Expansion: the Crucial Mechanism for Type Inference with Intersection Types: a Survey and Explanation

Sébastien Carlier J. B. Wells

Heriot-Watt University, http://www.macs.hw.ac.uk/ultra/

Abstract

The operation of *expansion* on typings was introduced at the end of the 1970s by Coppo, Dezani, and Venneri for reasoning about the possible typings of a term when using intersection types. Until recently, it has remained somewhat mysterious and unfamiliar, even though it is essential for carrying out *compositional* type inference. The fundamental idea of expansion is to be able to calculate the effect on the final judgement of a typing derivation of inserting a use of the intersection-introduction typing rule at some (possibly deeply nested) position, without actually needing to build the new derivation. Recently, we have improved on this by introducing *expansion variables* (E-variables), which make the calculation straightforward and understandable. E-variables make it easy to postpone choices of which typing rules to use until later constraint solving gives enough information to allow making a good choice. Expansion can also be done for type constructors other than intersection, such as the ! of Linear Logic, and E-variables make this easy. There are no significant new technical results in this paper; instead this paper surveys and explains the technical results of a quarter of a century of work on expansion.

Key words: intersection types, expansion, type inference

This paper uses colors. Although the colors are not essential and it is readable in black and white, the colors make distinctions that add to the readability of the examples and the paper will read better if printed on a color printer.

1 Background and Motivation

In the context of computer software, *types* are used to express and formally check properties of programs. This can help programming tools like compilers

[1] Partially supported by EC FP5/IST/FET grant IST-2001-33477 "DART", NSF grant 0113193 (ITR), and Sun Microsystems equipment grant EDUD-7826-990410-US.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

in such tasks as detecting programming mistakes, enforcing security properties, and generating smaller, faster, or more predictable code.

For many practical uses of types, it is important to have *type inference* and also *polymorphism*. Type inference allows benefiting from the use of types without imposing the burden that programmers must tediously enter them by hand. Polymorphism, which means reusing the same program fragment at different types, is required for any type system that allows generic code reuse (including abstract data types). Section 1.1 discusses ∀-quantifiers, the most widely used way of obtaining polymorphism, and some of their limitations, while section 1.2 introduces intersection types, a less widely used technique for polymorphism. For both types of polymorphism issues of type inference are discussed.

1.1 ∀-quantification, and its limitations

Most statically typed functional languages use extensions of the well known Hindley-Milner (HM) type system [35], which obtains polymorphism using ∀-*quantification*. Consider the following Standard ML (SML) program fragment, where type annotations (in superscript) have been added to some program points:

$$\texttt{let id}^{\forall \mathsf{a}.(\mathsf{a}\to\mathsf{a})} = \texttt{fn x} \Rightarrow \texttt{x}$$
$$\texttt{in (id}^{\mathsf{int}\to\mathsf{int}}\ \texttt{1}^{\mathsf{int}},\ \texttt{id}^{\mathsf{real}\to\mathsf{real}}\ \texttt{2.0}^{\mathsf{real}}\texttt{)}$$

The *type scheme* $\forall \mathsf{a}.(\mathsf{a} \to \mathsf{a})$ is assigned to `id` after typing its definition, and this type scheme is instantiated to more specific types when `id` is used, here at types $\mathsf{int} \to \mathsf{int}$ and $\mathsf{real} \to \mathsf{real}$.

Type inference using ∀-quantification is very popular, but it has some disadvantages. Quantifiers hide information which could be used to enable compiler optimizations, such as code and data representation specialization, which yield faster and smaller executable programs. In the example given above, if a value of type int can be stored as a 32-bits word, and a value of type real as a 64-bits word, then specialized sequences of machine code could be used for different uses of `id`. However, because the type given to `id` uses ∀-quantification, it yields no information about the different uses of `id`. As a result of using ∀-quantification, most compiler implementations using HM assume a uniform machine representation for all values (e.g., heap pointers). Some code specialization techniques for HM exploit the fact that ∀-quantification is introduced syntactically by `let` to remove some of the burden of uniform representations, but some opportunities for specialization are lost, especially when using higher-order functions.

Systems with only ∀-quantification also generally do not have *principal typings* [45], strongest typings which imply all the others for the same term. Wells [45] proved the absence of principal typings both for HM and for System F [19,37]. In the above example, the program fragment (`id 1`, `id 2.0`) cannot be properly analyzed on its own in HM without first being given a type

for `id`. Because analysis results depend on the context in which they were obtained, they are invalidated when this context changes. This makes it harder to achieve *compositional analysis*, where each program fragment is analyzed using only the analysis results of its immediate subcomponents. Principal typings have other practical applications such as smartest recompilation [24], and accurate type error messages [24,21].

1.2 Intersection types: some advantages, some issues

In contrast, intersection types provide type polymorphism by *listing* usage types [11]. Here is the same example SML program fragment as above, except annotated with intersection types:

$$\begin{array}{l}\texttt{let id}^{(\mathsf{int}\to\mathsf{int})\cap(\mathsf{real}\to\mathsf{real})} = \texttt{fn x} \Rightarrow \texttt{x} \\ \texttt{in (id}^{\mathsf{int}\to\mathsf{int}}\ \texttt{1},\ \texttt{id}^{\mathsf{real}\to\mathsf{real}}\ \texttt{2.0)}\end{array}$$

The original motivation for the name *intersection types* was suggested by their semantics [11]: if types are interpreted by sets of λ-terms, then the intersection type constructor could be interpreted by set intersection. Unlike the product type constructor $\times$ which joins types for possibly different terms[2], the intersection type constructor only joins types assigned to the *same* term. In the example above, the definition of `id` is a single term that can independently be given two different types, $\mathsf{int}\to\mathsf{int}$ and $\mathsf{real}\to\mathsf{real}$, so we may also give `id` the intersection type $(\mathsf{int}\to\mathsf{int})\cap(\mathsf{real}\to\mathsf{real})$. In logical terms, intersection is said to be a *proof-functional* connective [34] (i.e., the meaning of $\cap$ depends on the proofs of the propositions $\cap$ connects) while the usual logical conjunction (to which $\times$ corresponds) is *truth-functional*.

Intersection types make many more terms typable than other approaches. Consider the following program fragment in SML syntax:

```
fun self_apply2 z ⇒ (z z) z;
fun apply f x ⇒ f x;
fun reverse_apply y g ⇒ g y;
fun id w ⇒ w;
(self_apply2 apply not true,
 self_apply2 reverse_apply id false not);
```

This program fragment is rejected by the type system of SML, but according to the dynamic semantics of SML it *safely* computes the result (`false`, `true`). Urzyczyn [42] proved that a λ-term from which this example is derived is not typable in F_ω, considered the most powerful type system with $\forall$-quantifiers [18]. In contrast, the same λ-term is typable in the *rank-3 restriction* of intersection types.

[2] In SML, for example, (`fn x ⇒ x + 1, fn y ⇒ y * 0.5`) has type $(\mathsf{int}\to\mathsf{int})\times(\mathsf{real}\to\mathsf{real})$.

The notion of *rank* was introduced by Leivant as a measure on types that can be used to impose restrictions on type systems. The rank of T is the smallest integer k such that the path between the root of T and any occurrence of $\cap$ or $\forall$ in T goes to the left of a $\rightarrow$ less than k times. Other measures are possible (e.g., the depth of types), but the rank is more useful because it is related to the complexity of evaluation and of type inference [29].

The use of types is not necessarily just to prevent programs from causing run time errors, but can support many kinds of program analysis usable for justifying compiler optimizations in order to produce better machine code. When types are used to carry properties of programs, type polymorphism enables *polyvariant analysis*. Intersection type systems are particularly suitable for polyvariant type-based analysis. Some of the properties they help analyze are *flow* [2], *strictness* [23], *dead code* [15,16], and *totality* [8]. Thus, in addition to rejecting fewer safe programs, intersection types seem to have the potential to be a general, flexible framework for many useful program analyses. Intersection types also usually have principal typings, thereby enabling compositional analysis.

Although intersection types seem possibly better suited than $\forall$-quantifiers for compiler optimizations and compositional analysis of computer programs, they have not been widely adopted. Furthermore, when intersection types have been used, their full power has not been exploited; it is mainly the rank-2 restrictions of intersection types that have been used in type inference algorithms for practical languages [24,13,14].

