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

Sébastien Carlier J. B. Wells

Heriot-Watt University
School of Mathematical and Computing Sciences
Technical Report HW-MACS-TR-0029
February 2005

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.

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 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 $\forall$-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 $\forall$-quantification, and its limitations

Most statically typed functional languages use extensions of the well known Hindley-Milner (HM) type system [40], which obtains polymorphism using $\forall$-*quantification*. Consider the following Standard ML

(SML) program fragment, where type annotations (in superscript) have been added to some program points:

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

The *type scheme* $\forall a.(a \to a)$ 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 $\forall$-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 $\forall$-quantification, it yields no information about the different uses of `id`. As a result of using $\forall$-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 $\forall$-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 $\forall$-quantification also generally do not have *principal typings* [50], strongest typings which imply all the others for the same term. Wells [50] proved the absence of principal typings both for HM and for System F [22, 42]. In the above example, the program fragment $(\texttt{id}\ 1,\ \texttt{id}\ 2.0)$ 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 [27], and accurate type error messages [27, 24].

1.2 Intersection types: some advantages, some issues

In contrast, intersection types provide type polymorphism by *listing* usage types [13]. Here is the same example SML program fragment as above, except annotated with intersection types:

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

The original motivation for the name *intersection types* was suggested by their semantics [13]: 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[1], 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 [39] (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 $(\texttt{false}, \texttt{true})$. Urzyczyn [47] proved that a λ-term from which this

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

example is derived is not typable in F_ω, considered the most powerful type system with $\forall$-quantifiers [21]. In contrast, the same λ-term is typable in the *rank-3 restriction* of intersection types.

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 [33].

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* [26], *dead code* [18, 15], and *totality* [10]. 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 [27, 16, 17].

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 [46]. 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

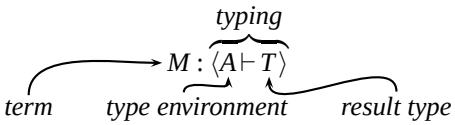
This section introduces a minimal intersection type system which will be used to support the discussion of expansion. In our experience, the traditional ways of presenting type systems are not optimal, especially for intersection types, so our notation differs from the usual in many ways. Before introducing our actual type system, we begin in subsections 2.1 and 2.2 by discussing notational changes for typing judgements and the intersection type constructor which we have found to help. We then present in subsection 2.3 the intersection type system we will use. Finally, we discuss in subsection 2.4 the reasons why we use two distinct kinds of proof terms in each typing judgement.

2.1 Better notation for typing judgements

In type systems for the λ-calculus, typing judgements have been traditionally written in the form $A \vdash M : T$, meaning that the λ-term M has the result type T when using the type assumptions for free variables given by the type environment A. This encourages regarding only the type T as a property of the M, and viewing A as the context of reasoning for this conclusion.

However, for an analysis to be compositional, it is important that each program fragment be analyzed independently, and hence the type environment must be regarded as part of the *result* of analysis rather than as part of the *context* of analysis. For example, at the module level, compositionality requires that analysis of a program module be performed using neither the source code of other modules nor information derived from them, such as type signatures. This goes beyond separate compilation, which considers "translation units" (text files containing source code, sometimes corresponding to the notion of modules of the language); compositionality works at the level of arbitrarily small expressions of the language.

To encourage compositional thinking, we avoid the traditional notation and write the components of typing judgements in this order instead:



Thus, a term M is assigned a typing $\langle A \vdash T \rangle$, which as a whole is regarded as a property assigned to M independently of any other information.

The motivation for this reordering of typing judgements is generally applicable beyond intersection types to all kinds of type systems. In addition to this reordering, for reasons specific to intersection types we will also add another kind of proof term to judgments, as discussed below in subsections 2.3 and 2.4.

2.2 Symbol used for the intersection type constructor

In the literature, an intersection of the types $T_1, \ldots, T_n$ is sometimes written as a *sequence* $[T_1, \ldots, T_n]$ (e.g., in [13, 14]) or more commonly with a binary type constructor like $T_1 \cap \cdots \cap T_n$, although symbols like $\cap$ [44, 41, 48] and $\wedge$ [43, 36, 38, 6, 34, 5] are more usual. This paper uses $\cap$ to avoid symbol overloading. We do not use $\cap$, set intersection, because the arguments of the intersection type constructor are types and not sets, and because the denotational semantics of intersection types can not always use set intersection (as discussed in section 8.2). We do not use $\wedge$, logical conjunction, because the arguments are not Boolean values, and also because it does not express the essential proof-functional (not truth-functional) character of the intersection type constructor when viewed as a constructor of logical propositions.

Related to the choice of symbol is the issue of which laws can be expected to hold for the intersection type constructor. In the literature there is much variation on the choice of whether it is associative and commutative, whether it has a neutral (sometimes called a unit or an identity), and whether it is idempotent. For this reason, we think it is better not to reuse a symbol which in readers' minds might already be viewed as having or not having these properties.

2.3 An intersection type system

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 ($\omega \cap T = T$). Although it is extremely convenient, this quotienting sometimes causes readers difficulty; this will be further discussed in section 6.2.

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:

$$M ::= x \mid \lambda x.M \mid M_1 @ M_2$$

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 $M_1 @ M_2$ 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[2]. The syntax of skeletons is this:

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

Typing judgements are in the shape mentioned already in subsection 2.1, except with the skeleton added as a prefix:

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

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 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 justification for having skeletons is discussed in section 2.4.

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

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

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

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

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 (∩ 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 [20]. In fact, this system has the further property that types are *linear*; this is further discussed in section 6.4.

