathsf{x} : \phi_1 \vdash \mathsf{x} : \phi_1}{\mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : \phi_1 \to \phi_1} \qquad \dfrac{\mathsf{y} : \tau, \mathsf{x} : \phi_2 \vdash \mathsf{x} : \phi_2}{\mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : \phi_2 \to \phi_2}}{\mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : (\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2)}}{\mathsf{y} : \tau \vdash \mathsf{y}(\lambda\mathsf{x}.\mathsf{x}) : \psi}$$

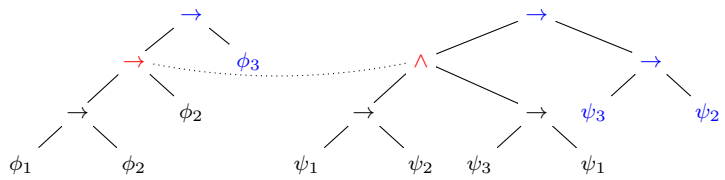
where $\tau = (\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2) \to \psi$. This example suggests the usefulness of replacing some types by an intersection of types obtained by renaming the original ones. Here $\phi \to \phi$ is replaced by $(\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2)$. Clearly the choice of the replaced types must follow syntactic rules. In this case the replacement is called *expansion* [30].

Expansion is also needed when we want to type the application of two λ-terms. Consider for example MN[1] where:

[1] This example is taken from [22].

- $\mathsf{M} = \lambda\mathsf{x}.\mathsf{x}(\lambda\mathsf{y}.\mathsf{yz})$ has type
$$(((\phi_1 \to \phi_2) \to \phi_2) \to \phi_3) \to \phi_3,$$
starting from the assumption $\mathsf{z} : \phi_1$;
- $\mathsf{N} = \lambda\mathsf{fa}.\mathsf{f}(\mathsf{fa})$ has type
$$(\psi_1 \to \psi_2) \wedge (\psi_3 \to \psi_1) \to \psi_3 \to \psi_2.$$

We need to unify the two types $((\phi_1 \to \phi_2) \to \phi_2) \to \phi_3$ and $(\psi_1 \to \psi_2) \wedge (\psi_3 \to \psi_1) \to \psi_3 \to \psi_2$, which can be drawn as the following trees:



The problem here is the clash between an arrow and an intersection, connected by a dotted line. Expansion allows us to have an intersection in the type of M:

$$\begin{array}{lccc} \mathsf{z}: & \phi_1 & \vdash \mathsf{M}: & (\phi_1 \to \phi_2) \to \phi_2 \to \phi_3 \\ & \Downarrow & & \Downarrow \\ \mathsf{z}: & \phi_1' \wedge \phi_1'' & \vdash \mathsf{M}: & ((\phi_1' \to \phi_2') \to \phi_2') \wedge ((\phi_1'' \to \phi_2'') \to \phi_2'') \to \phi_3 \end{array}$$

Note that also the assumption $\mathsf{z} : \phi_1$ must be expanded to $\mathsf{z} : \phi_1' \wedge \phi_1''$. After the expansion, unification using substitution is straightforward:

$$\mathsf{z} : \phi_1' \wedge \phi_1'' \vdash \mathsf{M} : (\sigma \to \tau) \to \tau,$$

$$\vdash \mathsf{N} : \sigma \to \tau,$$

where
$$\sigma = ((\phi_1' \to \phi_2') \to \phi_2') \wedge ((\phi_1'' \to \phi_1' \to \phi_2') \to \phi_1' \to \phi_2')$$
and $\tau = (\phi_1'' \to \phi_1' \to \phi_2') \to \phi_2'$. We can then derive

$$\mathsf{z} : \phi_1' \wedge \phi_1'' \vdash \mathsf{MN} : \tau.$$

A very elegant formalisation of expansion is done through *expansion variables*; see [22] and the references therein. The application of an expansion variable to a type τ can replace τ with the intersection of types obtained from τ by renaming variables. For instance, there is an expansion variable $\mathcal{E}$ such that the application $\mathcal{E}(\phi \to \phi) \to \psi$ produces

$$(\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2) \to \psi.$$

The most interesting use of expansion variables is inside type derivations, for example

$$\dfrac{\mathsf{y} : \sigma \vdash \mathsf{y} : \sigma \qquad \dfrac{\mathsf{y} : \sigma, \mathsf{x} : \mathcal{E}\phi \vdash \mathsf{x} : \mathcal{E}\phi}{\mathsf{y} : \sigma \vdash \lambda\mathsf{x}.\mathsf{x} : \mathcal{E}(\phi \to \phi)}}{\mathsf{y} : \sigma \vdash \mathsf{y}(\lambda\mathsf{x}.\mathsf{x}) : \psi}$$

where $\sigma = \mathcal{E}(\phi \to \phi) \to \psi$ produces

$$\dfrac{\dfrac{\dfrac{}{\mathsf{y} : \tau, \mathsf{x} : \phi_1 \vdash \mathsf{x} : \phi_1}}{\mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : \phi_1 \to \phi_1} \quad \dfrac{\dfrac{}{\mathsf{y} : \tau, \mathsf{x} : \phi_2 \vdash \mathsf{x} : \phi_2}}{\mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : \phi_2 \to \phi_2}}{\dfrac{\mathsf{y} : \tau \vdash \mathsf{y} : \tau \qquad \mathsf{y} : \tau \vdash \lambda\mathsf{x}.\mathsf{x} : (\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2)}{\mathsf{y} : \tau \vdash \mathsf{y}(\lambda\mathsf{x}.\mathsf{x}) : \psi}}$$

where $\tau = (\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2) \to \psi$.

Expansion variables can be used to implement expansion in a simple, clean and flexible way.

Principal typing is a key notion for type assignment systems. The best definition of principal typing [80] uses the following partial order on the pairs basis/type:

$\langle \Gamma, \tau \rangle \sqsubseteq \langle \Gamma', \tau' \rangle$ if for all M: $\quad \Gamma \vdash \mathsf{M} : \tau$ implies $\Gamma' \vdash \mathsf{M} : \tau'$.

Definition 3.1 $\langle \Gamma, \tau \rangle$ *is the* principal typing *for* M *if*
$\Gamma \vdash \mathsf{M} : \tau$ *and* $\Gamma' \vdash \mathsf{M} : \tau'$ *implies* $\langle \Gamma, \tau \rangle \sqsubseteq \langle \Gamma', \tau' \rangle$.

For example in our type system:

- $\langle \mathsf{y} : (\phi \to \phi) \to \psi, \psi \rangle$ is the principal typing of $\mathsf{y}(\lambda\mathsf{x}.\mathsf{x})$;
- $\langle \mathsf{z} : \phi_1, (((\phi_1 \to \phi_2) \to \phi_2) \to \phi_3) \to \phi_3 \rangle$ is the principal typing of $\lambda\mathsf{x}.\mathsf{x}(\lambda\mathsf{y}.\mathsf{yz})$;
- $\langle \emptyset, (\psi_1 \to \psi_2) \wedge (\psi_3 \to \psi_1) \to \psi_3 \to \psi_2 \rangle$ is the principal typing of $\lambda\mathsf{fa}.\mathsf{f}(\mathsf{fa})$;
- $\langle \mathsf{z} : \phi_1' \wedge \phi_1'', (\phi_1'' \to \phi_1' \to \phi_2') \to \phi_2' \rangle$ is the principal typing of $(\lambda\mathsf{x}.\mathsf{x}(\lambda\mathsf{y}.\mathsf{yz}))(\lambda\mathsf{fa}.\mathsf{f}(\mathsf{fa}))$.

It is interesting to build, starting from the principal typing of a λ-term, the other typings for the same λ-term. Substitution is enough to derive all typings from the principal typing in the simple type system à la Curry [10]. However, for intersection types, substitution is not enough: as the example at the beginning of this section shows, we need also expansion [30]. This is why expansion is crucial if one wants to add intersection types to a calculus or a programming language.

4 Complexity of β-reduction

Complexity theory has a prominent role in computer science. Turing machines are mainly used as a model of computation [46], but also the λ-calculus is exploited [59] in this sense.

In Section 2 we have seen that intersection types characterise computational properties of λ-terms. Actually, they can do more: they can give quantitative bounds to the number of β-reductions needed to reach the normal forms and to the number of symbols in the normal forms [11, 12]. The key move to increase the expressivity of intersection types is to consider *non-idempotent* intersections

in *relevant* type assignment systems. These relevant systems can then be used to count the number of the occurrences of variables during the reduction process.