We believe a large part of the reason for not using intersection types (and working only with rank-2 intersection types when they are used) has been the difficulty of understanding the notion of *expansion*, which is crucial for type inference for intersection types beyond the rank-2 restriction. To overcome this problem, this paper aims to be a gentle introduction to intersection types and the notion of expansion and related mechanisms.

Expansion is presented in this paper in the form needed for computing principal typings for the full (i.e., not rank-restricted) system of intersection types. However, computing these principal typings is as expensive as evaluation, for the simple reason that principal typings for a term in the full system express all of the information in the term's β-normal form [11,41]. This is obviously impractical, and readers might then legitimately wonder why they should care about the explanations that this paper provides.

In fact, there is no reason why one *must* use the full power of intersection types; for example, one can choose to use principal typings of the rank-k restriction. In the long run, if one wants to use intersection types, it seems best to view them as a flexible framework for typing with a choice of a wide variety of different levels of precision. However, before attempting to do this, it is extremely helpful to understand type inference for the full system, because it is simpler. Hence, it can be beneficial to understand the explanations given in this paper.

2 Intersection types

We now define an intersection type system that contains sufficient features to support discussing expansion. *Types* (ranged over by T) are defined as follows:

$$T ::= a \mid T_1 \to T_2 \mid T_1 \cap T_2 \mid \omega$$

Here, a ranges over an infinite set of *type variables (T-variables)*. We use lowercase Roman letters as metavariables over T-variables, generally those from the beginning of the alphabet like a, b, and c. We adopt the convention that distinct metavariables stand for distinct variables within any single example. To resolve ambiguities in the absence of parentheses, we define $\cap$ to have higher precedence than $\to$, so that for example $T_1 \cap T_2 \to T_3 = (T_1 \cap T_2) \to T_3$.

We quotient types by taking $\cap$ to be associative ($T_1 \cap (T_2 \cap T_3) = (T_1 \cap T_2) \cap T_3$), commutative ($T_1 \cap T_2 = T_2 \cap T_1$) and to have ω as a neutral ($\omega \cap T = T$).

Type environments, ranged over by A, are written in the form $(x_1 : T_1, \ldots, x_n : T_n)$ with all x_i distinct. As a special case, $()$ denotes the empty environment. If $A = (x_1 : T_1, \ldots, x_n : T_n)$, we let $A(x_i) = T_i$ for all $i \in \{1, \ldots, n\}$, and $A(y) = \omega$ for every y not mentioned by A. The notation $A, x : T$ stands for the new type environment A' such that $A'(x) = T$ and $A'(y) = A(y)$ if $y \neq x$. The notation $A_1 \cap A_2$ stands for pointwise application of the intersection type constructor, i.e., $(A_1 \cap A_2)(x) = A_1(x) \cap A_2(x)$ for every x.

We use an almost standard syntax for λ-terms:

$$M ::= x \mid \lambda x.\, M \mid M_1 \mathbin{@} M_2$$

Here, x ranges over an infinite set of *λ-term variables*. We use lowercase Roman letters as metavariables over term variables, generally those from the end of the alphabet like x, y, and z. As usual, we identify α-equivalent λ-terms. We write the "@" in $M_1 \mathbin{@} M_2$ instead of just writing $M_1 M_2$ so that there will be a better correspondence with the tree diagrams we will write.

In addition to the usual use of pure (type-free) λ-terms as proof terms in typing judgements when using intersection types, we add an additional kind of proof terms which we call *skeletons*, ranged over by Q[3]. The syntax of skeletons is this:

$$Q ::= x^{:T} \mid \lambda x.\, Q \mid Q_1 \mathbin{@} Q_2 \mid Q_1 \cap Q_2$$

Typing judgements are of the following shape:

$$Q \triangleright M : \langle A \vdash T \rangle$$

Such a judgement should be read as stating that "the skeleton Q is a proof that the term M can be assigned the typing $\langle A \vdash T \rangle$". Our skeletons are

[3] We use the metavariable Q because we already use S for substitutions and because Q is the second letter in "squelette", the French word for skeleton.

designed so that they have just enough information in them to completely reproduce the details of the proof of an assignment of a typing to a term.

The typing rules of the system are as follows. We begin with the rules common to almost all λ-calculus type systems:

$$\frac{}{x^{:T} \triangleright x : \langle (x : T) \vdash T \rangle}\ \mathsf{var} \qquad \text{(variable)}$$

$$\frac{Q \triangleright M : \langle A, x : T_1 \vdash T_2 \rangle}{\lambda x.\, Q \triangleright \lambda x.\, M : \langle A \vdash T_1 \to T_2 \rangle}\ \mathsf{abs} \qquad \text{(abstraction)}$$

$$\frac{Q_1 \triangleright M_1 : \langle A_1 \vdash T_1 \to T_2 \rangle \quad Q_2 \triangleright M_2 : \langle A_2 \vdash T_1 \rangle}{Q_1 \mathbin{@} Q_2 \triangleright M_1 \mathbin{@} M_2 : \langle A_1 \cap A_2 \vdash T_2 \rangle}\ \mathsf{app} \qquad \text{(application)}$$

The abs rule is conventional. Note that the var rule has a type environment that must assume type ω for every term variable except x. Note that the rule app joins the type environments of the two premises to form the type environment $A_1 \cap A_2$ in the conclusion. These points will be discussed at various places later.

Here is the significant additional rule, intersection introduction:

$$\frac{Q_1 \triangleright M : \langle A_1 \vdash T_1 \rangle \quad Q_2 \triangleright M : \langle A_2 \vdash T_2 \rangle}{Q_1 \cap Q_2 \triangleright M : \langle A_1 \cap A_2 \vdash T_1 \cap T_2 \rangle}\ \cap \qquad (\cap \text{ introduction})$$

Note that this rule has two premises which assign typings to the *same* term M; the skeletons Q_1 and Q_2 can differ, so the two subderivations can have different structures and assign quite different typings.

Although some type systems have a rule for eliminating intersection types, our example system does not need one because this task is handled implicitly by the way type environments in premises are joined in conclusions in the multiple-premise typing rules ($\cap$ and app). For example, assuming we allow types int and real, if a skeleton Q contains two free occurrences of `id` at types $\mathsf{int} \to \mathsf{int}$ and $\mathsf{real} \to \mathsf{real}$, and Q has no other free variable occurrences, then the type environment in the typing derived by Q is $(\mathtt{id} : (\mathsf{int} \to \mathsf{int}) \cap (\mathsf{real} \to \mathsf{real}))$.

Note that type environments contain all and only necessary assumptions. Systems with this property are called *relevant* in the literature, due to the correspondence with relevant logic [17]. In fact, this system has the further property that types are *linear*; this is further discussed in section 5.2.

The system presented in this section is essentially a streamlined version of the original system of intersection types by Coppo, Dezani and Venneri [11], which we call the CDV system. In [11] and also here, $\cap$ is associative and commutative (AC) but not idempotent (i.e., $T \cap T \neq T$ unless $T = \omega$) [4], and type environments contain only assumptions for variables that are actually used (the systems are relevant). The main difference (unimportant for the

[4] This is after translating to modern notation the types of [11], where intersection types are written in the form $[T_1, \ldots, T_n]$.

examples in this section, but useful for later sections) is that, for simplicity, we allow $\cap$ and ω to the right of $\to$.

3 Expansion

This section first demonstrates in subsection 3.1 the importance of expansion in the context of an intersection type system, then explains in subsection 3.2 how expansion works as it was designed historically, and then presents in subsection 3.3 the modern way of doing expansion through expansion variables.

3.1 Why we need expansion

3.1.1 A problematic type inference example

Consider typing this example λ-term:

$$M = \underbrace{(\lambda x.\, x \mathbin{@} (\lambda y.\, y \mathbin{@} z))}_{M_1} \mathbin{@} \underbrace{(\lambda f.\, \lambda x.\, f \mathbin{@} (f \mathbin{@} x))}_{M_2}$$

In the intersection type system considered, term M is typable because it has a normal form [11], so we should be able to build a typing derivation for it.