The system presented in this section is essentially a streamlined version of the original system of intersection types by Coppo, Dezani and Venneri [13], which we call the CDV system. In [13] and also here, ∩

[2] 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.

is associative and commutative (AC) but not idempotent (i.e., $T \cap T \neq T$ unless $T = \omega$)[3], and type environments contain only assumptions for variables that are actually used (the systems are relevant). The main difference (unimportant for the examples in this section, but useful for later sections) is that, for simplicity, we allow $\cap$ and ω to the right of $\rightarrow$.

2.4 Skeletons and typing judgments

In type systems and other logical systems, proofs are often formed by trees of judgments where each tree node is obtained from its children via a rule of the system. Such trees are often called *derivations*. Logical systems often enhance judgements with various forms of term annotations, generically called *proof terms*. Proof terms have at least two important roles to play. (1) Sometimes, these proof terms take the role of a *subject* about which the rest of judgment is a statement or *predicate* which the subject satisfies. (2) Sometimes, the proof term in each judgement concisely represents the important details of a derivation concluding with that judgement, in which case proof terms can make reasoning about and manipulating derivations easier.

Due to difficulties with the $\cap$-introduction typing rule, intersection type systems have until recently typically only used pure (type-free) λ-terms in typing judgements. Unfortunately, pure λ-terms fail to capture the important structure of derivations with intersection types, and hence can only fulfill role (1), not role (2). This has meant that discussions of derivations (e.g., presenting example derivations or rules for transforming derivations) become either exceedingly verbose or dangerously informal. This issue is much more serious for intersection type systems than for systems without intersection types due to the way that intersection types can contain detailed information about different uses of the same functions. We estimate that this paper would at least double in length if presented in the old style (or worse, some examples would have to be omitted or have details chopped out of them).

To solve this problem, like some earlier systems [33, 34, 7, 8] our intersection type system has two distinct kinds of proof terms in typing judgements, pure λ-terms which fulfill role (1), and also skeletons which fulfill role (2). We put skeletons inside judgements so that their correspondence with derivations is made precise by the formal rules.

Skeletons were first used in [35] (although not by that name), where they *replaced* the pure λ-terms in judgements. This required a complicated notion of type erasure for obtaining pure λ-terms from skeletons in order to be able to enforce the condition of the intersection-introduction typing rule that both premises mention the *same* pure λ-term. This approach was further developed in [51, 52]. Skeletons first appeared in judgements *in addition to* pure λ-terms in [33]. Skeletons in a form close to that used in this paper first appeared in [34]. The skeleton form used in this paper first appeared in [7]. To avoid confusion, we must point out that the word "skeleton" in [36, 38] has a somewhat different meaning, where skeletons are in fact trees of judgements plus the names of the typing rules used to derive each judgement. The skeletons we present here are equivalent to that notion in the sense that each skeleton uniquely determines a tree of judgements and the typing rule used to derive each judgement in the tree from its children.

Our approach with two distinct kinds of proof terms in each judgement is needed because the usual "Church-style" approach to type-annotated λ-terms fails to work for intersection types. Wells and Haack [53, 54] provide a detailed discussion of the difficulties of having type-annotated proof terms for the intersection-introduction typing rule. They also define *branching types*, a way to achieve the power of intersection introduction without needing to check equality of the pure λ-term in multiple premises of a typing rule. This results in a system equivalent in a certain sense to intersection types that has proof terms that simultaneously fulfill both roles (1) and (2). Unfortunately, there are some technical difficulties that must be overcome before it is sensible to discuss and explain expansion using branching types. There are also some other approaches surveyed in [53, 54] that do not provide proof terms that can fulfill role (2) and which anyway would have similar difficulties serving as a host for an explanation of expansion, so we are not using any of those approaches in this paper.

[3] This is after translating to modern notation the types of [13], where intersection types are written in the form $[T_1, \ldots, T_n]$.

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:

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

In the intersection type system considered, term M is typable because it has a normal form [13], so we should be able to build a typing derivation for it.

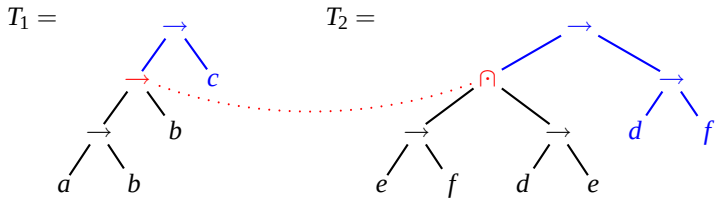
Subterms M_1 and M_2 are in normal form and we can easily obtain their *principal typings* using the algorithm of Coppo, Dezani and Venneri [13]:

$$\begin{array}{ll}
Q_1 = \lambda x.\, x^{:((a\to b)\to b)\to c}\,@\,(\lambda y.\, y^{:a\to b}\,@\,z^{:a}) & \\
Q_1 \triangleright M_1 : \langle (z:a) \vdash T_1 \to c\rangle & \text{with } T_1 = ((a\to b)\to b)\to c \\
Q_2 = \lambda f.\lambda x.\, f^{:e\to f}\,@\,(f^{:d\to e}\,@\,x^{:d}) & \\
Q_2 \triangleright M_2 : \langle () \vdash T_2\rangle & \text{with } T_2 = ((e\to f)\cap(d\to e))\to(d\to f)
\end{array}$$

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.