More recently, *exact bounds* have been provided for various reduction strategies with a uniform methodology [1]. We consider here only the reduction to head normal form. The size of the head normal form $\lambda \mathsf{x}_1 \ldots \mathsf{x}_n.\mathsf{y}\mathsf{M}_1 \ldots \mathsf{M}_m$ is $n + m$.

In the type system of [1] there are atomic types that can only be assigned to λ-terms producing results of given shapes. In particular, considering the reduction to head normal form, the type abs can only be assigned to λ-terms whose head normal forms are λ-abstractions, while the type $\mathsf{neutral}$ can only be assigned to λ-terms whose head normal forms are λ-free terms. The typing judgments carry two counters, the counter b for the step number and the counter r for the size of the normal form, i.e., they are of the shape

$$\Gamma \vdash^{(b,r)} \mathsf{M} : \tau.$$

Three typing rules are:

$$\frac{}{\mathsf{x} : \tau \vdash^{(0,0)} \mathsf{x} : \tau}\ (Ax\mathsf{c}) \qquad \frac{\Gamma, x : \sigma \vdash^{(b,r)} \mathsf{M} : \tau}{\Gamma \vdash^{(b+1,r)} \lambda\mathsf{x}.\mathsf{M} : \sigma \to \tau}\ (\to I\mathsf{c})$$

$$\frac{\Gamma \vdash^{(b,r)} \mathsf{M} : \sigma \to \tau \quad \Gamma' \vdash^{(b',r')} \mathsf{N} : \sigma}{\Gamma \bigwedge \Gamma' \vdash^{(b+b',r+r')} \mathsf{MN} : \tau}\ (\to E\mathsf{c})$$

where the basis in the conclusion of rule $(\to E\mathsf{c})$ is obtained by taking the intersections of the types for the same variables:

$$\begin{array}{rcl} \Gamma \bigwedge \Gamma' & = & \{\mathsf{x} : \sigma \wedge \sigma' \mid \mathsf{x} : \sigma \in \Gamma \ \&\ \mathsf{x} : \sigma' \in \Gamma'\} \cup \\ & & \{\mathsf{x} : \sigma \mid \mathsf{x} : \sigma \in \Gamma \ \&\ \mathsf{x} \notin dom(\Gamma')\} \cup \\ & & \{\mathsf{x} : \sigma' \mid \mathsf{x} : \sigma' \in \Gamma' \ \&\ \mathsf{x} \notin dom(\Gamma)\}, \end{array}$$

and $dom(\Gamma) = \{\mathsf{x} \mid \mathsf{x} : \sigma \in \Gamma\}$. Rule $(\to I\mathsf{c})$ can only be used when the typed λ-abstraction will be applied and reduced to obtain the head normal form. For this reason, the counter b is incremented by 1, while the counter r remains the same. In rule $(\to E\mathsf{c})$ the values of the counters for the application are simply obtained by summing those of the application arguments.

Writing a for abs and b for $\mathsf{abs} \to \mathsf{abs}$, an example of type derivation taken from [1] is shown in Figure 9, where we omit the derivation of $\vdash^{(1,1)} \lambda\mathsf{z}.\mathsf{z} : \mathsf{a} \wedge \mathsf{b}$ which requires other typing rules. The head reduction of $(\lambda\mathsf{y}.(\lambda\mathsf{x}.\mathsf{xy})\mathsf{y})(\lambda\mathsf{z}.\mathsf{z})$ returns $\lambda\mathsf{z}.\mathsf{z}$ in three steps. Therefore, the judgment

$$\vdash^{(3,1)} (\lambda\mathsf{y}.(\lambda\mathsf{x}.\mathsf{xy})\mathsf{y})(\lambda\mathsf{z}.\mathsf{z}) : \mathsf{a}$$

gives the exact number of β-reductions and the exact size of the obtained head normal form. Instead $\lambda\mathsf{y}.(\lambda\mathsf{x}.\mathsf{xy})\mathsf{y}$ reduces in one step to $\lambda\mathsf{y}.\mathsf{yy}$, which has size 2, and we derive

$$\vdash^{(2,0)} \lambda\mathsf{y}.(\lambda\mathsf{x}.\mathsf{xy})\mathsf{y} : \mathsf{a} \wedge \mathsf{b} \to \mathsf{a}.$$

As a matter of fact, in the judgments $\Gamma \vdash^{(b,r)} \mathsf{M} : \tau$, the counter b is always an upper bound on the length of the head reduction and r is always a lower bound

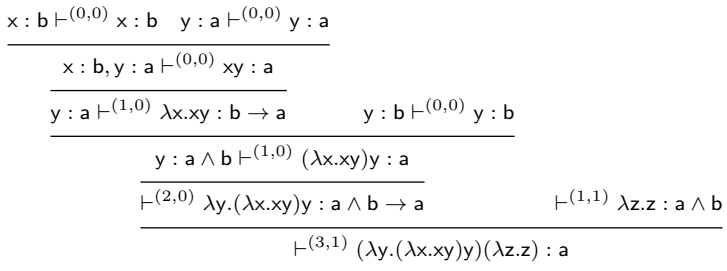


Figure 9: A type derivation with complexity counters.

on the size of the head normal form. These counters become exact only when both the types in the bases and the predicates are one of the two constants abs and neutral.

5 Type theories

Since the very beginning of intersection types, there have been two main variants of the system discussed in Section 2: the restriction to *strict types* and the addition of subtyping. Strict types are defined without intersections on the right-side of the arrows. This requires two syntactic categories:

$$\tau ::= \phi \mid \omega \mid \sigma \to \tau \qquad\qquad \sigma ::= \tau \mid \sigma \wedge \sigma$$

For example $(\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2)$ is not a strict type, but $(\phi_1 \to \phi_1) \wedge (\phi_2 \to \phi_2) \to \psi$ is a strict type. We refer the interested reader to [6] for a comprehensive treatment of strict types. We would like to remark that the expansion of derivations (discussed in Section 3) is simpler for strict types than for the other ones, because it is more constrained.

Subtyping between intersection types is naturally induced by set-theoretic inclusion:

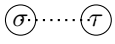
$$\tau \leq \tau \qquad \sigma \wedge \tau \leq \sigma \qquad \sigma \wedge \tau \leq \tau \qquad \tau \leq \omega$$

$$\frac{\sigma \leq \tau \quad \sigma \leq \rho}{\sigma \leq \tau \wedge \rho} \qquad\qquad \frac{\sigma \leq \rho \quad \rho \leq \tau}{\sigma \leq \tau}$$

Considering the arrow as the function space constructor, we also have:

$$(\sigma \to \tau) \wedge (\sigma \to \rho) \leq \sigma \to \tau \wedge \rho \qquad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

These last subtypings are illustrated in Figure 10, where



denotes the set of functions mapping inputs of type σ to outputs of type τ. The second rule depicts the co-variant and contra-variant behaviour of the arrow type constructor with respect to subtyping.

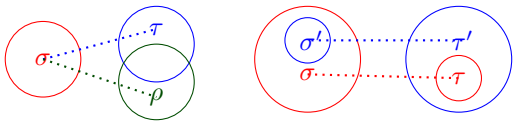


Figure 10: Subtyping between arrow and intersection types.

Many subtyping relations have been shown to be useful for characterising computational properties of λ-terms [37] and/or for building λ-models, see Section 6. Subtyping can also modify the set of basic types. For example, by assuming $\phi \leq \psi$ for all type variables ϕ, ψ, we obtain a type system with only two basic types, ϕ and ω. It is then natural to parametrise intersection types assignment systems by subtyping. These parametrised systems are usually called *intersection type theories* [9] (Sections 13.1 and 13.2).

Definition 5.1 *An* intersection type theory $\mathcal{T}$ *is a set of sentences of the form* $\sigma \leq \tau$ *satisfying at least the axioms and rules induced by set-theoretic inclusion and involving only the intersection type constructor.*

We write $\sigma \leq_{\mathcal{T}} \tau$ for derivability in the type theory $\mathcal{T}$.

Definition 5.2 *The* intersection type assignment system induced by the type theory $\mathcal{T}$ *is obtained by adding to the typing rules of Figure 7 the subsumption rule*

$$\frac{\Gamma \vdash \mathsf{M} : \sigma \quad \sigma \leq_{\mathcal{T}} \tau}{\Gamma \vdash \mathsf{M} : \tau}\ (\leq_{\mathcal{T}})$$

We write $\Gamma \vdash_{\mathcal{T}} \mathsf{M} : \tau$ for derivability in the type assignment system induced by the type theory $\mathcal{T}$.