Subterms M_1 and M_2 are in normal form and we can easily obtain their *principal typings* using the algorithm of Coppo, Dezani and Venneri [11]:

$$Q_1 = \lambda x.\, x^{:((a\to b)\to b)\to c} \mathbin{@} (\lambda y.\, y^{:a\to b} \mathbin{@} z^{:a})$$

$$Q_1 \triangleright M_1 : \langle (z : a) \vdash T_1 \to c \rangle \quad \text{with } T_1 = ((a \to b) \to b) \to c$$

$$Q_2 = \lambda f.\, \lambda x.\, f^{:e\to f} \mathbin{@} (f^{:d\to e} \mathbin{@} x^{:d})$$

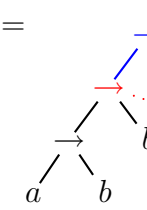
$$Q_2 \triangleright M_2 : \langle () \vdash T_2 \rangle \quad \text{with } T_2 = ((e \to f) \cap (d \to e)) \to (d \to f)$$

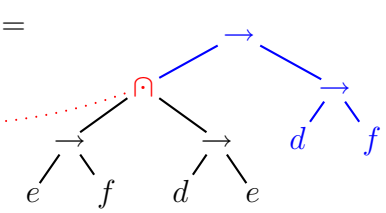
Note that we cannot use the application typing rule directly to join Q_1 and Q_2 because T_1 (the domain of the result type of Q_1) is not equal to T_2 (the result type of Q_2). So we must somehow "unify" T_1 and T_2.

Can we do this merely by substitution, replacing type variables by types? Lining up matching bits and drawing the two types as trees makes things easier to follow:

$$T_1 = ((a\to b) \to \quad b \quad) \to \quad c$$

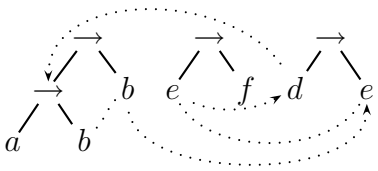
$$T_2 = ((e\to f) \cap (d\to e)) \to (d \to f)$$

$T_1 =$ 

$T_2 =$ 

The first problem we encounter is that there is a clash between type constructors $\to$ and $\cap$, indicated by the dotted line.

To solve the example, we could try to make the intersection go away by placing ourselves in a system with intersection idempotence ($T \cap T = T$), allowing us to unify the two branches of the intersection, $e \to f$ and $d \to e$, which could then be unified with $(a \to b) \to b$. This would require unifying these three subtrees:

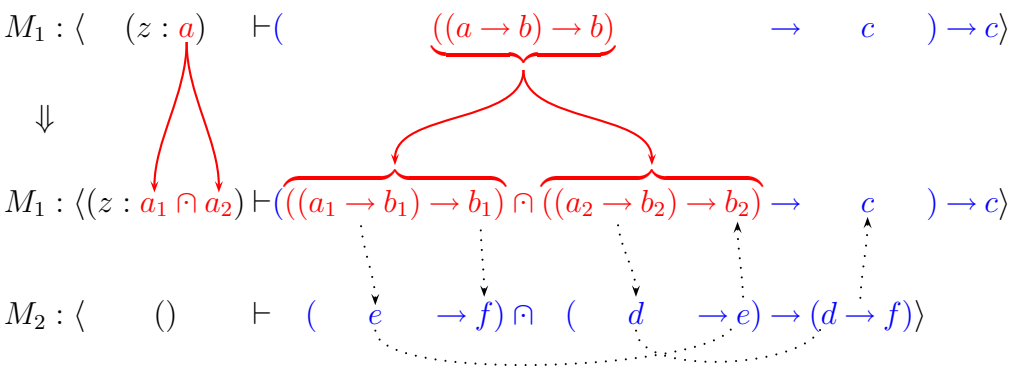


Dotted arrows depict some of the required variable substitutions, while dotted lines show connections between occurrences of the same variable. Note that there is a cycle; the equation $b = e = d = a \to b$ cannot be solved without recursive types. Although recursive types would solve this example, they are not such a good solution, because they would not provide a principal typing. Historically there have also been other objections to recursive types, such as the extra difficulty of automatically explaining type errors discovered during type inference when recursive types are allowed. Thus, simply making intersection idempotent would not solve the problem in a fully satisfactory way.

We designed the example in this section to be untypable when using only simple types, to demonstrate that substitution is not enough to obtain adequate typings for M_1 and M_2 from their principal typings. The solution requires using intersection types, which are introduced by another operation.

3.1.2 Expansion to the rescue!

Historically, in intersection type systems the solution has been to do *expansion* [11] on the typing of M_1:



Solid arrows denote the transformation performed by expansion in this case. The rules for expansion will be discussed later in section 3.2. After doing expansion, we can unify types as required by applying this substitution,

denoted above by dotted arrows:

$$S = (e := a_1 \to b_1,\ f := b_1,\ d := a_2 \to a_1 \to b_1,$$
$$b_2 := a_1 \to b_1,\ c := (a_2 \to a_1 \to b_1) \to b_1\ )$$

Here are the new typings for M_1 and M_2:

$$M_1 : \langle (z : a_1 \cap a_2) \vdash (T_3 \to T_4) \to T_4 \rangle \qquad M_2 : \langle () \vdash T_3 \to T_4 \rangle$$
$$\text{where} \quad T_3 = ((a_1 \to b_1) \to b_1) \cap ((a_2 \to a_1 \to b_1) \to a_1 \to b_1)$$
$$\text{and} \quad T_4 = (a_2 \to a_1 \to b_1) \to b_1$$

Finally, the example can be completed by using the application typing rule to type $M_1 \mathbin{@} M_2$. Thus, expansion solves the problem. But what is expansion?

3.2 What historical expansion is

Coppo, Dezani, and Venneri [11] showed that intersection types support principal typings. Their motivation for studying intersection types was to be able to type more terms than with Curry's system of simple types, and to get preservation of types under β-conversion. As our example in section 3.1 points out, they noticed that, unlike what is the case with simple types, substitution (replacing type variables with types) and weakening (adding type assumptions to a type environment) are not enough to obtain all typings of a term from a principal typings for the same term, and therefore introduced the *expansion* operation

Instead of using the intersection type constructor ($\cap$), they used *sequences* of types (identified modulo reordering of the componentssequences actually behave like multi-sets) written between square brackets and only allowed to occur to the left of arrows. For example, they write the type $((e \to f) \cap (d \to e)) \to (d \to f)$ in the form $[e \to f, d \to e] \to (d \to f)$. We will use the modern notation instead.

Their definition of expansion relies on the notion of *nucleus* of a typing. Informally, a nucleus is delimited by underlining some of the types (result type or types assumed for free term variables) in a typing or components of sequences in those types, such that no type variable occurs both in and out of the nucleus.

The original definition of nucleus relied on an implicit formalism for identifying occurrences in types, although it is not obvious how to achieve this when using an associative and commutative intersection type constructor (or, as was the case, sequences of types identified modulo reordering of components). Although the original definition of nucleus was somewhat informal for this reason and sometimes difficult to understand, we will now introduce a new formal yet accessible way of notating a nucleus. We now define *marked*

types, ranged over by U:

$$U ::= a \mid \underline{a} \mid U \to U \mid T \underline{\to} T \mid U \cap U \mid \omega$$

Marked types are like normal types except that some occurrences of type variables and $\to$ are marked by underlining. Marks are not allowed to nest by the grammar; if an occurrence of $\to$ is marked, then all types underneath it must be mark-free. We allow underlining a whole expression as shorthand for underlining its top-level constructor (we do not use it, but for this purpose $\underline{T_1 \cap T_2}$ would be regarded as $\underline{T_1} \cap \underline{T_2}$, and underlining ω would be disregarded). We let B range over environments of marked types. We define a function $|\cdot|$ on marked types that erases underlining, i.e., $|\underline{a}| = a$ and $|T_1 \underline{\to} T_2| = T_1 \to T_2$.

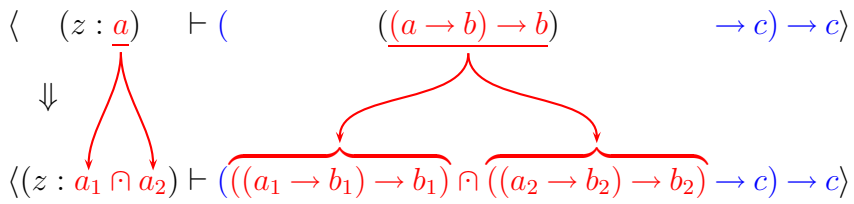
A nucleus is formally delimited by underlining: a nucleus of $\langle A \vdash T \rangle$ is of the form $\langle B \vdash U \rangle$, differing from $\langle A \vdash T \rangle$ only by the addition of marks in certain positions, with the condition that if a type variable a is marked or occurs underneath a marked $\to$, then all other occurrences of a must also be. There are other rules for valid nuclei which we will not discuss, partly because the reasons for the other rules are obscure and partly because we feel the modern approach presented below in section 3.3 is clearer and more important for the reader to understand.