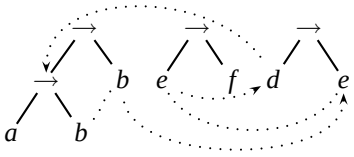
Can we do this merely by substitution, replacing type variables by types? Lining up matching bits and drawing the two types as trees makes things easier to follow:

$$\begin{array}{l}
T_1 = ((a\to b)\to\quad b\quad)\to\quad c \\
T_2 = ((e\to f)\ \cap\ (d\to e))\to(d\to f)
\end{array}$$



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:



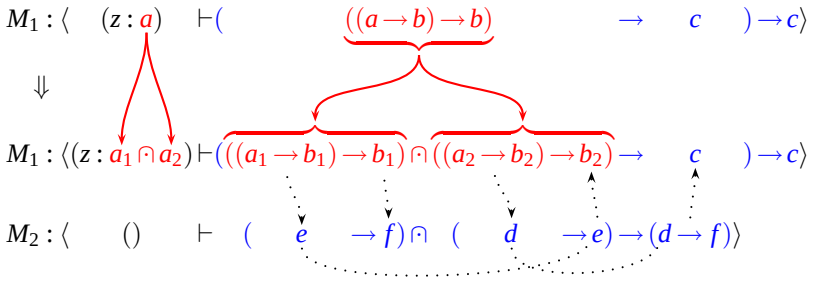
Dotted arrows depict some of the required variable substitutions, while dotted lines show connections between occurrences of the same variable. Note that there is a cycle; the equation $b = e = d = a\to b$ cannot be solved without recursive types. Although recursive types would solve this example, they 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* [13] 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 $M_1 @ M_2$. Thus, expansion solves the problem. But what is expansion?

3.2 What historical expansion is

Coppo, Dezani, and Venneri [13] 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 components, so sequences 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.

Although the definitions in the literature of nucleus were somewhat informal 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:

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

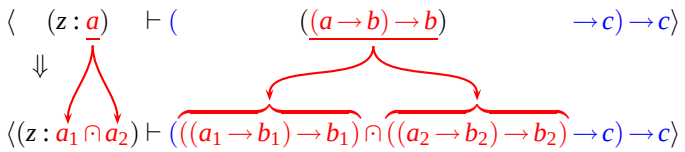
Marked types are like normal types except that some occurrences of type variables and $\rightarrow$ are marked by underlining. Marks are not allowed to nest by the grammar; if an occurrence of $\rightarrow$ is marked, then all types underneath it must be mark-free. We allow underlining a whole expression 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 $|T_1 \underline{\rightarrow} T_2| = T_1 \rightarrow T_2$.

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 $\rightarrow$, then all other occurrences of a must also be.

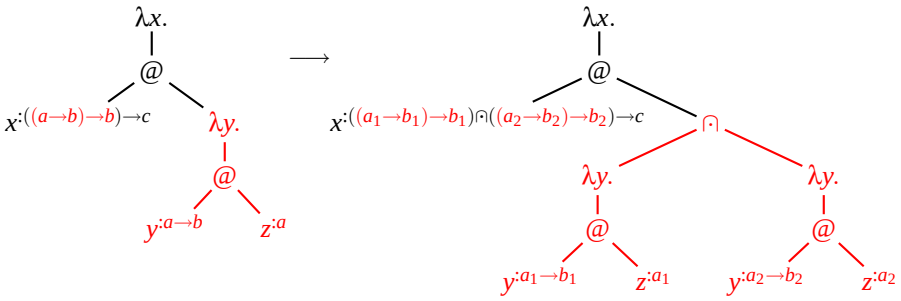
We can now explain what expansion does. Let a *renaming* function be a total injective function from T-variables to T-variables. A renaming function r is applied to types like a substitution, so that $r(T)$ is the new type that results from T by replacing every T-variable a by $r(a)$. An expansion operation 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 $\cap$ marks a use of the intersection-introduction typing rule:



□

EXAMPLE 3.2. In the following example, underlining does *not* delimit a nucleus, because b occurs both inside and outside underlining:

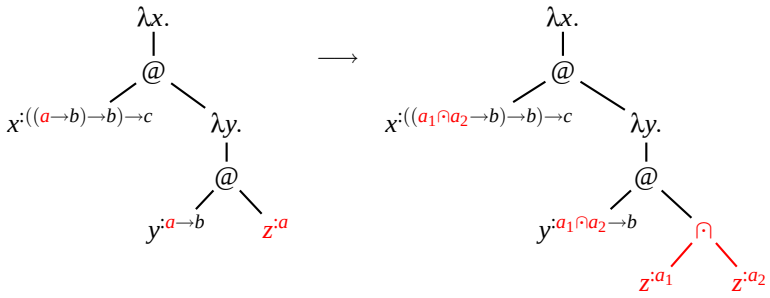
$$\langle (z : \underline{a}) \vdash (((\underline{a \rightarrow b}) \rightarrow b) \rightarrow c) \rightarrow c \rangle$$

Trying to do expansion with this marked typing (which is not a nucleus) does not correspond to a meaningful typing derivation transformation. For instance, the resulting typing would mention b_1, b_2, and b, breaking the link between the two occurrences of b that exists in the original typing. □

EXAMPLE 3.3. Here is another (smaller) nucleus and one possible expansion:

$$\langle (z : \underline{a} \vdash (((\underline{a} \to b) \to b) \to c) \to c \rangle$$
$$\langle (z : \underline{a_1 \cap a_2} \vdash (((\underline{a_1 \cap a_2} \to b) \to b) \to c) \to c \rangle$$

Again, expansion with this nucleus corresponds to a transformation on a derivation of the original typing, for example:



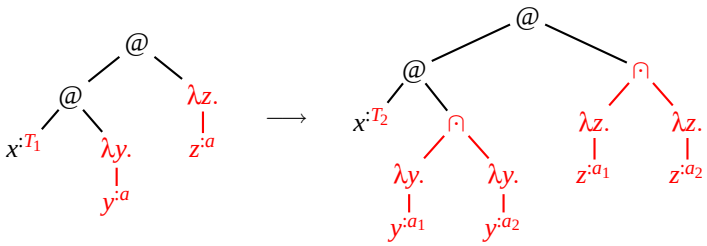
□

The reader may notice at this point that, in example 3.3, the same effect as the expansion performed can be obtained by replacing a by $a_1 \cap a_2$ using a type-level substitution. This is of course not the case in general, as can be seen by replacing a by, for example, $a \to a$ prior to expansion; in this case, type-level substitution can only turn this type into $(a_1 \cap a_2) \to (a_1 \cap a_2)$, while expansion can get the stronger type $(a_1 \to a_1) \cap (a_2 \to a_2)$.