Note that rule $(\wedge E)$ is derived in $\vdash_{\mathcal{T}}$ for all $\mathcal{T}$.

As expected, the subsumption rule gives more types to λ-terms, for example, for all $\mathcal{T}$ including the subtyping rule for the co-variance and the contra-variance of the arrow, we obtain $\vdash_{\mathcal{T}} \lambda \mathsf{x}.\mathsf{x} : (\phi \to \psi) \to \phi \wedge \phi' \to \psi$. This judgement does not hold for the type assignment system described in Section 2.

6 λ-models

It is folklore that the λ-calculus can be considered as the pure core of functional programming. An important indicator of this fact is the influence of Scott's λ-models [74] on the denotational semantics of programming languages [75]. In this section we show how to use type theories for building finitary logical descriptions of λ-models.

We start by recalling the definition of λ-model, using the notions of environment and applicative structure [48].

Definition 6.1 *1. An* environment $\mathtt{E}$ *in the set* $\mathcal{D}$ *is a total mapping from term variables to elements of* $\mathcal{D}$.

2. *An applicative structure is a pair $\langle \mathcal{D}, \cdot \rangle$, where $\mathcal{D}$ is a set and $\cdot$ is a binary operation on $\mathcal{D}$.*

As usual, we denote by $\mathtt{E}[\mathsf{x} := \mathsf{d}]$ the environment that applied to x returns d and applied to $\mathsf{y} \neq \mathsf{x}$ returns $\mathtt{E}(\mathsf{y})$.

Definition 6.2 *A λ-model is a triple $\langle \mathcal{D}, \cdot, [\![\ ]\!]^{\mathcal{D}} \rangle$, where $\langle \mathcal{D}, \cdot \rangle$ is an applicative structure, $[\![\ ]\!]^{\mathcal{D}}$ is a mapping from λ-terms and environments in $\mathcal{D}$ to elements of $\mathcal{D}$, and $[\![\ ]\!]^{\mathcal{D}}$ satisfies:*

1. $[\![\mathsf{x}]\!]^{\mathcal{D}}_{\mathtt{E}} = \mathtt{E}(\mathsf{x})$;
2. $[\![\mathsf{MN}]\!]^{\mathcal{D}}_{\mathtt{E}} = [\![\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}} \cdot [\![\mathsf{N}]\!]^{\mathcal{D}}_{\mathtt{E}}$;
3. $[\![\lambda\mathsf{x}.\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}} = [\![\lambda\mathsf{y}.\mathsf{M}[\mathsf{y}/\mathsf{x}]]\!]^{\mathcal{D}}_{\mathtt{E}}$;
4. $\forall \mathsf{d} \in \mathcal{D}. [\![\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}[\mathsf{x}:=\mathsf{d}]} = [\![\mathsf{N}]\!]^{\mathcal{D}}_{\mathtt{E}[\mathsf{x}:=\mathsf{d}]}$ *implies* $[\![\lambda\mathsf{x}.\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}} = [\![\lambda\mathsf{x}.\mathsf{N}]\!]^{\mathcal{D}}_{\mathtt{E}}$;
5. $\mathtt{E}(\mathsf{x}) = \mathtt{E}'(\mathsf{x})$ *for all variables* x *which occur free in* M *implies* $[\![\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}} = [\![\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}'}$;
6. $[\![\lambda\mathsf{x}.\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}} \cdot \mathsf{d} = [\![\mathsf{M}]\!]^{\mathcal{D}}_{\mathtt{E}[\mathsf{x}:=\mathsf{d}]}$.

The set $\mathcal{D}$ is called the *domain* of the λ-model.

The intersection type assignment system allows us to build λ-models by considering as domains sets of filters induced by type theories [4]. Filters of types are sets closed under subtyping and intersection.

Definition 6.3 *Let $\mathcal{T}$ be a type theory. A non-empty set of types $\mathtt{F}$ is a $\mathcal{T}$-*filter *if:*

1. $\sigma \leq_{\mathcal{T}} \tau$ *and* $\sigma \in \mathtt{F}$ *imply* $\tau \in \mathtt{F}$;
2. $\sigma, \tau \in \mathtt{F}$ *imply* $\sigma \wedge \tau \in \mathtt{F}$.

We use $\mathcal{F}^{\mathcal{T}}$ for the set of $\mathcal{T}$-filters. If X is a set of types we denote by $\uparrow^{\mathcal{T}} X$ the smallest $\mathcal{T}$-filter containing X. It is easy to verify that $(\mathcal{F}^{\mathcal{T}}, \subseteq)$ is an ω-algebraic complete lattice, with bottom $\uparrow^{\mathcal{T}}\{\omega\}$ and top the set of all types.

We can turn $\mathcal{F}^{\mathcal{T}}$ into an applicative structure by defining:

$$\mathtt{F} \cdot^{\mathcal{T}} \mathtt{F}' = \uparrow^{\mathcal{T}}\{\tau \mid \sigma \to \tau \in \mathtt{F} \ \&\ \sigma \in \mathtt{F}'\}.$$

A natural interpretation of λ-terms in $\mathcal{F}^{\mathcal{T}}$ is:

$$[\![\mathsf{M}]\!]^{\mathcal{F}^{\mathcal{T}}}_{\mathtt{E}} = \{\tau \mid \exists \Gamma.\ \Gamma \models \mathtt{E} \ \&\ \Gamma \vdash \mathsf{M} : \tau\},$$

where $\Gamma \models \mathtt{E}$ (to be read: the basis Γ agrees with the environment $\mathtt{E}$) if $\Gamma(\mathsf{x}) = \sigma$ implies $\sigma \in \mathtt{E}(\mathsf{x})$. Therefore we can ask:

"Is $\langle \mathcal{F}^{\mathcal{T}}, \cdot^{\mathcal{T}}, [\![\ ]\!]^{\mathcal{F}^{\mathcal{T}}} \rangle$ a λ-model?"

This clearly amounts to checking whether the conditions of Definition 6.2 are satisfied. It is easy to see that all type theories verify all conditions but the last one. The last condition can be ensured by considering the following subtyping condition:

$$\bigwedge_{i\in I}(\sigma_i \to \tau_i) \leq_{\mathcal{T}} \sigma \to \tau \text{ implies}$$
$$\exists J \subseteq I.\sigma \leq_{\mathcal{T}} \bigwedge_{i\in J} \sigma_i \;\&\; \bigwedge_{i\in J} \tau_i \leq_{\mathcal{T}} \tau.$$

A type theory is called *β-sound* if it satisfies this condition. We have then a family of λ-models, called *filter models*, one for each β-sound type theory. Note that β-soundness is not a necessary condition to obtain filter models; a counterexample is given in [3].

It is interesting to note that we can build a filter model isomorphic to any $\mathcal{D}_\infty$ model as defined in [74]. The advantage is that we obtain finitary logical descriptions of $\mathcal{D}_\infty$ models.

Let $(\mathcal{D}, \sqsubseteq)$, $(\mathcal{D}', \sqsubseteq')$ be ω-algebraic complete lattices. We denote by $[\mathcal{D} \to \mathcal{D}']$ the set of continuous functions from $\mathcal{D}$ to $\mathcal{D}'$. We equip $[\mathcal{D} \to \mathcal{D}']$ with the pointwise partial order $\sqsubseteq''$ defined by: $f \sqsubseteq'' g$ if $f(\mathsf{d}) \sqsubseteq' g(\mathsf{d})$ for all $\mathsf{d} \in \mathcal{D}$. We use $\mathsf{id}_{\mathcal{D}}$ for the identity function on $\mathcal{D}$. The same symbol $\sqsubseteq$ will denote the partial order on various ω-algebraic complete lattices without ambiguity, because the compared elements will always belong to the same set.

Definition 6.4 *1. Let $(\mathcal{D}_0, \sqsubseteq)$ be an ω-algebraic complete lattice and*
$\mathsf{i}_0 : \mathcal{D}_0 \to [\mathcal{D}_0 \to \mathcal{D}_0]$ $\quad$ $\mathsf{j}_0 : [\mathcal{D}_0 \to \mathcal{D}_0] \to \mathcal{D}_0$
be such that:

- $\mathsf{i}_0 \circ \mathsf{j}_0 \sqsubseteq \mathsf{id}_{[\mathcal{D}_0 \to \mathcal{D}_0]}$;
- $\mathsf{j}_0 \circ \mathsf{i}_0 = \mathsf{id}_{\mathcal{D}_0}$.