We can now explain what expansion does. Let a *renaming* function be a total injective function from T-variables to T-variables. A renaming function r is applied to types like a substitution, so that $r(T)$ is the new type that results from T by replacing every T-variable a by $r(a)$. An expansion operation takes as input a nucleus $\langle B \vdash U \rangle$ and renaming functions r_1, ..., r_k where $k \geq 1$, the ranges of r_i and r_j are disjoint when $1 \leq i < j \leq k$, and the range of r_i is disjoint from the T-variables occurring in $\langle B \vdash U \rangle$ when $1 \leq i \leq k$. The operation $\mathsf{expand}(\langle B \vdash U \rangle, r_1, \ldots, r_k)$ replaces each underlined U' in $\langle B \vdash U \rangle$ by $r_1(|U'|) \cap \ldots \cap r_k(|U'|)$, producing a new typing.

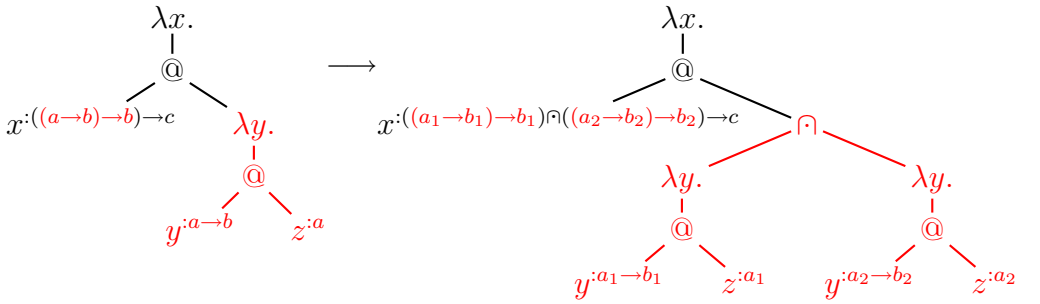
Starting with the principal typing of a term, expansion allows obtaining typings that can not be obtained just by applying a substitution. This is because expansion simulates on a typing the effect of inserting uses of the intersection-introduction typing rule into a derivation of that typing, but without needing to actually construct a new derivation in the process of calculating the typing in the new final judgement. Note that when looking at the effect of substitution application on a typing derivation, only types (which are at the leaves of a typing derivation) are altered, whereas the effect of expansion is to add uses of intersection introduction at internal nodes in a typing derivation. The next example illustrates expansion.

Example 3.1 The following valid nucleus (denoted by underlining) is used

to perform the expansion in the example at the beginning of section 3.1.2:



Here is the corresponding transformation on a simple derivation (skeleton) of the original typing, where ∩ marks a use of the intersection-introduction typing rule:



□

In its original definition [11], expansion was only applied to (expansions of) principal typings, and was used to prove that all and only the derivable typings for a term can be obtained from its principal typing by applying first a sequence of expansions, and then a sequence of substitutions. Later work [39] showed that expansion can safely be applied to typings that are not necessarily principal, i.e., that expansion applications, substitution applications, and uses of subtyping can be interleaved.

This concludes the presentation of the original notion of expansion devised by Coppo, Dezani and Venneri. Similar operations developed later are discussed in section 6.1.

3.3 Modern expansion with expansion variables

Expansion variables (E-variables) were first used as type constructors by Kfoury and Wells in System I [31,33] to simplify reasoning about and implementing the operation of expansion. The most modern system with E-variables is System E [5], and we will use it as the context for explaining E-variables, with some simplifications to ease presentation.

System E can be viewed as an extension of the system presented at the beginning of section 3. One of the changes is to extend types with a case for *E-variable application*, written simply $e\,T$, where e is an E-variable. We extend the precedence convention to have E-variable application bind tighter than ∩, so that for example $e\,T_1 \cap T_2 \to T_3 = ((e\,T_1) \cap T_2) \to T_3$. E-variable

application is extended to type environments ($e\,A$), where it just applies the E-variable to each type in the environment.

In a typing, the multiple occurrences of some E-variable simply delimit a (generalized notion of) nucleus, a set of positions that can be affected by a single expansion operation.

Marked types, introduced in section 3.1, correspond to a very weak form of E-variables, where only a single E-variable is allowed; if e denotes this unique E-variable, then $T \underline{\to} T$ corresponds to $e\,(T \to T)$, and $\underline{a}$ corresponds to $e\,a$. The next example illustrates this.

Example 3.2 Here is a typing of M_1 from the example shown in section 3.1, with an expansion variable:

$$M_1 : \langle (z : e\,a) \vdash (e\,((a \to b) \to b) \to c) \to c \rangle$$

As can be expected, e is applied to all (and only) the underlined types of the nucleus shown in example 3.1. The expansion shown in section 3.1.2 would be obtained by substituting for e an *expansion term* $E = (a := a_1, b := b_1) \pitchfork (a := a_2, b := b_2)$. The meaning of the pieces of E are explained throughout the rest of this section and the result of substituting E for e will be shown in example 3.6. □

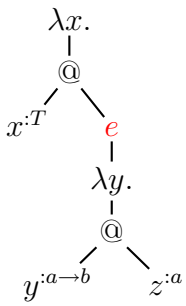
3.3.1 E-variable application typing rule

In addition to types and type environments, E-variable application is also defined for skeletons ($e\,Q$), with the following corresponding typing rule:

$$\frac{Q \triangleright M : \langle A \vdash T \rangle}{e\,Q \triangleright M : \langle e\,A \vdash e\,T \rangle}$$

Example 3.3 Here is a skeleton deriving the typing in example 3.2:

$$Q = \lambda x.\, x^{:T} \mathbin{@} e\,(\lambda y.\, y^{:a \to b} \mathbin{@} z^{:a}) =$$



where $T = e\,((a \to b) \to b) \to c$ □

E-variable application in a skeleton acts as a *placeholder* for unknown uses of other typing rules, such as intersection introduction. Filling this placeholder is done via substitution, by replacing the E-variable with an expansion term. We call expansion terms just expansions.

Expansions, ranged over by E, are pieces of syntax standing for some number of uses of typing rules that act uniformly on every type in a judgement and do not change the judgement's term. Because E-variable application itself satisfies these criteria, it is included as a case of expansion (in addition to already being a case of types and skeletons):

$$E ::= e\,E \mid \cdots$$

Sections 3.3.2 through 5 introduce other cases of expansion in an incremental fashion, and also incrementally define *substitution application* and *expansion application*.

3.3.2 Substitution basics

Substitutions, ranged over by S, replace T-variables with types, and E-variables with expansions. Details are given incrementally throughout this section.

We write $[S]\,X$ for the application of substitution S to an entity X (such as a T-variable, E-variable, type, skeleton, or an expansion). Substitutions apply to type environments pointwise, i.e., $[S]\,A$ is the environment such that $([S]\,A)(x) = [S]\,A(x)$. We reuse the notation and write $[E]\,X$ for the application of an expansion E to an entity X. Expansion also applies to environments pointwise. The definitions of $[S]\,X$ and $[E]\,X$ consist only of very simple cases, but to detail each case we will give the definitions in an incremental fashion. Note also that these definitions can be directly translated into programming languages like SML and Haskell.

The key case of substitution application is for E-variable application:

$$[S]\,(e\,X) = [[S]\,e]\,X$$

In words, when a substitution S is applied to $e\,X$, we first apply S to e to obtain an expansion E, which is then applied to X. Hence, replacing an E-variable e with an expansion E makes e go away unless it is re-introduced by E. The following rule of expansion application allows this:

$$[e\,E]\,X = e\,[E]\,X$$

Thus, replacing e with $e\,E$ has the same effect as applying E *underneath* e.