EXAMPLE 3.4. An expansion of the following typing and nucleus corresponds to the simultaneous addition of two[4] uses of the intersection-introduction typing rule to any derivation of this typing:

$$\langle x : (\ \underline{a \to a}\ ) \to (\ \underline{a \to a}\ ) \to b \vdash b \rangle$$
$$\langle x : (\ \underline{(a_1 \to a_1) \cap (a_2 \to a_2)}\ ) \to (\ \underline{(a_1 \to a_1) \cap (a_2 \to a_2)}\ ) \to b \vdash b \rangle$$

Here is one possible corresponding transformation on a derivation:



where $T_1 = (\ \underline{a \to a}\ ) \to (\ \underline{a \to a}\ ) \to b$
and $T_2 = (\ \underline{(a_1 \to a_1) \cap (a_2 \to a_2)}\ ) \to (\ \underline{(a_1 \to a_1) \cap (a_2 \to a_2)}\ ) \to b.$

□

Example 3.4 illustrates the case where a nucleus contains the types in the judgements of *several* subderivations, here of $\lambda y.y^{:a}$ and $\lambda z.z^{:a}$, whose typings both mention the type $a \to a$. Note that $\langle x : (\underline{a \to a}) \to (a \to a) \to b \vdash b \rangle$ is *not* a nucleus (some occurrences of a are left out). This shows that expansion simulates a *global* operation on typing derivations that might require the *simultaneous* addition of several uses of the intersection-introduction typing rule.

In its original definition [13], 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 [44]

[4] At least two. In fact terms that β-reduce to $x\,@\,(\lambda y.y)\,@\,(\lambda y.y)$ can derive the same typing but may need more uses of the intersection-introduction typing rule added to their typing derivations to carry out this expansion.

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 4.

3.3 Modern expansion with expansion variables

Expansion variables (E-variables) were first used as type constructors by Kfoury and Wells in System I [36, 38] to simplify reasoning about and implementing the operation of expansion. The most modern system with E-variables is System E [7], 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 $\cap$, 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 two examples illustrate this.

EXAMPLE 3.5. 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* $E = (a := a_1, b := b_1) \cap (a := a_2, b := b_2)$. 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 3.11. □

EXAMPLE 3.6. Here is another typing of M_1 from the example shown in section 3.1, with an expansion variable in a different position:

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

In this typing e delimits the smaller nucleus shown in example 3.3. As noted earlier, here an intersection can also be introduced underneath e without doing expansion, by substituting for the T-variable a. In general, an E-variable wrapped around T-variables only in a typing is redundant, but in this example simply replacing a by an arrow type reestablishes the importance of e. □

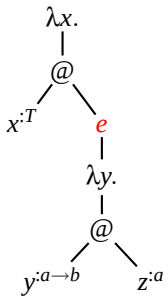
3.3.1 E-variable application typing rule

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 3.7. Here is a skeleton deriving the typing in example 3.5:

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



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 6 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. We say more about this later, in remark 3.13.

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:

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

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

EXAMPLE 3.8. Here are some applications of a substitution to a variable:

$$\begin{aligned} &[a := T]\, a = T \\ &[a := T]\, b = b \\ &[e := E]\, e = E \\ &[a := T_1, b := T_2]\, a = T_1 \\ &[a := T_1, b := T_2]\, b = [b := T_2]\, b = T_2 \\ &[a := T_1, b := T_2]\, c = [b := T_2]\, c = [\boxdot]\, c = c \\ &[a := T_1, a := T_2]\, a = T_1 \end{aligned}$$

As the last case shows, if a substitution has several assignments for the same variable, then only the first one is considered. This is for simplicity of the definitions; it would be possible to make a more complicated definition that prevented multiple assignments for the same variable from arising but it is not clear that the gain would be worth the complexity. □

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

$$\begin{aligned} [S]\,(T_1 \to T_2) &= [S]\, T_1 \to [S]\, T_2 & [S]\, x^{:T} &= x^{:[S]\,T} \\ [S]\,(X_1 \cap X_2) &= [S]\, X_1 \cap [S]\, X_2 & [S]\, \lambda x.\, Q &= \lambda x.\, [S]\, Q \\ [S]\, \omega &= \omega & [S]\,(Q_1 \mathbin{@} Q_2) &= [S]\, Q_1 \mathbin{@} [S]\, Q_2 \end{aligned}$$

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

EXAMPLE 3.9. The following equalities hold:

$$\begin{aligned} &[e := \boxdot]\,(x^{:e\,a \to b} \mathbin{@} e\, y^{:a}) = x^{:a \to b} \mathbin{@} y^{:a} \\ &[e_1 := \boxdot]\,(e_1\, e_2\, a) = [[e_1 := \boxdot]\, e_1]\,(e_2\, a) = [\boxdot]\,(e_2\, a) = e_2\, a \\ &[e_2 := \boxdot]\,(e_1\, e_2\, a) = [[e_2 := \boxdot]\, e_1]\,(e_2\, a) = [\boxdot]\,(e_1\, e_2\, a) = e_1\, e_2\, a \\ &[a := T]\,(e\, a) = [[a := T]\, e]\, a = [[\boxdot]\, e]\, a = [e\,\boxdot]\, a = e\, a \end{aligned}$$

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. The rationale for the behavior of our substitutions is explained later, in section 3.3.4. □

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.