2. We define:

- $\mathcal{D}_{n+1} = [\mathcal{D}_n \to \mathcal{D}_n]$;
- $\mathsf{i}_n(f) = \mathsf{i}_{n-1} \circ f \circ \mathsf{j}_{n-1}$ *for* $f \in \mathcal{D}_n$;
- $\mathsf{j}_n(g) = \mathsf{j}_{n-1} \circ g \circ \mathsf{i}_{n-1}$ *for* $g \in \mathcal{D}_{n+1}$.

3. The set $\mathcal{D}_\infty$ is:

$$\mathcal{D}_\infty = \{\mathsf{d} \in \Pi_{n\in\mathbb{N}}\mathcal{D}_n \mid \forall n \in \mathbb{N}.\mathsf{d} \downarrow n \in \mathcal{D}_n \;\&\; \mathsf{j}_n(\mathsf{d} \downarrow n+1) = \mathsf{d} \downarrow n\},$$

where $\mathsf{d} \downarrow n$ is the projection of d on $\mathcal{D}_n$,
and the partial order on $\mathcal{D}_\infty$ is:

$$\mathsf{d} \sqsubseteq \mathsf{d}' \text{ if } \forall n \in \mathbb{N}.\mathsf{d} \downarrow n \sqsubseteq \mathsf{d}' \downarrow n.$$

To build the type theory $\mathcal{T}_\infty$ inducing a filter model isomorphic to a given $\mathcal{D}_\infty$ model we need some notation.

Let $\mathcal{K}(\mathcal{D})$ be the set of compact elements of $\mathcal{D}$,
$\mathsf{d} \mapsto \mathsf{d}' : \mathcal{D} \to \mathcal{D}'$ be the step function defined by:

$$\mathsf{d} \mapsto \mathsf{d}'(\mathsf{d}'') = \begin{cases} \mathsf{d}' & \text{if } \mathsf{d} \sqsubseteq \mathsf{d}'', \\ \bot_{\mathcal{D}'} & \text{otherwise,} \end{cases}$$

and $\sim_{\mathcal{T}}$ be short for $\leq_{\mathcal{T}}$ and $\geq_{\mathcal{T}}$.

Definition 6.5 *[5] The type theory $\mathcal{T}_\infty$ for the ω-algebraic complete lattice $\mathcal{D}_\infty$ is given by:*

1. *the set of atomic types is the set $\mathcal{K}(\mathcal{D}_0)$, where $\bot$ is renamed ω;*
2. *if $\mathsf{d}, \mathsf{d}' \in \mathcal{K}(\mathcal{D}_0)$ and $\mathsf{d} \sqsubseteq \mathsf{d}'$, then $\mathsf{d}' \leq_{\mathcal{T}_\infty} \mathsf{d}$;*
3. *$\mathsf{d} \wedge \mathsf{d}' \sim_{\mathcal{T}_\infty} \mathsf{d} \sqcup \mathsf{d}'$;*
4. *if $\mathsf{i}_0(\mathsf{d}) = \bigsqcup_{i \in I} (\mathsf{d}_i \mapsto \mathsf{d}'_i)$, then $\mathsf{d} \sim_{\mathcal{T}_\infty} \bigwedge_{i \in I} (\mathsf{d}_i \to \mathsf{d}'_i)$;*
5. *$\leq_{\mathcal{T}_\infty}$ contains all axioms and rules listed in Section 5.*

In this construction the set of atomic types is the set of compact elements of $\mathcal{D}_0$ with the reverse order. Intersection corresponds to join and arrow corresponds to the step function. Each atomic type is equivalent to an intersection of arrows between atomic types as prescribed by the mapping i_0.

For example, if $\mathcal{D}_0 = \{\bot, \top\}$, $\bot \sqsubseteq \top$, and $\mathsf{i}_0(\mathsf{d}) = \bot \mapsto \mathsf{d}$, then we obtain the atomic types $\omega, \top$ with

$$\top \leq_{\mathcal{T}_\infty} \omega \qquad \omega \sim_{\mathcal{T}_\infty} \omega \to \omega \qquad \top \sim_{\mathcal{T}_\infty} \omega \to \top$$

plus all axioms and rules listed in Section 5.

Intersection type systems can also provide finitary logical descriptions of other kinds of λ-models. In particular, [65] gives a correspondence with stable λ-models [13] and [66] gives a correspondence with relational λ-models [19]. The switch to other semantics requires deep changes. In fact, the type assignment systems in [65] and [66] are relevant. Moreover intersection is not idempotent in [66].

In the following section we discuss how intersection types can characterise the behaviour of π-calculus processes.

7 Behaviour of π-processes

Proving properties for concurrent systems is still a challenge, even though type systems offering guarantees of various forms of safety have been explored extensively. There are, for example, type systems guaranteeing deadlock and livelock freedom of processes [55, 58, 64, 76], type systems for checking the correctness of communications among processes, such as session types [51], and type systems for proving the termination of processes [35, 82]. However, little was known about the existence of type systems able not only to guarantee but also to *characterise* some relevant property of concurrent processes, i.e., yielding *completeness* in addition to soundness, until the intersection type system in [33].

The process calculus in [33] is based on the polyadic asynchronous localised π-calculus [73] (Section 5.6), where a notion of asynchronous hyperlocalised π-process is added. This "hyperlocalisation" means that the resulting calculus is actually a fragment of the localised calculus, i.e., with further constraints on the presence of input names under input prefixes (the limitations introduced by these restrictions are discussed in [33], where it is shown that they have a modest impact on the expressiveness of the calculus)[2].

As an example of the use of (non-idempotent) intersection types in [33] we consider the typing of parallel composition. Typing judgments are of the shape $\Gamma \vdash P :: \Delta$, where:

- Γ associates types and channel dependencies to channels used by the process P as inputs;
- Δ associates types to channels used by P in outputs.

The parallel composition of processes is typed by taking intersections of channel types. An example is:

$$\frac{(x; y, z) : \tau_1 \vdash P :: y : \sigma_1, z : \sigma \quad (x; y) : \tau_2 \vdash Q :: y : \sigma_2}{(x; y, z) : \tau_1 \wedge \tau_2 \vdash P \,|\, Q :: y : \sigma_1 \wedge \sigma_2, z : \sigma}$$

where $(x; y_1, \ldots, y_n)$ means that the use of the channels $y_1, \ldots, y_n$ in outputs depends on a reception on channel x.

The intersection-based type system in [33] defines and characterises (by means of a completeness theorem) a safety property for π-processes, named *good behaviour*, which is essentially a form of deadlock-freedom and may-termination. The verification of the good-behaviour property is shown to be very hard. In particular, it is proved to be Π^0_2-complete, essentially because π-processes are non-deterministic. This shows that good behaviour of π-processes is not the same as termination in purely sequential languages, for which the property of being terminating (in any reasonable sense) can be proved by finitary systems. The intuition behind this type system is that it does not capture the "whole" good behaviour by means of a finite type derivation, but each derivation does capture a fraction of it, and the type system is complete in the sense that it does not miss any of these fractions.

The system stems from a recently introduced construction [62], which is based on the following ideas: (*i*) intersection types can be seen as approximations in linear logic; (*ii*) a programming language has an intersection-flavoured type discipline if it can be encoded in linear logic; (*iii*) it is known that the π-calculus itself can be translated in linear logic. Many encodings are present the literature, such as the one in [50], which is exploited in [33].

Essentially the Π^0_2-completeness makes this type system not immediate for practical use, however in [33] the authors depict non-trivial examples in which it is possible to describe totally the type derivations for a process, which is

[2] A justification of this restriction is impossible without introducing deep technical details.

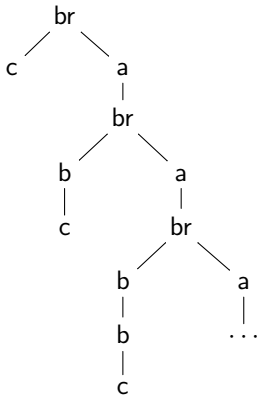


Figure 11: Tree generated by the recursion scheme of the running example.

equivalent to describing its entire behaviour. This kind of description could be a form of parametric derivation, suggesting a type discipline in the style of bounded linear logic [45], which is known to be related to intersection types [62].

8 Model checking = type checking

Higher-order functional programs can be translated into higher-order recursion schemes. A *recursion scheme* is a kind of tree grammar for generating a single, possibly infinite, ranked tree with the start symbol S as the root. As a running example, we use the following recursion scheme[3]:

$$S \longrightarrow F\,\mathsf{c} \qquad F\,\mathsf{x} \longrightarrow \mathsf{br}\,\mathsf{x}(\mathsf{a}(F(\mathsf{b}\mathsf{x})))$$

which generates the tree shown in Figure 11.