Substitutions are given syntactically, as comma-separated lists of assignments, terminated by the symbol $\boxdot$ (which we sometimes omit for brevity):

$$\phi \in \mathsf{Assignment} ::= a := T \mid e := E$$

$$S \in \mathsf{Substitution} ::= \boxdot \mid \phi, S$$

We write $\boxdot$ for the identity substitution, which is also the *null expansion*:

$$E ::= \cdots \mid \boxdot \mid \cdots$$

We initially only consider the identity substitution $\boxdot$ as a case of expansion, but as we will see later, it turns out that $\boxdot$ can be generalized to arbitrary substitutions. The effect of $\boxdot$ on T- and E-variables is simply:

$$[\boxdot]\, a = a \qquad\qquad [\boxdot]\, e = e\, \boxdot$$

Note that $[\boxdot]\, e\, X = [[\boxdot]\, e]\, X = [e\, \boxdot]\, X = e\, [\boxdot]\, X$, so $\boxdot$ indeed leaves variables unchanged, as can be expected of the identity substitution.

Let v range over $\mathsf{T\text{-}Variable} \cup \mathsf{E\text{-}Variable}$ and let Φ range over $\mathsf{Type} \cup \mathsf{Expansion}$. The application of substitutions to T- and E-variables is completed thus:

$$[v := \Phi, S]\, v = \Phi$$

$$[v := \Phi, S]\, v' = [S]\, v' \quad \text{if } v \neq v'$$

The syntax of assignments guarantees that $[S]\, a = T$ for some T and $[S]\, e = E$ for some E.

To allow substitutions to be applied to types and skeletons, we use these trivial recursive descent rules:

$$[S]\, (T_1 \to T_2) = [S]\, T_1 \to [S]\, T_2 \qquad [S]\, x^{:T} = x^{:[S]\, T}$$

$$[S]\, (X_1 \cap X_2) = [S]\, X_1 \cap [S]\, X_2 \qquad [S]\, \lambda x.\, Q = \lambda x.\, [S]\, Q$$

$$[S]\, \omega = \omega \qquad [S]\, (Q_1 \mathbin{@} Q_2) = [S]\, Q_1 \mathbin{@} [S]\, Q_2$$

It is easy to show that $[\boxdot]\, T = T$ and $[\boxdot]\, Q = Q$ for all T and Q.

Example 3.4 The following equalities hold:

$$[e := \boxdot]\, (x^{:e\, a \to b} \mathbin{@} e\, y^{:a}) = x^{:a \to b} \mathbin{@} y^{:a}$$

$$[e_1 := \boxdot]\, (e_1\, e_2\, a) = [[e_1 := \boxdot]\, e_1]\, (e_2\, a) = [\boxdot]\, (e_2\, a) = e_2\, a$$

$$[e_2 := \boxdot]\, (e_1\, e_2\, a) = [[e_2 := \boxdot]\, e_1]\, (e_2\, a) = [\boxdot]\, (e_1\, e_2\, a) = e_1\, e_2\, a$$

$$[a := T]\, (e\, a) = [[a := T]\, e]\, a = [[\boxdot]\, e]\, a = [e\, \boxdot]\, a = e\, a$$

The last two examples might be surprising, as one might have expected results to be respectively $e_1\, a$ (instead of $e_1\, e_2\, a$) and $e\, T$ (instead of $e\, a$). In fact, each E-variable establishes a namespace and a inside e is not connected to a outside e. □

3.3.3 The intersection expansion

In addition to the E-variable application and the null expansions, System E also has an *intersection* expansion:

$$E ::= \cdots \mid E_1 \cap E_2 \mid \cdots$$

The intersection expansion corresponds to using the intersection typing rule.

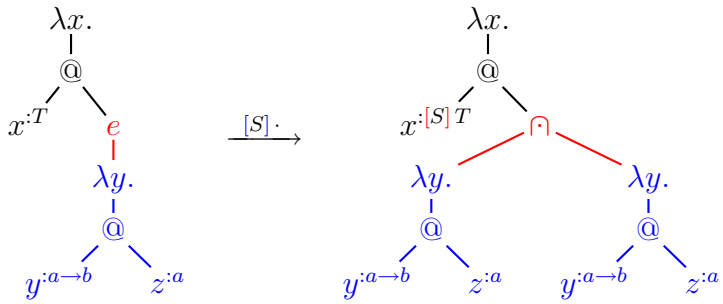
Expansion application of an intersection expansion is simply:

$$[E_1 \cap E_2]\, X = [E_1]\, X \cap [E_2]\, X$$

Note that unlike definitions of expansion not using E-variables, the two copies of X are not renamed by this case of expansion. As we show in section 3.3.4, this can instead be achieved by E_1 and E_2. Removing the built-in renaming of expansion makes reasoning much easier.

Using just the simple cases of substitution and expansion application we have given so far, we can now present a complete example.

Example 3.5 Here is the effect of applying $S = (e := (\boxdot \cap \boxdot))$ to the skeleton Q from example 3.3:



Here $T = e\,((a \to b) \to b) \to c$ and $[S]\,T = ((a \to b) \to b) \cap ((a \to b) \to b) \to c$. We can also apply the same operation directly to the typing of Q:

$$\begin{aligned} &\langle (z : e\,a) \vdash (e\,((a \to b) \to b) \to c) \to c \rangle \\ \xrightarrow{[S]\cdot}\ &\langle (z : a \cap a) \vdash (((a \to b) \to b) \cap ((a \to b) \to b) \to c) \to c \rangle \end{aligned}$$ □

Note that the result obtained in example 3.5 is not very useful, because adding intersections just made identical copies. How do we make these copies different?

3.3.4 The substitution expansion

This "rule" is admissible:

$$\frac{Q \triangleright M : \langle A \vdash T \rangle}{[S]\,Q \triangleright M : \langle [S]\,A \vdash [S]\,T \rangle}$$

In words, given a substitution S and a skeleton Q deriving $\langle A \vdash T \rangle$ for M, the skeleton $Q' = [S]\,Q$ derives the typing $\langle [S]\,A \vdash [S]\,T \rangle$ for M. This is the key idea of expansion: it is an operation defined on typings that corresponds to manipulating typing derivations.

Because substitution application has a corresponding (admissible [5]) typing rule which acts uniformly on every judgement component and does not change

[5] In fact, in the original System E paper [5], there is an explicit rule for substitution

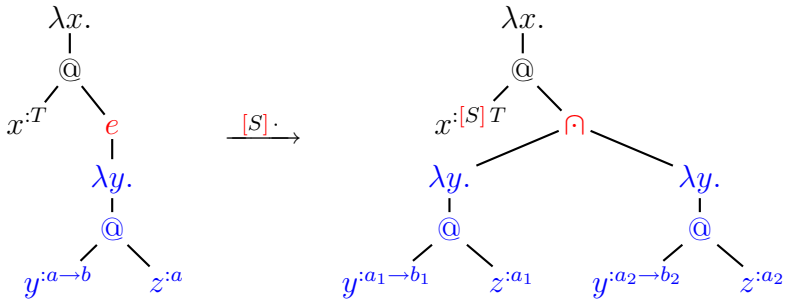
the term, it is natural to include substitution as a case of expansion:

$$E ::= \cdots \mid S \mid \cdots$$

This is a generalization of earlier $\boxdot$, the identity substitution.

We can now finally show how the example used in section 3.1.2 to demonstrate the need for expansion is solved with E-variables.

Example 3.6 Let $S_1 = (a := a_1, b := b_1)$ and $S_2 = (a := a_2, b := b_2)$. The substitution $S = (e := (S_1 \cap S_2))$ thus has distinct substitutions for each of the two copies introduced by the intersection expansion given for e. Here is the effect S when it is applied to the skeleton Q from example 3.3:



Here T is as in example 3.5 and $[S]\,T = ((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c$. On the typing (given originally in example 3.2) obtained from the skeleton Q, the substitution S has exactly the effect required to solve our original motivating example from section 3.1:

$$\begin{aligned} &\langle (z : e\,a) \vdash (e\,((a \to b) \to b) \to c) \to c \rangle \\ \xrightarrow{[S]\,\cdot}\ &\langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle \end{aligned}$$ □

We have shown how the effects of historical expansion can be obtained with expansion variables in a way that is more robust, easier to understand, and straightforward to implement. Sections 4 and 5 discuss extensions of the theory of expansion variables beyond what was done with historical expansion.

4 The omega expansion

Some intersection type systems have a type written ω, which was originally added by Sallé [40] to type systems developed by Coppo and Dezani [9,10]. The type ω is given a case in skeletons and a corresponding typing rule:

$$Q ::= \cdots \mid \omega^M \qquad\qquad \frac{}{\omega^M \triangleright M : \langle () \vdash \omega \rangle}\ \omega$$

application, but it is there for a completely different purpose and is not needed otherwise. The discussion of this section assumes there is no such rule.