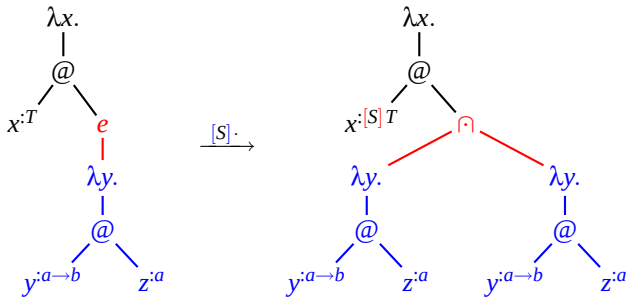
Expansion application of an intersection expansion is simply:

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

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 3.10. Here is the effect of applying $S = (e := (\boxdot \cap \boxdot))$ to the skeleton Q from example 3.7:



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

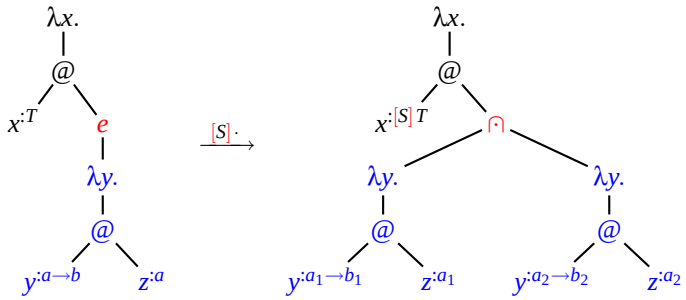
$$E ::= \cdots \mid S \mid \cdots$$

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 3.11. 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 3.7:

[5] In fact, in the original System E paper [7], there is an explicit rule for substitution 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.

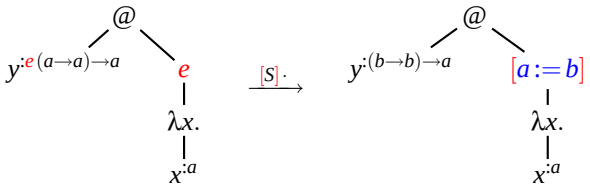


Here T is as in example 3.10 and $[S]\,T = ((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c$. On the typing (given originally in example 3.5) obtained from the skeleton Q, the substitution S has exactly the effect required to solve our original motivating example from section 3.1:

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

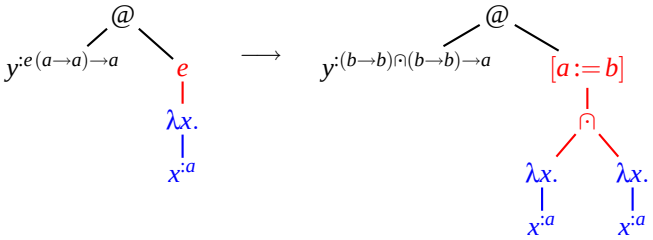
Allowing substitution as a case of expansion has interesting implications. Consider this example:

EXAMPLE 3.12. Let $T_1 = e\,(a \to a) \to a$ and $S = (e := (a := b))$. Applying S to T_1 yields $T_2 = [S]\,T_1 = (b \to b) \to a$. Here is a corresponding skeleton transformation:



See how the smaller substitution $(a := b)$ is applied at a nested position in the derivation, and thus does not affect the result type of the entire derivation, which remains a.

At this point, the reader way wonder why we do not allow splicing in more typing rules underneath a substitution, as illustrated by the following transformation:



This would justify the use of "$S\,E$" as the case of expansion for substitution application, the preceding transformation being obtained by applying a substitution of the form $(e := ((a := b)\ (\boxdot \cap \boxdot)))$. It turns out that this more general syntax is not needed, because we can always achieve the same effect with the simpler syntax. In the above example, this is done by applying the substitution $(e := (a := b) \cap (a := b))$. □

REMARK 3.13. Having substitution as a case of expansion makes E-variables effectively establish *namespaces* that can be manipulated separately. Variables living in distinct namespaces are thus independent. Consider for example:

$$\begin{array}{rl} [e := e\,(a := b)]\,(e\,a \to a) & = e\,b \to a \\ [a := c, e := e\,(a := b)]\,(e\,a \to a) & = e\,b \to c \\ [e_1 := e_1\,(a := b), e_2 := e_2\,(a := c)]\,(e_1\,a \to e_2\,a) & = e_1\,b \to e_2\,c \end{array}$$ □

	$\cap$ *is AC*	*has linear types*	*has non-linear types*	$\cap, \omega$ *to the right of* $\rightarrow$	$\cap, \omega$ *distribut over* $\rightarrow$
Coppo, Dezani and Venneri, 1980 [13]	•	•			
Barendregt, Coppo and Dezani, 1983 [3]	•		•	•	•
van Bakel, 1993 [49]	•		•		
Kfoury and Wells, 1999 [36, 38]		•			
Carlier et al, 2004 [7] with $\leq_{\mathsf{refl}}$	•	•		•	
Carlier et al, 2004 [7] with $\leq_{\mathsf{nlin}}$	•	•	•	•	
Carlier et al, 2004 [7] with $\leq_{\mathsf{flex}}$	•	•	•	•	•

Figure 1: Features of some intersection type systems.

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. Section 4 discusses other variants of historical expansions and their limitations. Sections 5 and 6 discuss extensions of the theory of expansion variables beyond what was done with historical expansion.

4 Other variants of historical expansion

Figure 1 gives a summary of some of the features of various intersection type systems that use expansion. Note that "AC" means associative and commutative. The table has distinct columns for linearity (lack of weakening and contraction or idempotence) and non-linearity because one system allows combining both. The rest of this section gives details on the various type systems and inference algorithm using (or, in the case of section 4.4, avoiding) the historical notion of expansion.