In this way, verifying properties of higher-order functional programs corresponds to model checking of recursion schemes. Model checking in [56] amounts to verify whether a *trivial automaton* [2] accepts the tree generated by the recursion scheme. A trivial automaton is a quadruple $(\Sigma,\, Q,\, \delta,\, q_0)$, where Σ is a ranked alphabet, Q is the set of states with initial state q_0, and δ is a partial map from $Q \times dom(\Sigma)$ to Q^* which respects ranks, i.e., if $\delta(q, \mathsf{a}) = q_1 \ldots q_k$, then $k = \Sigma(\mathsf{a})$.
An example is $\mathcal{B} = (\Sigma,\, \{q_0, q_1\},\, \delta,\, q_0)$, where

$$\Sigma = \{\mathsf{br} \mapsto 2,\, \mathsf{a} \mapsto 1,\, \mathsf{b} \mapsto 1,\, \mathsf{c} \mapsto 0\}$$

and

$$\delta(q_0, \mathsf{br}) = q_0 q_0 \qquad \delta(q_1, \mathsf{br}) = q_1 q_1 \qquad \delta(q_0, \mathsf{a}) = q_0$$
$$\delta(q_0, \mathsf{b}) = \delta(q_1, \mathsf{b}) = q_1 \qquad \delta(q_0, \mathsf{c}) = \delta(q_1, \mathsf{c}) = \epsilon$$

Figure 12 shows that the tree generated by the recursion scheme of the running example is accepted by $\mathcal{B}$.

[3]This example and the following ones are taken from [56].

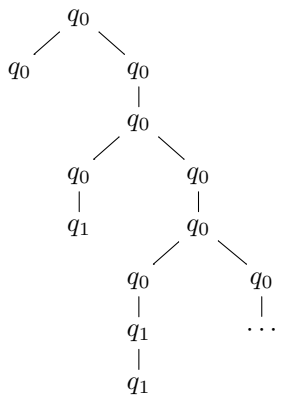


Figure 12: A run tree of $\mathcal{B}$ for the tree in Figure 11.

Intersection types transform automata acceptance in type checking. The basic types are the states of the automata, and hence the types of trees. An arrow type $q_1 \to q_2$ represents a function mapping a tree of type q_1 into a tree of type q_2. An intersection type $q_1 \wedge q_2$ represents a tree having both types q_1 and q_2. The types of the terminals are prescribed by the transition function δ, i.e., if $\delta(q, \mathsf{a}) = q_1 \dots q_k$, then we have the axiom

$$\Gamma \vdash \mathsf{a} : q_1 \to \ldots \to q_k \to q$$

For example, the axioms for the automaton $\mathcal{B}$ are

$$\Gamma \vdash \mathsf{br} : q_0 \to q_0 \to q_0 \qquad \Gamma \vdash \mathsf{br} : q_1 \to q_1 \to q_1$$
$$\Gamma \vdash \mathsf{a} : q_0 \to q_0 \qquad \Gamma \vdash \mathsf{b} : q_0 \to q_1 \qquad \Gamma \vdash \mathsf{b} : q_1 \to q_1$$
$$\Gamma \vdash \mathsf{c} : q_0 \qquad \Gamma \vdash \mathsf{c} : q_1$$

We must check if $\Gamma \vdash S : q_0$ for some Γ, where the non-terminals have types refining their sorts. This condition on Γ ensures that there are only a finite number of types we can assume for the non-terminals. In the running example, the sorts of the non-terminals are $S : \circ$ and $F : \circ \to \circ$, where $\circ$ is the sort of trees. Taking $\Gamma = \{S : q_0,\ F : q_0 \wedge q_1 \to q_0\}$, we can derive $\Gamma \vdash F\,\mathsf{c} : q_0$ and

$$\Gamma \vdash \lambda \mathsf{x}.\mathsf{br}\,\mathsf{x}(\mathsf{a}(F(\mathsf{bx}))) : q_0 \wedge q_1 \to q_0,$$

which imply $\Gamma \vdash S : q_0$. Notice that, in typing $\lambda \mathsf{x}.\mathsf{br}\,\mathsf{x}(\mathsf{a}(F(\mathsf{bx})))$ with $q_0 \wedge q_1 \to q_0$, the first occurrence of x is typed by q_0 and the second occurrence of x is typed by q_1.

The keystone paper [56] contains a complete treatment including a practical model checking algorithm with implementation and experimental results.

In [57] the verification of higher-order recursion schemes amounts to check if the trees generated by the schemes satisfy a modal μ-calculus formula. An intersection type system is built such that a recursion scheme is typable if and

only if the generated tree satisfies the formula. Given that the modal μ-calculus is more expressive than trivial automata, the construction of the corresponding type system is trickier than the one of [56].

9 Program synthesis

Retrieving and composing automatically compatible pieces of software from a library are tasks at the core of efficient code reuse. A theoretical approach to such tasks is illustrated in [38], where a framework for automatic composition synthesis from a repository of software components is defined. This is a form of typed-based code synthesis and it relies on combinatory logic with intersection types [70]. Software components are typed combinators, and an algorithm for inhabitation, i.e., "is there a combinatory term N with type τ relative to an environment Δ?", can be used to synthesise compositions. Here, Δ represents a repository of components indicated only by their names (combinators) and their types (intersection types), τ specifies the synthesis goal, an inhabitant N is a program obtained by an applicative combination of components in Δ, and is automatically constructed (synthesized) by the inhabitation algorithm. In particular, because Δ may vary, this is an instance of the *relativized* inhabitation problem. Indeed, differently to the standard combinatory logic for which a fixed basis of combinators is usually considered, this particular inhabitation problem is relativized to an arbitrary environment Δ, which is part of the input.

In [70], the authors consider expressive combinatory logics under the restriction that axioms are not interpreted schematically but *literally*, corresponding to a monomorphic interpretation of types, resulting in *finite combinatory logic*. They show that the provability (inhabitation) problem for finite combinatory logic with intersection types is EXPTIME-complete with or without subtyping (note that standard intersection type subtyping is in PTIME). This implies, from an application point of view, that intersection types are an expressive specification formalism for supporting automatic functional composition synthesis. For example, starting from the standard sets of combinators $\{\mathbf{S}, \mathbf{K}\}$ [49] (Definition 2.1), where $\mathbf{S} = \lambda xyz.xz(yz)$ and $\mathbf{K} = \lambda xy.x$, we can look for an inhabitant of $\phi \to \phi$. In order to get the expected result, i.e., $\mathbf{SKK}$, we need to take the types $\sigma \to \tau \to \phi \to \phi$ for $\mathbf{S}$ and $\sigma \wedge \tau$ for $\mathbf{K}$, where $\sigma = \phi \to (\omega \to \phi) \to \phi$ and $\tau = \phi \to \omega \to \phi$.

The variant *k-bounded combinatory logic* (named BCL_k) [39] is obtained by imposing the bound k on the depth of types that can be used to instantiate polymorphic combinator types. In [38], BCL_0 (that is, level 0-bounded polymorphism) is exploited. BCL_0 is already very expressive, as it is possible to define within BCL_0 a framework for 2-EXPTIME-complete synthesis problems, equivalent in complexity to other known synthesis frameworks (e.g., variants of temporal logic and propositional dynamic logic).

A step forward from [38] is in [14], where the same approach is basing on mixins, instead of generic software components. Here, a library is a repository of mixin-based object-oriented code. Mixins [17] are a form of linearised multi-

ple inheritance and are essentially composable functions over classes returning classes. With this focus on composition instead of inheritance, they are an ideal program construct to constitute the library Δ.

With $\{C_1 : \sigma_1, \ldots, C_p : \sigma_p, M_1 : \tau_1, \ldots, M_q : \tau_q\} = \Delta$ as the abstract specification of a library including classes C_i and mixins M_j with interfaces as types σ_i and τ_j, respectively, and given a type τ specifying an unknown class, a combinatory synthesis of classes via intersection-typed mixin combinators is performed.

The first result of [14] is to combine expressive-enough features of type systems used for typing classes and mixins, and the system of intersection types. Starting from a type assignment system of intersection and record types [60], a record merge operation is added, to obtain mixin combinators via extensible records and a subsequent suitable type system.