The skeleton ω^M needs to mention M to uniquely determine the typing derivation it corresponds to. The typing $\langle () \vdash \omega \rangle$, since it can be assigned to any term by the ω typing rule, can be regarded as the most uninformative typing.

If intersection introduction were generalized to have a variable number of premises, then the ω typing rule would be an instance with 0 premises; similarly, if the intersection type constructor were generalized to be of variable arity, then ω would be its 0-ary version. So, the ω type may intuitively be thought of as the neutral of the intersection type constructor ($\omega \cap T = T$), though this is not technically true in all intersection type systems.

With E-variables, ω is straightforward to add as a case of expansion, as first done by Carlier [4]:

$$E ::= \cdots \mid \omega \mid \cdots$$

Substitution application is trivial. Expansion application just needs some care on skeletons because we have to keep track of the term:

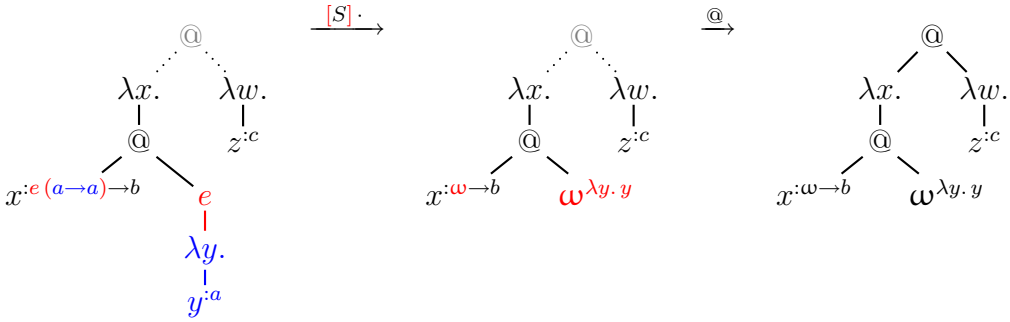
on skeletons:	$[S]\,\omega^M = \omega^M$	$[\omega]\,Q = \omega^{\mathsf{term}(Q)}$
on other sorts:	$[S]\,\omega \quad = \omega$	$[\omega]\,X = \omega$

We define $M = \mathsf{term}(Q)$ iff $Q \triangleright M : \langle A \vdash T \rangle$ is derivable.

Example 4.1 Consider typing this example λ-term:

$$M = \underbrace{(\lambda x.\, x \mathbin{@} (\lambda y.\, y))}_{M_1} \mathbin{@} \underbrace{(\lambda w.\, z)}_{M_2}$$

Suppose we have build independently the following two typing derivations for M_1 and M_2, and we want to join them using the application typing rule to build a typing derivation for M. This requires applying the substitution $S = (e := \omega)$:



We indicate here by $\xrightarrow{@}$ that we can then legally use the application typing rule to combine the two skeletons. □

5 Other significant issues

This section discusses issues related to expansion and expansion variables. Section 5.1 discusses the composition of substitution and expansion. Section 5.2 discusses a generalization of expansion to the ! type constructor, which distinguishes between linear and non-linear types.

5.1 Composition of substitutions and expansions

Although expansion was introduced as an operation complementing substitution in the context of intersection types, and substitution usually supports composition, a good theory of their composition took time to develop. In the intersection type literature using expansion but not expansion variables (e.g., [11,39,38,44]), the problem of composition is not addressed; instead, *chains* of individual operations are constructed, and whenever a chain is applied all its operations have to be performed in sequence. This is somewhat unsatisfactory because (1) substitutions alone usually compose, and it is frustrating that adding expansion breaks this property, and (2) every operation in a chain is applied to the result of the previous operation; since most expansions and substitutions increase the size of types, composing the operations might save work if a chain is to be applied to many types.

In the first system with expansion variables, System I [31,33], composition of substitutions (replacing T-variables by types and E-variables by expansions, which unlike in System E do *not* include substitutions) could only be done in a weak way. In System I, the composition of two arbitrary substitutions can not always be expressed as a substitution; when it can, a notion of *safe composition* is needed to compute it, and this operation is both context-dependent (it requires more information than just substitutions), and very difficult to understand and implement correctly.

In contrast, to compose substitutions in System E, we just need to add these cases to the definition of substitution application:

$$\begin{aligned}
[S]\,\boxdot &= S \\
[S]\,(a := T, S') &= (a := [S]\,T, [S]\,S') \\
[S]\,(e := E, S') &= (e := [S]\,E, [S]\,S')
\end{aligned}$$

Note that these cases simultaneously complete the definitions of (1) substitution application to substitutions ($[S]\,S'$ for any S, S'), (2) substitution application to expansions ($[S]\,E$ for any S, E; all cases except $[S]\,S'$ were given earlier), and (3) expansion application to expansions ($[E]\,E'$ for any E, E'; all cases except $[S]\,E'$ were given earlier).

It is proved in [5] that this equality holds:

$$[E_2]\,[E_1]\,X = [[E_2]\,E_1]\,X$$

Thus, composition of expansions is merely $[E_2]\,E_1$, which we sometimes write as $E_1; E_2$, and composition of substitutions is the special case where $E_1 = S_1$ and $E_2 = S_2$.

This simplicity comes from the principled way in which expansion is done in System E, namely that each case of expansion terms corresponds exactly to a typing rule that can be spliced in at any point. In System I, composition of substitutions is a complex operation because substitution application, though an admissible typing rule, is not a case of expansion, and instead a complicated notion of renaming is built into the machinery for replacing E-variables by expansion terms.

5.2 Linearity and non-linearity

The semantics of intersection types depend on whether they are *linear*. Whether this holds depends on such things as whether the intersection type constructor is idempotent ($T \cap T = T$), whether weakening is allowed, and whether contraction is allowed. Weakening can be allowed in a general way via subtyping ($T \leq \omega$), or in a weaker way by allowing adding type assumptions. Similarly, contraction can be allowed in a general way via subtyping ($T \leq T \cap T$) or in a weaker way by changing all multiple-premise typing rules to use the same type environment in all judgments (premises and conclusion) and also adding some form of intersection elimination (e.g., as a typing rule). If intersection is not idempotent and neither weakening nor contraction are allowed, then the types are linear. The combination of idempotence and weakening or of weakening and contraction allows full non-linearity. Other different feature combinations may yield different results; for example, in System I the types are *affine* which means each singular component of an intersection type stands for at most one use instead of exactly one use.

Every typing $\langle A \vdash T \rangle$ can be interpreted by the set of λ-terms to which it can be assigned. A typing interpreted by a smaller set of terms is more discriminating (more precise) than one interpreted by a larger set of terms. A typing has different interpretations in different intersection type systems, depending on whether the types are linear or non-linear (among other things). For example, consider this typing:

$$\langle () \vdash (a \to a) \to a \to a \rangle$$

In an intersection type system allowing full non-linearity like the BCD system [3], this typing may be interpreted by the set $\{\mathtt{I}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$ where $\mathtt{I} = \lambda x.\, x$ and $\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots$ are the Church numerals $\mathtt{0} = \lambda f.\, \lambda x.\, x$, $\mathtt{1} = \lambda f.\, \lambda x.\, f \mathbin{@} x$, $\mathtt{2} = \lambda f.\, \lambda x.\, f \mathbin{@} (f \mathbin{@} x)$, and so on. (We are considering here only the β-normal forms in the typing's interpretation, because they are the most interesting members.) In contrast, if intersection types are solely linear as they are in the CDV system or our example type system, then this typing is interpreted by the smaller set of terms $\{\mathtt{I}, \mathtt{1}\}$. For example, the typing given above is not

a proper typing of $\mathsf{0}$ in our example type system, whereas $\langle () \vdash \omega \to a \to a \rangle$ is. In our example type system the typing $\langle () \vdash (a \to a) \cap (a \to a) \to a \to a \rangle$ can be assigned to the Church numeral $\mathsf{2}$, but not to any other Church numeral.

Linear types are more precise than fully non-linear types, but too much precision is sometimes undesirable. In a non-rank-restricted system of linear intersection types such as the example type system of this paper, in order for a type inference algorithm to be complete, it must produce principal typings. However under these conditions the principal typings of a term are known to be isomorphic to its β-normal form [11,41], so type inference has the same cost as evaluation. This is illustrated by a type inference algorithm of Carlier and Wells [6] which is proven to be step-by-step equivalent to β-normalization. Thus, for type inference to be practical, types must be limited to some finite rank k. Unfortunately, for every value of k, with linear intersection types, there are simply typable terms that are *not* typable with only linear types below rank k. For example, the Church numeral $\mathsf{2}$ has no linear typings below rank 2, though it is simply typable. Clearly, this is unsatisfactory.