4.1 Principality and type inference for the BCD system

Barendregt, Coppo, and Dezani [3] proposed a system of intersection types which has become commonly known as the BCD system. It uses an intersection type constructor (as opposed to sequences) which is not restricted in where it can appear in types (e.g., it can appear immediately to the right of arrows), and features a very flexible subtyping relation $\leq$, which in particular allows weakening ($T \leq \omega$), contraction ($T \leq T \cap T$), and also distribution of $\cap$ and ω over $\rightarrow$ (for instance, $(T \rightarrow T_1) \cap (T \rightarrow T_2) \leq T \rightarrow (T_1 \cap T_2)$ and $\omega \leq \omega \rightarrow \omega \leq T \rightarrow \omega$). Types are quotiented by the largest symmetric subset $\sim$ of $\leq$, which effectively makes intersection associative, commutative, and idempotent with neutral ω.

A notion of expansion similar to that the original definition of Coppo, Dezani, and Venneri [13] is used by Ronchi della Rocca and Venneri [44] to define principal typings for the BCD system and to show that all and only the derivable typings can be obtained by combining substitution, use of a subtyping relation (this operation is called *rise*), and expansion. This result is similar to the main result of [13] but it is done in the extended BCD system. This work does not use the earlier notion of nucleus; instead it defines a generalized mechanism that works for all typings of the BCD system.

Ronchi della Rocca [43] gave the first principal typing inference algorithm for a variant of the BCD system.[6] Unlike in the BCD system, in Ronchi's work the intersection type constructor is not associative, commutative or idempotent, and it does not have ω as a neutral; explicit uses of a typing rule for subtyping are needed instead to achieve the same effect (e.g., $T \cap \omega \neq T$, but $T \cap \omega \sim T$). This difference is crucial for

[6] Ronchi's system is more precise than the BCD system; for example, it has different principal typings for $(x@x)$ and $((\lambda y.\lambda z.z@y)@x@x)$.

the type inference technique of that paper. As in the BCD system, intersections are allowed to appear to the right of arrows, though this feature is never used in the principal typings that are inferred. All the typings of the BCD system are proved to be obtainable from principal typings via expansion and subtyping. Type inference starts at the leaves of a pure (type-free) λ-term and proceeds toward the root; typing application nodes requires solving a unification problem. The unification algorithm uses expansion.

Zimmer and Boudol [5] have a type inference algorithm that is a variation on the work of Ronchi della Rocca [43] for a more restricted system. A set of T-variables used to identify what corresponds to the older notion of nucleus is added to constraints for type inference.

4.2 Strict intersection types

Van Bakel [49] advocated the use of a leaner system of intersection types called *strict intersection types*. It is a restriction of the BCD system [3] where $\cap$ and ω can not occur immediately to the right of arrows. Like the BCD system, this system supports non-linearity ($\cap$ is idempotent), and the proofs are made simpler by the restriction to strict types. In van Bakel's paper, expansion is based on the technique used by Ronchi della Rocca and Venneri [44], but the use of strict types simplifies its definition.

Although strict intersection types do not restrict the set of typable pure λ-terms, they preclude certain applications. They prevent representing program analysis results (other than just type safety) for call-by-need and call-by-value languages, where the result of a function (to the right of an arrow) needs to be given a polyvariant analysis (an intersection type). An example of this for usage analysis is given in section 6.4.

Furthermore, when using expansion variables, strict types disallow types of the form $T_1 \to e\, T_2$, because expansion could turn e into an intersection. This restriction prevents some uses of E-variables to establish namespaces. This is significant because one type inference algorithm for System E [8] uses the ability to put E-variables to the right of arrow to work with only *three* distinct E-variables; this makes proofs simpler by removing any need for renaming and also makes it easy to relate the strategy used during type inference to β-reduction.

4.3 Expansion and "simple" intersection types

The system of *"simple" intersection types*[7] by Coppo and Giannini [14] uses a restricted form of expansion which corresponds to introducing intersections only in the result type position of judgements, and only does renaming on type variables which do not also occur in the environment. We noticed a striking similarity between this system and Jim's System P [28], in which $\forall$-quantification offers the same power as Coppo and Giannini's restricted form of intersection introduction. Establishing this correspondence is beyond the scope of this paper, but it seems possible that both systems have exactly the same typing power.

4.4 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 rather than try to use the full power of intersection types.

Usually, intersection types restricted to rank 2 have been defined as types from the following grammars:

$$T^0 ::= a \mid T^0 \to T^0 \qquad T^1 ::= T^0 \mid T_1^1 \cap T_2^1 \mid \omega \qquad T^2 ::= T^1 \mid T^1 \to T^2$$

This definition omits the possibilities of an intersection $T_1^2 \cap T_2^2$ of two rank-2 types and also a function type $T_1^0 \to T_2^1$ from a rank-0 type to a rank-1 type. Although these would be included in the original definition of rank 2, it turns out these forms can always be eliminated from derivations (because they are not strict as defined in section 4.2) so papers on rank-2 intersection types have forbidden them grammatically.

The restriction to rank-2 intersection types constrains the possible typing derivations; the key idea is that a λ-abstraction can be given a T^2 type only if it is syntactically applied, not counting redexes surrounding the abstraction. It follows that a λ-term is rank-2 typable iff it is simply typable after one iteration of

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

complete development [4] generalized to mark all *prime redexes* [41, 19]. Each additional rank beyond 2 corresponds to doing one more iteration of complete development of all prime redexes [33].

Note that unlike partial evaluation followed by simple type inference, intersection typing derivations give information about the unreduced term. This is what makes intersection types useful for program analysis. Note also that programs using polymorphic/polyvariant higher-order functions will benefit most from the use of arbitrary-rank 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. 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 there is no longer a strong reason to restrict intersection types to only rank 2.

4.5 Limitations of historical expansion

There were many difficulties with presentations of expansion before E-variables were introduced.