The second result is to show that it is possible to type classes and mixins in a expressive-enough way, where the former are essentially recursive records and the latter are made of a combination of fixed-point combinators and record merge. Such terms, which usually require recursive types, can be typed in the resulting system by means of an iterative procedure exploiting the ability of intersection types to represent approximations of the potentially infinite unfolding of recursive definitions. This is shown by the typing of the fixed point combinator $\mathbf{Y}$, discussed in Section 2.

The third result concerns how to deal with non-monotonic properties of record merge (for instance, the absence of labels), which are incompatible with the existing theory of BCL_k synthesis. Therefore, in [14] there is an encoding of intersection types with record types and merge into a conservative extension of bounded combinatory logic, where unary type constructors are monotonic and distribute over intersection. The encoding is proved sound and partially complete.

10 Bounded dimension

Programming languages with intersection types require decidable type checking. This can be achieved by writing types in the code. Examples are the $\mathbb{C}$Duce function in Section 11, the Forsythe typing rule in Section 12, and the Java code in Section 13. The theoretical basis is the formulation of a λ-calculus with intersection types à la Church. This means that λ-terms are decorated by their types. There is no obvious solution, for example we can derive

$$\vdash \lambda \mathsf{x}.\mathsf{x} : (\phi \to \phi) \wedge (\psi \to \psi),$$

but what is a suitable decoration for the variable x in this λ-term? In the literature there are many proposals, see [16, 61, 81] and the references there. In particular the relevant parallel term constructor $|$ representing the intersection is introduced in [16]. This allows us to obtain, for any type derivation in the system of Section 2, a corresponding type decorated term. For example, the typed term $\lambda \mathsf{x}^{\phi}.\mathsf{x}^{\phi} | \lambda \mathsf{y}^{\psi}.\mathsf{y}^{\psi}$ corresponds to $\vdash \lambda \mathsf{x}.\mathsf{x} : (\phi \to \phi) \wedge (\psi \to \psi)$.

The amount of type annotations can be reduced by means of *bidirectional type checking* [34]. The key idea is to distinguish between terms for which a type can be synthesised from terms that can be checked against a given type.

We think that a new and interesting solution is represented by *dimensional intersection type calculi* [40,41]. The typing judgments are of the shape $\Gamma \vdash \mathtt{P} : \tau$, where $\mathtt{P}$ is an *elaboration*, i.e., a term where each sub-term is decorated with the set of types assigned to it. For example, in the standard intersection type system we can derive the following judgments for identity:

$$\vdash \lambda \mathsf{x}.\mathsf{x} : \phi \to \phi, \qquad \vdash \lambda \mathsf{x}.\mathsf{x} : \phi \wedge \psi \to \phi,$$
$$\vdash \lambda \mathsf{x}.\mathsf{x} : (\phi \to \phi) \wedge (\psi \to \psi).$$

The corresponding judgments in dimensional intersection type calculi are:

$$\vdash (\lambda \mathsf{x}.\mathsf{x}\langle\phi\rangle)\langle\phi \to \phi\rangle : \phi \to \phi,$$
$$\vdash (\lambda \mathsf{x}.\mathsf{x}\langle\phi\rangle)\langle\phi \wedge \psi \to \phi\rangle : \phi \wedge \psi \to \phi,$$
$$\vdash (\lambda \mathsf{x}.\mathsf{x}\langle\phi, \psi\rangle)\langle\phi \to \phi, \psi \to \psi\rangle : (\phi \to \phi) \wedge (\psi \to \psi),$$

where each sub-term is decorated by the set of the types assigned to it. These decorations are enclosed between angle brackets.

A crucial role in these calculi is played by the norms of elaborations, which induce the dimensions of λ-terms with respect to bases and types, as in the following definition.

Definition 10.1 *1. The* norm $||\mathtt{P}||$ *of* $\mathtt{P}$ *is the maximum number of types that occur in the same set.*

2. The dimension of M at Γ *and* τ *is the minimum* n *such that:*

- M *is obtained from* $\mathtt{P}$ *by erasing decorations;*
- $\Gamma \vdash \mathtt{P} : \tau$*;*
- n *is the norm of* $\mathtt{P}$*;*

for some $\mathtt{P}$*.*

For example:

$$||(\lambda \mathsf{x}.\mathsf{x}\langle\phi\rangle)\langle\phi \to \phi\rangle|| = 1,$$
$$||(\lambda \mathsf{x}.\mathsf{x}\langle\phi\rangle)\langle\phi \wedge \psi \to \phi\rangle|| = 1,$$
$$||(\lambda \mathsf{x}.\mathsf{x}\langle\phi, \psi\rangle)\langle\phi \to \phi, \psi \to \psi\rangle|| = 2,$$

which imply that:

- the dimension of $\lambda \mathsf{x}.\mathsf{x}$ at $\emptyset$ and $\phi \to \phi$ is 1;
- the dimension of $\lambda \mathsf{x}.\mathsf{x}$ at $\emptyset$ and $\phi \wedge \psi \to \phi$ is 1;
- the dimension of $\lambda \mathsf{x}.\mathsf{x}$ at $\emptyset$ and $(\phi \to \phi) \wedge (\psi \to \psi)$ is 2.

Type preservation under subject reduction holds and the dimension cannot increase, as the following theorem states.

Theorem 10.2 *(Subject Reduction) If* M *has dimension* m *at* Γ *and* τ *and* $\mathsf{M} \longrightarrow \mathsf{N}$*, then* N *has dimension* $n \leq m$ *at* Γ *and* τ*.*

Dimensional intersection type calculi naturally induce meaningful restrictions of the standard system for which essential decision problems become decidable.

Theorem 10.3 *(Typability) Given a* λ*-term* M *and an integer* $n > 0$ *it is decidable whether* M *can be typed in dimension* n*.*

This problem is shown to be PSPACE-complete [41].

Theorem 10.4 *(Inhabitation) Given a basis* Γ*, a type* τ *and an integer* $n > 0$*, the existence of a* λ*-term with dimension* n *at* Γ *and* τ *is decidable.*

This problem is EXPSPACE-complete [40].

11 Semantic subtyping

The accuracy of a type systems for characterising programs' behaviour is important, in order to allow type checkers to spot more errors and to accept more correct programs. Subtyping creates a hierarchy of types, from the less precise ones to the more precise ones, for the same code (for example, the number 3 has both types Int and Real, with $\mathsf{Int} < \mathsf{Real}$).

Subtyping is usually defined syntactically in an axiomatic way, as we did in Section 5. Semantic subtyping (see [43] and the references there) instead starts from a *set-theoretic model of types*. In this model a type τ is interpreted as the set of values having type τ:

$$[\![\tau]\!] = \{\mathsf{v} \mid \vdash \mathsf{v} : \tau\}.$$

Then subtyping is simply defined as subset inclusion between type interpretations:

$$\tau \leq \sigma \quad \stackrel{\text{def}}{\iff} \quad [\![\tau]\!] \subseteq [\![\sigma]\!].$$

This approach can be used with arbitrary type constructors

$$\times, \{\ldots\}, \rightarrow, \ldots$$

but requires the addition of intersection ($\wedge$), union ($\vee$) and negation ($\neg$). These boolean combinators must behave set-theoretically:

$$\begin{aligned} [\![\tau \wedge \sigma]\!] &= [\![\tau]\!] \cap [\![\sigma]\!], \\ [\![\tau \vee \sigma]\!] &= [\![\tau]\!] \cup [\![\sigma]\!], \\ [\![\neg\tau]\!] &= \mathcal{V} \setminus [\![\tau]\!], \end{aligned}$$

where $\mathcal{V}$ denotes the set of all values. Notably a model of all terms is not required, neither is its restriction to the well-typed ones. The interpretation of typed values is enough.

A great advantage of semantic subtyping is its completeness: if $\tau \leq \sigma$ does not hold, we can exhibit a value of type τ which does not have type σ. This is theoretically elegant and practically useful, because it allows us to write informative error messages.

The XML processing language $\mathbb{C}$Duce is grounded on semantic subtyping. The core $\mathbb{C}$Duce is a λ-calculus with explicitly typed functions and overloading. It reconciles functions typed with intersection types and overloaded functions. Code usable by all possible input types can be mixed with code requiring specific input types. An example is

$$\mu f^{(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})}.\lambda \mathsf{x}.(\mathsf{y} = \mathsf{x} \in \mathsf{Int})?(\mathsf{y}+1) : \neg \mathsf{y},$$

where y is bound and replaced by x. This function applied to an integer n returns $n+1$ and applied to a boolean b returns $\neg b$. The type of the function is written explicitly. The binder μ allows us to define recursive functions.