System E [5] adds to intersection types a ! operator that serves to relax linearity in a controlled way. In System E, whenever $T \neq !\,T'$ for some T', then $T \not\leq T \cap T$, but it always holds that $!\,T \leq !\,T \cap !\,T \leq T \cap T$. This feature makes it possible to obtain the precision of linear types when it is useful, while preventing linearity from getting in the way when it is not needed, or too expensive to have. For example, in System E, we can assign the typing $\langle () \vdash !\,(a \to a) \to a \to a \rangle$ to the entire set of Church numerals, thereby avoiding the difficulties mentioned above. Both flexibility and expressiveness are provided via both intersection types and the ! type constructor: intersection types give a polymorphic/polyvariant analysis and ! distinguishes linear vs. non-linear types. When non-linear types are allowed, type inference restricted to rank-k has complexity that is complete for $\mathsf{DTIME}[\mathsf{K}(k-1,n)]$, where $\mathsf{K}(0,n) = n$ and $\mathsf{K}(t+1,n) = 2^{\mathsf{K}(t,n)}$, which is significantly better than the cost of normalization for the terms typable at rank k [29].

The integration of ! with E-variables is extremely simple; ! is added as a case of types (and also type environments so that $(!\,A)(x) = !\,A(x)$), expansions, and skeletons, and has this typing rule:

$$\frac{Q \triangleright M : \langle A \vdash T \rangle}{!\,Q \triangleright M : \langle !\,A \vdash !\,T \rangle}$$

These rules are added to expansion and substitution application:

$$[!\,E]\,X = !\,[E]\,X \qquad [S]\,!\,X = !\,[S]\,X$$

Finally, these subtyping rules give ! its meaning:

$$\frac{}{!\,T \leq \omega}\ \text{weakening} \qquad \frac{}{!\,T \leq T}\ \text{dereliction} \qquad \frac{}{!\,T \leq !\,T \cap !\,T}\ \text{contraction}$$

6 Related Work

6.1 Other variants of historical expansion

Barendregt, Coppo, and Dezani [3] proposed a system of intersection types which has become commonly known as the BCD system and features a very flexible subtyping relation. Ronchi della Rocca and Venneri [39] generalized the original definition of expansion to define principal typings for the BCD system. Ronchi della Rocca [38] gave the first principal typing inference algorithm for the BCD system. Van Bakel [44] advocated the use of a leaner system of intersection types called *strict intersection types*, which uses a simpler definition of expansion based on the technique used by Ronchi della Rocca and Venneri [39]. Coppo and Giannini [12] presented a system of *"simple" intersection types* [6] that uses a restricted form of expansion.

The various historical presentations of expansion have had difficulties which we believe kept expansion from being well understood. At the most basic level, there were notational difficulties with the earliest notions of identifying a nucleus via *underlining*. (We believe our definition of marked types in section 3.2 avoids these difficulties.) Attempting to sidestep the early notational difficulties Ronchi and Venneri [39] defined a more robust replacement of the notion of nucleus, but as a result their approach is very complicated and hard to understand. But more important than difficulties with merely defining expansion, a more fundamental issue is that historical expansion is a complex, non-local operation, in contrast with the modern use of E-variables where each case of the definition of expansion is given by a purely local algebraic rule. The non-local nature of historical expansion has made it difficult for readers to understand, makes proofs using expansion complicated, and makes it more difficult to generalize expansion to constructors other than intersection (an example generalization using modern expansion is discussed in section 5.2). The combination of the non-local nature of historical expansion and the inability to nest historical nuclei has meant that "composition" of interleaved uses of substitution and expansion has been only by building chains of individual operations; the use of such a chain merely applies the operations in sequence.

6.2 Expansion and rank-2 intersection types

The older definitions of expansion have been a bit hard to understand and implement, leading (in our opinion) people to focus on the easier rank-2 intersection types [43,24,13,14] rather than try to use the full power of intersection types. The key advantage of rank-2 intersection types over higher ranks is that when doing compositional type inference where constraints are always solved as soon as they are discovered, expansion never corresponds to inserting uses of intersection-introduction at deeply nested positions in typing derivations.

[6] This use of "simple" has nothing to do with the usual use of "simple" in the phrase "simple types".

Related to this, expansion never needs to insert uses of the intersection type constructor in typings underneath arrow types. Most of the complications of expansion can be avoided when using only rank-2 intersection types. However, because of the recent development of E-variables, expansion is no longer as difficult to understand and implement, and hence we believe there is no longer a strong reason to restrict intersection types to only rank 2.

6.3 Expansion and the omega type

Expansion takes each part of a nucleus and makes renamed copies of it joined by $\cap$. From the modern point of view it is clear that one can generalize this to leave *zero* copies, having the effect of replacing every part of a nucleus by ω, as we show in section 4. Nonetheless, historically this effect was achieved via more complicated (and also more delicate) mechanisms. For example, Coppo et al. [11] define an operation of *normalization* which, in addition to removing occurrences of ω from sequences, replaces by ω all types of the form $T_1 \to \cdots \to T_n \to \omega$. This requires ω to be a type constant (distinct from the empty sequence), which in turn requires forcing term variables to be given normalized types, to avoid compromising principal typings. Van Bakel's system of strict intersection types [44] also does not have a 0-ary expansion and defines substitution application to perform the same transformation as the normalization of [11]. Ronchi and Venneri [39] proved their results within the flexible BCD system, which does not need normalization because of the quotienting done on types, but still used substitution to introduce ω. In Ronchi's type inference algorithm for a variant of the BCD system [38], only part of a nucleus is turned into ω using a substitution during unification, and the rest needs to be cleaned up as an extra step following unification. Here again, using the same mechanism as for expansion would have made things much simpler.

6.4 The history of expansion variables

The origin of expansion variables can by traced back to the work of Kfoury on linearization of the λ-calculus [26,28], which contains neither E-variables nor types, but a germ of the later idea. Expansion variables first appeared in "Beta-reduction as unification" [27], but were still restricted to "type schemes" used during unification and did not yet appear officially in the type system, or even in "expansions".[7]

Kfoury and Wells later proposed System I [31,33], a type system where E-variables officially appear in types and expansions, and gave a principal typing algorithm for it. System I was later updated by Carlier [4] to add ω

[7] At this point, the connection between expansion variables and the earlier concept of "expansion" was not yet understood, as illustrated by this (mistaken) quote [27, footnote 3]: " 'Expansions' in this paper are unrelated to 'expansions' as defined in various articles by researchers at the University of Turin ..."

as a type and an expansion. Kfoury, Washburn and Wells [30] discussed implementation of type inference and compositional analysis. Various attempts to solve difficulties with System I were made but remained unpublished.

In work using E-variables through System I, solving unification problems generated during typing inference for λ-terms is referred to as "β-unification". We now avoid this name because of the confusion it can cause. In unification theory in general, given an equational theory E, the name E-unification refers to unification of terms modulo the theory E. So the reader might logically deduce that the name "β-unification" should refer to unification of λ-terms modulo the β equation, i.e., the name seems to refer to a variant of ordinary higher-order unification. There does not seem to be a nice short replacement for this name, so we usually now simply refer to "unification with E-variables".

It is worth noting that a mechanism similar to the expansion variables of System I was developed independently by Laurent Regnier in his Ph.D. thesis [36], which unfortunately is only available in French. In Regnier's work, *labels* ("*etiquettes*" in French) appear as superscripts in types and are used to explicitly delimit nuclei and guide expansion (for example, $e\,(a \to a) \to b$ would appear as $(a \to a)^{l} \to b$, where l is a label that plays the same role as the E-variable e). These labels are similar to the E-variables of System I and share the same problems.

System E [5] is the most recent system with E-variables and solves many of the problems that were present in System I. We briefly summarize the changes that were made.

The built-in renaming mechanism of expansion application that is present in System I (and all older notions of expansions) does not exist in System E, but instead substitutions are allowed as leaves of expansions, as discussed in section 3.3.4. As a consequence of this change, expansion corresponds in System E to splicing in typing rules (or admissible typing rules), and E-variable application establishes namespaces. A major benefit of this more principled way of doing expansion is that arbitrary substitutions compose easily. In contrast, composition is extremely painful in System I.