The first problem was notational. Our definition of marked types given in section 3.2 accurately reflects the intention of the early notions of identifying a nucleus via *underlining*, but masks the difficulties of the earlier notation. The problem is that the early notion of underlining was informal, operating on the *syntactic expression* of the types, and therefore was not truly compatible with the underlying *mathematical* types, in which $\cap$ is AC. What was needed was a notion of *occurrences* of types within types, but this is difficult to formulate in the presence of an AC operator, and so the early notation of underlining was left informal. Perhaps understanding this, Ronchi and Venneri [44] defined their replacement of the notion of nucleus more robustly, but as a result it became more complicated because it requires building a set of all types that can possibly be affected by expansion based on the entire typing to which expansion is applied. Our definition using marks on arrows and T-variables cleans some of this difficulty up and makes the notion of nucleus precise even when $\cap$ is AC.

However, a more fundamental difficulty of historical expansion is that it 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 later in section 6.4). 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.

Historical expansion also is not easily extended to handle additional type constructors, because it relies on T-variables to identify where to operate. Adding type constants can cause the historical expansion definitions to introduce intersections in different positions before and after replacing a type variable by a type constant. Historical expansion may also produce types with completely different semantics in refined systems such as System E, which has both linear and non-linear types.

5 The omega expansion

Some intersection type systems have a type written ω, which was originally added by Sallé [45] to type systems developed by Coppo and Dezani [11, 12]. The type ω is given a case in skeletons and a corresponding typing rule:

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

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. For example in [13] which has both types and sequences of types, the ω type constant is distinct from the empty sequence (written $[\,]$), and a *normalization* operation is defined to (as just one of its several purposes) remove occurrences of ω from sequences.

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, but historically this effect was achieved via more complicated mechanisms.

For example, Coppo et al. [13] define an operation of *normalization* which, in addition to removing occurrences of ω from sequences, replaces by ω all types of the form $T_1 \to \cdots \to T_n \to \omega$. Substitution is first used to replace some T-variables to the right of arrows by ω, and normalization performed afterward turns entire chains of arrows ending with ω into a single ω. Of course, this requires that ω be permitted to the right of arrows; since sequences can only occur to the left in [13], a consequence of this technique is that ω had to be kept distinct from the empty sequence. Also, to preserve principality, derivations of judgements such as this one have to be ruled out:

$$(\lambda w.\, x^{:T})\,@\,(y^{:T' \to \omega}\,@\,z^{:T'}) : \langle (x : T, y : T' \to \omega, z : T') \vdash T \rangle$$

The typing of this skeleton would not be obtainable from the principal typing of $(\lambda w.\, x)\,@\,(y\,@\,z)$ except via weakening, which is not allowed in [13]. The solution used by Coppo et al. is to force term variables to be given normalized types.

Van Bakel's system of strict intersection types [49] also does not have a 0-ary expansion. Because he does not allow ω to the right of arrows, a transformation equivalent to the normalization of [13] is built into substitution application:

$$[a := \omega]\,(T \to a) = \omega$$

Because weakening is allowed, there is no loss of principality.

Ronchi and Venneri [44] did not need normalization because their results are proved within the flexible BCD system, in which types are quotiented by a relation such that $\omega = T_1 \to \cdots \to T_n \to \omega$ for any $T_1, \ldots, T_n$. The BCD system has types (but not sequences) with an intersection type constructor (whose use is unrestricted), so ω is the unit of $\cap$. The problem with loss of principality mentioned above also does not happen, because the BCD system allows weakening via subtyping. Still, expansion could have naturally been generalized as explained above, but this was not done.

In Ronchi's type inference algorithm for a variant of the BCD system [43], 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.

Note that the historical approach of using substitution together with either normalization or built-in equalities to do the equivalent of ω expansion does not survive λ-calculus extensions. For example, it is impossible to turn $\mathsf{int} \to \mathsf{int}$ into ω in the following typing via substitution:

$$\lambda x.\, x\,@\,(\lambda y.\, y + 5) : \langle () \vdash ((\underline{\mathsf{int} \to \mathsf{int}}) \to a) \to a \rangle$$

However, typing the application of the above term to $\lambda w.\, z$ would need the change from $\mathsf{int} \to \mathsf{int}$ into ω.

To the best of our knowledge, Carlier [6] first used expansion to introduce ω; all papers before it used substitution plus some other mechanism instead. With E-variables, ω is straightforward to add as a case of expansion, as first done by Carlier [6]:

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

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

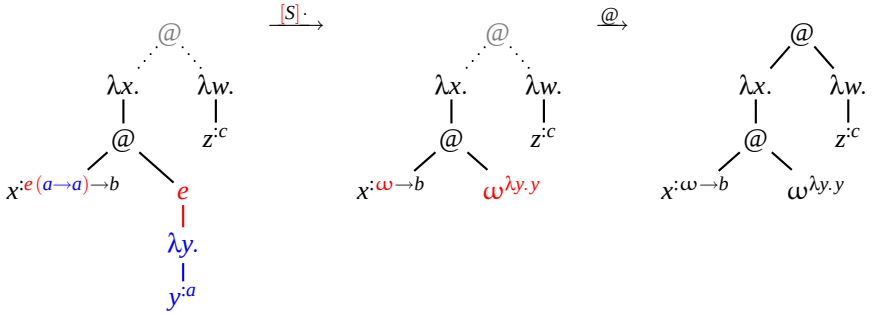
$$\begin{array}{lll} \text{on skeletons:} & [S]\,\omega^M = \omega^M & [\omega]\,Q = \omega^{\mathsf{term}(Q)} \\ \text{on other sorts:} & [S]\,\omega \quad = \omega & [\omega]\,X = \omega \end{array}$$

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

EXAMPLE 5.1. Consider typing this example λ-term:

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

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. □

6 Other significant issues

This section discusses issues related to expansion and expansion variables. Section 6.1 discusses the composition of substitution and expansion. Section 6.2 discusses issues that arise when imposing algebraic laws on types. Section 6.3 discusses the integration of type constraints with E-variables to help with type inference. Section 6.4 discusses a generalization of expansion to the ! type constructor, which distinguishes between linear and non-linear types.