In the work [24], an XML language with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives is studied. The key idea for extending semantic subtyping, defined for ground types, to types with type variables, is to make indivisible types "splittable", so that type variables can range over strict subsets of any type. More precisely, *convexity* is the key property. Convexity states that, given a finite set of types, if every assignment makes some of these types empty, then there exists one particular type that is empty for all possible assignments. Convexity imposes that the subtyping relation has a uniform behaviour, thus convexity is a semantic characterisation of "uniformity", which is a feature of parametricity. A subtyping algorithm for convex well-founded models was developed and proved sound and complete. A crucial result is that every model for ground types with infinite denotations is convex. Therefore, to construct a convex model, it just suffices to take any model for ground types and modify straightforwardly the interpretation of basic types so that they have infinite denotations.

In [23] set-theoretic types are combined with polymorphic gradual typing by means of a single typing rule which prescribes where casts must be added. A significant problem is that a direct set-theoretic interpretation of the constructor "?" (at the core of gradual typing) is unsound. The proposed elegant solution uses variables with the constraint of having only all positive or all negative occurrences.

Subtyping has been considered under various aspects. According to [21], inheritance in object-oriented languages can be modelled using subtyping. This work has been a starting point for the formal modelling of the object-oriented paradigm, leading to fundamental discussions, such as the one in [28]. In [63], types are seen as sets of terms and subtyping (there called containment) is just subset inclusion. Therefore this approach can be viewed as an ancestor of semantic subtyping, but with sets of terms instead of sets of values as interpretations of types.

12 Forsythe

Intersection types made their way into imperative languages thanks to John C. Reynolds. This was presented during his LICS invited talk [71], which we briefly recall.

Variables of basic types, like Int, Real, Char, can be used either in reading mode, or in writing mode, or both in reading and writing mode. In the first case they can be typed by Intexp, $\mathsf{Realexp}$, $\mathsf{Charexp}$ and in the second case by Intacc, $\mathsf{Realacc}$, $\mathsf{Characc}$. Intersection types avoid the introduction of a third suffix for the third case, since we can simply use:

$$\mathsf{Intexp} \wedge \mathsf{Intacc},\ \mathsf{Realexp} \wedge \mathsf{Realacc},\ \mathsf{Charexp} \wedge \mathsf{Characc}.$$

In this way, we obtain for free the expected subtyping:

$$\mathsf{Intexp} \wedge \mathsf{Intacc} \leq \mathsf{Intexp},\ \mathsf{Intexp} \wedge \mathsf{Intacc} \leq \mathsf{Intacc} \text{ etc.}$$

The programming language *Forsythe* [72], an evolution of Algol 60, uses intersection types in this and two other ways.

Records are typed by the intersection of label/type pairs, for example the record $\{item : \text{“a”}, price : 5\}$ can have type $(item : \mathsf{Char}) \wedge (price : \mathsf{Int})$. In this way, we get the record subtyping in width.

The λ-abstractions are explicitly decorated by all the types used in various derivations for the same Forsythe term. More precisely, the typing rule is:

$$\frac{\Gamma, \mathsf{x} : \sigma_i \vdash \mathsf{M} : \tau_i}{\Gamma \vdash \lambda \mathsf{x} : \sigma_1 | \cdots | \sigma_n . \mathsf{M} : \sigma_i \to \tau_i}$$

where $1 \leq i \leq n$ and M denotes a Forsythe term. For example, $\lambda \mathsf{x} : \mathsf{Int}|\mathsf{Real}.\mathsf{x}$ has both types $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Real} \to \mathsf{Real}$, hence we can derive $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Real} \to \mathsf{Real})$ for this term using the intersection introduction rule.

Note that the type information on abstracted variables permits the type checking of Forsythe programs.

The work [67] enhances Forsythe with the union type constructor and parametric polymorphism. We present a simple, but effective, example on Church numerals, as defined in [26] (page 28).

The type Nat of natural numbers can be considered as the union of the type Zero, which contains only the Church numeral for zero ($\lambda xy.y$), with the type Pos of the Church numerals for positive numbers, that is $\mathsf{Nat} = \mathsf{Zero} \vee \mathsf{Pos}$. Then, the successor function must map Zero to Pos and Pos to Pos, yielding the type $(\mathsf{Zero} \to \mathsf{Pos}) \wedge (\mathsf{Pos} \to \mathsf{Pos})$. Because Church numeral n applies n-times a function to an argument, this justifies the definitions:

$$\mathsf{Zero} = \forall \phi\psi.(\phi \to \psi) \wedge (\psi \to \psi) \to \phi \to \phi,$$
$$\mathsf{Pos} = \forall \phi\psi.(\phi \to \psi) \wedge (\psi \to \psi) \to \phi \to \psi,$$

which yield a refinement of the standard polymorphic type of Church numerals: $\forall \phi.(\phi \to \phi) \to \phi \to \phi$.

13 A boosted Java

The Java™ Programming Language is a general-purpose, concurrent, strongly typed, class-based object-oriented language [54], and one of the most popular programming languages with types ever. Over the years, Java developers have incorporated sophisticated typing features, notably *generics* in Java 5, and later intersection types appeared as bounds of type variables. Actually, there was a proposal in [20] of introducing in Java 1 intersection types (called there compound types) as parameter types and return types of methods, allowing for programming techniques to support code reuse. Indeed, present Java limits intersection types to be target types of type-casts and bounds of type variables. Nevertheless, thanks to the fact that a generic type variable with an intersection type bound can appear as a parameter type as well as a return type of a method, we can affirm that the extension described in [20] is achieved indirectly.

We observe, however, that this flexibility does not stretch to Java 8 λ-expressions, because a generic type variable cannot be instantiated by the type of a λ-expression. Java λ-expressions are *poly expressions* because they can have various types according to the requirements of the context in which they appear. Each context enforces the *target type* for the λ-expression in that context, but the program does not compile if this target type is unspecified. The target type can be either a *functional interface* (which is an interface with a single abstract method) or an intersection of interfaces that induces a functional interface. In particular, the λ-expression must match the signature of the unique abstract method of its functional interface. By casting a λ-expression to an intersection type, this intersection becomes its target type, and therefore the λ-expression exploits its poly nature, inhabiting all the types of the intersection. In this way, it implements the abstract method and it owns all the default methods present in the interfaces belonging to the intersection. This power is however limited, because if a λ-expression is passed as argument to a method or returned by a method, its target type cannot be an intersection type.

In [36], starting from the core language [15], there is a proposal for a Java where intersection types can appear as types of fields, types of formal parameters and value results of methods, therefore playing the role of target types for λ-expressions anywhere these expressions can be used. Moreover, intersection types are exploited to overcome the restriction of having a unique abstract method in the functional interface, because a λ-expression is able to match multiple signatures of abstract methods. The idea here is that an intersection type expresses multiple, possibly unrelated, properties of one term in a single type.

As shown in Figure 8, auto-application in the λ-calculus can be typed using intersection types. We can adapt this powerful feature from intersection type theory to Java, by allowing any intersection of interfaces to be a functional interface having multiple abstract methods. For example, we can consider the method

```
C auto (Arg&Fun x){return x.mFun(x).mArg(new C( ));}
```

where `C` is any class (without fields for simplicity), and `Arg` and `Fun` are two Java

interfaces with the abstract methods

`C mArg (C y)` and `Arg mFun (Arg z)`,

respectively. Although the method `auto` is greedy with respect to the requirements about its argument, there are many λ-expressions matching the target type `Arg&Fun`, first of all the identity `x->x`.

Other object-oriented languages exploit successfully intersection types, with Scala [31], TypeScript [77], Ceylon [25] and Flow [42] being some remarkable examples. In these programming languages intersection types are used essentially to model forms of multiple inheritance. A foundational work on the usage of intersection types to model multiple inheritance is [27].

14 Conclusion

We hope the reader enjoyed our tale of intersection types, found a thread of thoughts to follow, and maybe even imagined novel ideas for continuing the research on this topic. On our side, we would like to explore the possibility of finding a behavioural characterisation (in the vein of Section 7) of the multiparty session calculus [52], a disciplined π-calculus guaranteeing safety of communication protocols.

Acknowlegments Mariangiola could never imagine when writing with Mario Coppo their first paper on intersection types that it would inspire so many works related to this topic. For this reason, she is very grateful to all authors who contributed to the development and diffusion of intersection types, and she asks for forgiveness to the authors she could not mention here due to space limits.