E-variable application appears in all entities in System E: types, expansions, skeletons, and also constraints. In contrast, this is not the case in System I and it causes unnecessary complications there. The ω type is also better integrated in System E.

System E also removes restrictions about where intersections, ω, and E-variables can occur, and adds flexible subtyping, non-linearity, and subject reduction, which were all missing from System I. Non-linear types are needed for efficient analysis, and together with flexible subtyping they allow gaining the power of the BCD system [3], which appears to be needed for call-by-need and call-by-value analysis. Not only does System E have the ! type constructor for relaxing linearity, but it was very easy to add.

7 Conclusion

Expansion is an operation needed to obtain *principal typings* for intersection types and also completeness of type inference. Expansion is an operation on typings that simulates the effect of splicing in typing rules uses at nested positions in some derivation of that typing. Expansion was originally used to introduce intersections but its use has been extended, first by Carlier [4] with ω, the nullary case of intersection, and later by Carlier, Polakow, Wells and Kfoury [5] with substitution application and non-linearity. *Expansion variables* can be used to implement expansion in a simple, clean and flexible way.

7.1 Near future of expansion variables

Ongoing work with expansion variables using System E includes developing type inference techniques making use of ! to allow efficient analysis and to cope with common programming language features such as tagged variants and mutually recursive definitions. Doing this smoothly seems to require several classes of E-variables ranging over different subsets of expansions.

So far, all uses of expansion correspond to introducing uses of typing rules that operate uniformly on every component of a typing. In the future, expansion may be generalized to introduce non-uniform typing rules (for example, this appears to be needed to handle union types). We expect that E-variables will make this considerably easier than previous ways of doing expansion.

System E was shown to enjoy subject reduction [5], but more theoretical issues remain to be investigated. In particular, principality has not yet been proven. Although typings can be inferred for all normalizing λ-terms via a unification procedure that exactly follows β-reduction [6], and the typings produced would be principal in the BCD system, System E's ability to distinguish between linear and non-linear types may complicate things.

7.2 Some interesting open challenges

Unification with E-variables has been well studied with constraints generated from pure λ-terms, but a general theory going beyond these cases still has to be developed. A first start is [1].

E-variable application may be considered as a restricted form of function application, where the function is determined by the expansion that ultimately replaces the E-variable. In this sense, unification with E-variables and expansion may be related to 2nd-order unification (2U), semi-unification (SU), or some restriction of 2U or SU. It would be interesting to define a direct reduction between any two of these problems.

A denotational semantics should be built for System E. It is not clear how to build a set-based model (such as a filter model) for System E, even if E-variables and ! are omitted, because the intersection type constructor is

not idempotent. In particular, it seems clear that the semantics of the type $T_1 \cap T_2$ can *not* be obtained in the usual way simply via set intersection from the semantics of T_1 and T_2. The existing systems for which such models have been built all have idempotent $\cap$.

Finally, we expect E-variables to add an additional challenging level of complication to any denotational semantics.

8 Acknowledgements

We are grateful to Mario Coppo for useful discussions on the history of expansion. We would also like to thank Mario Coppo, Mariangiola Dezani-Ciancaglini, Betti Venneri, A. J. Kfoury, and Adam Bakewell for comments on drafts of this paper.

References

[1] Adam Bakewell and Assaf J. Kfoury. Unification with expansion variables. Technical report, Department of Computer Science, Boston University, December 2004.

[2] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.

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

[4] Sébastien Carlier. Polar type inference with intersection types and ω. In ITRS '02 [22].

[5] Sébastien Carlier, Jeff Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *LNCS*, pages 294–309. Springer-Verlag, 2004.

[6] Sébastien Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. In *Proc. 6th Int'l Conf. Principles & Practice Declarative Programming*, 2004. Completely supersedes [7].

[7] Sébastien Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. Technical Report HW-MACS-TR-0012, Heriot-Watt Univ., School of Math. & Comput. Sci., January 2004. Completely superseded by [6].

[8] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard type inference. *Theoret. Comput. Sci.*, 272(1-2):69–111, February 2002.

[9] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type-assignment for lambda terms. *Archiv für Mathematische Logik*, 19:139–156, 1978.

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

[11] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantics. In J. R[oger] Hindley and J[onathan] P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560. Academic Press, 1980.

[12] Mario Coppo and Paola Giannini. Principal types and unification for simple intersection type systems. *Inform. & Comput.*, 122(1):70–96, 1995.

[13] F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. on Prog. Langs. & Systs.*, 25(4):401–451, 2003.

[14] F. Damiani. Rank 2 intersection types for modules. In *Proc. 5th Int'l Conf. Principles & Practice Declarative Programming*, pages 67–78, 2003.

[15] F. Damiani and P. Giannini. Automatic useless-code detection and elimination for HOT functional programs. *J. Funct. Programming*, pages 509–559, 2000.

[16] Ferruccio Damiani. A conjunctive type system for useless-code elimination. *Math. Structures Comput. Sci.*, 13:157–197, 2003.

[17] Mariangiola Dezani, Robert Meyer, and Yoko Motohama. The semantics of entailment omega. *Notre Dame J. Formal Logic*, 43(3):129–145, 2002.

[18] Paola Giannini, Furio Honsell, and Simona Ronchi Della Rocca. Type inference: Some results, some problems. *Fund. Inform.*, 19(1/2):87–125, September/October 1993.

[19] J[ean]-Y[ves] Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'Etat, Université de Paris VII, 1972.

[20] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, volume 2618 of *LNCS*, pages 284–301. Springer-Verlag, 2003. Superseded by [21].

[21] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004. Supersedes [20].

[22] *Proceedings of the 2nd Workshop on Intersection Types and Related Systems*, 2002. The ITRS '02 proceedings appears as vol. 70, issue 1 of *Elec. Notes in Theoret. Comp. Sci.*

[23] Thomas Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[24] Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[25] Assaf J. Kfoury. Beta-reduction as unification. A refereed extensively edited version is [27]. This preliminary version was presented at the Helena Rasiowa Memorial Conference, July 1996.

[26] Assaf J. Kfoury. A linearization of the lambda-calculus. A refereed version is [28]. This version was presented at the Glasgow Int'l School on Type Theory & Term Rewriting, September 1996.

[27] Assaf J. Kfoury. Beta-reduction as unification. In D. Niwinski, editor, *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*, pages 137–158. Springer-Verlag, 1999. Supersedes [25] but omits a few proofs included in the latter.

[28] Assaf J. Kfoury. A linearization of the lambda-calculus. *J. Logic Comput.*, 10(3), 2000. Special issue on Type Theory and Term Rewriting. Kamareddine and Klop (editors).

[29] Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101. ACM Press, 1999.

[30] Assaf J. Kfoury, Geoff Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In ITRS '02 [22]. The ITRS '02 proceedings appears as vol. 70, issue 1 of *Elec. Notes in Theoret. Comp. Sci.*

[31] Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pages 161–174, 1999. Superseded by [33].

[32] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [31], August 2003.

[33] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. Supersedes [31]. For omitted proofs, see the longer report [32].

[34] E. K. G. Lopez-Escobar. Proof-functional connectives. In C. Di Prisco, editor, *Methods of Mathematical Logic, Proceedings of the 6th Latin-American Symposium on Mathematical Logic, Caracas 1983*, volume 1130 of *Lecture Notes in Mathematics*, pages 208–221. Springer-Verlag, 1985.

[35] Robin Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17:348–375, 1978.

[36] Laurent Regnier. *Lambda calcul et réseaux.* PhD thesis, University Paris 7, 1992.

[37] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[38] Simona Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2):181–209, March 1988.

[39] Simona Ronchi Della Rocca and Betti Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, January 1984.

[40] Patrick Sallé. Une extension de la théorie des types en λ-calcul. In G. Ausiello and Corrado Böhm, editors, *Fifth International Conference on Automata, Languages and Programming*, volume 62 of *LNCS*, pages 398–410. Springer-Verlag, July 1978.

[41] Émilie Sayag and Michel Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, October 16, 1996.

[42] Paweł Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Structures Comput. Sci.*, 7(4):329–358, 1997.

[43] Steffen J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

[44] Steffen J. van Bakel. Principal type schemes for the strict type assignment system. *J. Logic Comput.*, 3(6):643–670, December 1993.

[45] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.