Viviana feels privileged not only to have been supervised and to work with Mariangiola, but also to have been blessed with her friendship. It is a honour to be her co-author for this paper.

Both Mariangiola and Viviana would like to thank Jeremy Sproston for his suggestions that helped to improve this work.

References

[1] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018.

[2] Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA*, volume 3461 of *LNCS*, pages 39–54, Berlin, 2005. Springer.

[3] Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. Tailoring filter models. In *Types*, volume 3085 of *LNCS*, pages 17–33, Berlin, 2004. Springer.

[4] Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. Intersection types and lambda models. *Theoretical Computer Science*, 355(2):108–126, 2006.

[5] Fabio Alessi, Mariangiola Dezani-Ciancaglini, and Furio Honsell. Inverse limit models as filter models. In *HOR*, pages 3–25, Aachen, 2004. RWTH Aachen. ISSN 0935A3232AIB-2004-03.

[6] Steffen van Bakel. Strict intersection types for the lambda calculus. *ACM Computing Surveys*, 43(3):20:1–20:49, 2011.

[7] Henk Barendregt. *The Lambda Calculus - its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, Amsterdam, 1985.

[8] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[9] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, Cambridge, 2013.

[10] Choukri-Bey Ben-Yelles. *Type-assignment in the lambda-calculus*. PhD thesis, University College of Swansea, 1979.

[11] Erika De Benedetti and Simona Ronchi Della Rocca. Bounding normalization time through intersection types. In *ITRS*, volume 121 of *EPTCS*, pages 48–57, Waterloo, Australia, 2012. Open Publishing Association.

[12] Alexis Bernadet and Stéphane Graham-Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4):1–46, 2013.

[13] Gérard Berry. Stable models of typed lambda-calculi. In *ICALP*, volume 62 of *LNCS*, pages 72–89, Berlin, 1978. Springer.

[14] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Tzu-Chun Chen, Ugo de'Liguoro, and Jakob Rehof. Mixin composition synthesis based on intersection types. In *TLCA*, volume 38 of *LIPIcs*, pages 76–91, Dagstuhl, 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[15] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Logical Methods in Computer Science*, 14(3):1–24, 2018.

[16] Viviana Bono, Betti Venneri, and Lorenzo Bettini. A typed lambda calculus with intersection types. *Theoretical Computer Science*, 398(1-3):95–113, 2008.

[17] Gilad Bracha and William R. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, New York, 1990. ACM.

[18] Kim B. Bruce. *Foundations of object-oriented languages - types and semantics*. MIT Press, Cambridge, Mass., 2002.

[19] Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. Not enough points is enough. In *CSL*, volume 4646 of *LNCS*, pages 298–312, Berlin, 2007. Springer.

[20] Martin Büchi and Wolfgang Weck. Compound types for Java. In *OOPSLA*, pages 362–373, New York, 1998. ACM.

[21] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[22] Sébastien Carlier and Joe B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *ENTCS*, 136:173–202, 2005.

[23] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: a new perspective. *PACMPL*, 3(POPL):16:1–16:32, 2019.

[24] Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, pages 94–106, New York, 2011. ACM.

[25] Ceylon, 1.3.3. https://ceylon-lang.org.

[26] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[27] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[28] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL*, pages 125–135, New York, 1990. ACM.

[29] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[30] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and lambda-calculus semantics. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 535–560. Academic Press, London, UK, 1980.

[31] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *MFCS*, volume 4162 of *LNCS*, pages 1–23, Berlin, 2006. Springer.

[32] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1958.

[33] Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. Intersection types and runtime errors in the pi-calculus. *PACMPL*, 3(POPL):7:1–7:29, 2019.

[34] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, New York, 2000. ACM.

[35] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in higher-order concurrent calculi. *Journal of Logic and Algebraic Programming*, 79(7):550–577, 2010.

[36] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Intersection types in java: back to the future. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*, pages 68–86, Berlin, 2018. Springer.

[37] Mariangiola Dezani-Ciancaglini, Furio Honsell, and Yoko Motohama. Compositional characterisations of λ-terms using intersection types. *Theoretical Computer Science*, 340(3):459–495, 2005.

[38] Boris Düdder, Oliver Garbe, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Using inhabitation in bounded combinatory logic with intersection types for composition synthesis. In *ITRS*, volume 121 of *EPTCS*, pages 18–34, Waterloo, Australia, 2012. Open Publishing Association.

[39] Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Bounded combinatory logic. In *CSL*, volume 16 of *LIPIcs*, pages 243–258, Dagstuhl, 2012. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[40] Andrej Dudenhefner and Jakob Rehof. Intersection type calculi of bounded dimension. In *POPL*, pages 653–665, New York, 2017. ACM.

[41] Andrej Dudenhefner and Jakob Rehof. Typability in bounded dimension. In *LICS*, pages 1–12, New York, 2017. IEEE Computer Society.

[42] Flow, v0.122.0. https://flow.org.

[43] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of ACM*, 55(4):19:1–19:64, 2008.

[44] Jean-Yves Girard. *Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur*. PhD thesis, Université Paris VII, 1972.

[45] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[46] Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117(5):285–306, 1965.

[47] Roger Hindley. The simple semantics for coppe-dezani-sallé types. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 212–226, Berlin, 1982. Springer.

[48] Roger Hindley and Giuseppe Longo. Lambda calculus models and extensionality. *Mathematical Logic Quarterly*, 26(19-21):289–310, 1980.

[49] Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, Cambridge, 1986.

[50] Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theoretical Computer Science*, 411(22-24):2223–2238, 2010.

[51] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, Berlin, 1998. Springer.

[52] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of ACM*, 63(1):9:1–9:67, 2016.

[53] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016.

[54] Java, 14. https://docs.oracle.com/en/java/javase/14/.

[55] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[56] Naoki Kobayashi. Model checking higher-order programs. *Journal of ACM*, 60(3):20:1–20:62, 2013.

[57] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188, Washington, 2009. IEEE Computer Society.

[58] Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Transactions on Programming Languages and Systems*, 32(5):16:1–16:49, 2010.

[59] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 159–191. Academic Press, London, UK, 1980.

[60] Ugo de' Liguoro. Characterizing convergent terms in object calculi via intersection types. In *TLCA*, volume 2044 of *LNCS*, pages 315–328, Berlin, 2001. Springer.

[61] Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la church. *Information and Computation*, 205(9):1371–1386, 2007.

[62] Damiano Mazza, Luc Pellissier, and Pierre Vial. Polyadic approximations, fibrations and intersection types. *PACMPL*, 2(POPL):6:1–6:28, 2018.

[63] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[64] Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*, pages 72:1–72:10, New York, 2014. ACM.

[65] Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Logical semantics for stability. In *MFPS*, volume 249 of *ENTCS*, pages 429–449, Amsterdam, 2009. Elsevier.

[66] Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Mathematical Structures in Computer Science*, 27(5):626–650, 2017.

[67] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[68] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Mass., 2002.

[69] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 561–578. Academic Press, London, UK, 1980.

[70] Jakob Rehof and Pawel Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, volume 6690 of *LNCS*, pages 169–183, Berlin, 2011. Springer.

[71] John C. Reynolds. Conjunctive types and Algol-like languages. In *LICS*, page 119, Washington, 1987. IEEE Computer Society.

[72] John C. Reynolds. Design of the programming language Forsythe. In *Algol-like Languages*, pages 173–233. Birkhäuser, Boston, 1997.

[73] Davide Sangiorgi and David Walker. *The Pi-Calculus - a Theory of Mobile Processes*. Cambridge University Press, Cambridge, 2001.

[74] Dana S. Scott. Continuous lattices. In *Toposes, algebraic geometry and logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136, Berlin, 1972. Springer.

[75] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Mass., 1977.

[76] Kohei Suenaga and Naoki Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *ESOP*, volume 4421 of *LNCS*, pages 490–504, Berlin, 2007. Springer.

[77] TypeScript, 3.8. https://www.typescriptlang.org.

[78] Pawel Urzyczyn. Type reconstruction in $\mathrm{f}_{\mathrm{omega}}$. *Mathematical Structures in Computer Science*, 7(4):329–358, 1997.

[79] Pawel Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

[80] Joe B. Wells. The essence of principal typings. In *ICALP*, volume 2380 of *LNCS*, pages 913–925, Berlin, 2002. Springer.

[81] Joe B. Wells and Christian Haack. Branching types. In *ESOP*, volume 2305 of *LNCS*, pages 115–132, Berlin, 2002. Springer.

[82] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi-calculus. *Information and Computation*, 191(2):145–202, 2004.