6.1 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., [13, 44, 43, 49]), 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 [36, 38], 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:

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

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 [7] 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.

6.2 Equalities on types, expansion, and E-variables

Some equalities can be safely imposed and are useful. For example, consider these equalities:

$$\begin{aligned}&(1) \quad e\,(T_1 \cap T_2) = e\,T_1 \cap e\,T_2\\ &(2) \quad e\,\omega = \omega\end{aligned}$$

Because intersection is AC and has ω as its neutral, it can be checked that any substitution S that removes all E-variables from both sides of these equations must make them true. Therefore, these equalities are sensible. Given these equalities, the following further equalities hold:

$$\begin{aligned}[E]\,(T_1 \cap T_2) &= [E]\,T_1 \cap [E]\,T_2\\ [E]\,\omega &= \omega\end{aligned}$$

Imposing the equalities (1) and (2) is very helpful, because they allow writing types in very convenient forms. For example, given an E-variable e, any type can always be split into the form $T = T_1 \cap e\,T_2$, where all the parts of the type T_1 are not governed by e in the sense that any occurrences of e in T_1 are under another E-variable (and hence in a different namespace and not really the same as e) or under $\to$ type constructors. Note that T_1 or T_2 might be ω.

Also, with equalities (1) and (2), each type can be split into what we refer to as its *singular components*. Let $\vec{e}$ range over sequences of E-variables, and extend E-variable application to E-variable sequences so that if $\vec{e} = e_1 \cdots e_n$, then $\vec{e}\,T = e_1 \cdots e_n\,T$. Let $\bar{T}$ range over types of the form a or $T_1 \to T_2$. We refer to a type of the form $\vec{e}\,\bar{T}$ as being *singular*. With these definitions and equalities (1) and (2), any type T can be split into its singular components in the form $T = \vec{e}_1\,\bar{T}_1 \cap \cdots \cap \vec{e}_n\,\bar{T}_n$ for some $n \geq 0$, $\vec{e}_1$, …, $\vec{e}_n$, $\bar{T}_1$, …, $\bar{T}_n$. Note that if $n = 0$ then $T = \omega$.

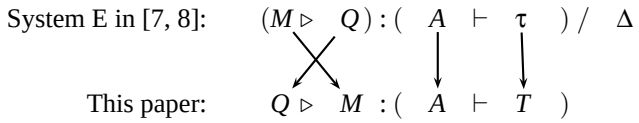
We have made heavy use of these equalities throughout our papers on System E [7, 8]. Without these equalities, we would have had to define a lot of auxiliary machinery to replace them, and the papers would have been much longer and more of a burden for the reader.

When imposing equalities on types such as making $\cap$ AC or the equalities (1) and (2) of this section, one must be quite careful. In general, we find that even if readers are informed of the equalities, they tend to forget and assume they are seeing rigid syntax, which leads them to make logical errors when they try to build on the work. It seems that only long training enables readers to remember that $f(a,b) = f(c,d)$ does not imply that $a = c$ and $b = d$ when f is a well known operator like numeric addition or set union; in contrast readers seem to have much difficulty in remembering this when f is the intersection type

constructor. It is also very easy for authors to make mistakes leading to inconsistencies. For example, it is not uncommon in the literature to find an article where the bulk of the technical material of the article clearly requires either the presence or absence of such equalities, but at various places the authors accidentally write things that are only true in the opposite case. This clearly makes things difficult for the readers.

6.3 E-variables, type inference, and subtyping constraints

The judgements used in this paper differ from judgements in earlier presentations of System E by (1) a different order for the components to improve readability and (2) omitting *constraints*, which are important for type inference. For comparison:



In these other papers, each System E judgement has a *constraint* Δ. If Δ is *solved* (i.e., all the subtyping statements in Δ are true), then the judgement is like a traditional typing judgement. Unsolved constraints are to be solved by type inference. The typing rules do not actually use any subtyping relation. Instead, they merely accumulate subtyping constraints. The user of the typing rules can then choose with which subtyping relation they wish to interpret the constraints. So far System E has been used with 3 distinct subtyping relations which give it different properties, as can be seen in figure 1.

Constraints also share structure with other entities in System E:

$$\Delta ::= e\,\Delta \mid \Delta_1 \cap \Delta_2 \mid \omega \mid (T_1 \leq T_2)$$

By reusing common cases, expansion and substitution application are trivially extended to constraints:

$$[S]\,(T_1 \leq T_2) = ([S]\,T_1 \leq [S]\,T_2)$$

Thus, the problem of type inference is merely to find some substitution S that solves a constraint Δ, i.e., it makes it true in the subtyping relation the user has chosen. E-variables make the process of solving constraints simpler than with previous techniques, because (1) they make it obvious when to trigger expansion and what E-variable to use to obtain the right effect, and (2) they precisely and robustly identify the effect of expansion on types, constraints, and skeletons. Carlier and Wells [8] have a type inference algorithm following this approach.

6.4 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 $\mathtt{0} = \lambda f.\, \lambda x.\, x$, $\mathtt{1} = \lambda f.\, \lambda x.\, f\,@\,x$, $\mathtt{2} = \lambda f.\, \lambda x.\, f\,@\,(f\,@\,x)$, 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 $\mathtt{0}$ 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 $\mathtt{2}$, but not to any other Church numeral.

In general, typings are more precise when the types are linear. In particular, linear types allow expressing number of uses with types. Consider doing usage analysis for a call-by-need language such as Haskell for the term $M = (\lambda x.\, f\,@\,x\,@\,x)\,@\,(y\,@\,z)$. These skeletons derive the following typing judgements:

$$\begin{array}{l} Q_1 = (\lambda x.\, f^{:b\to c\to d}\,@\,x^{:b}\,@\,x^{:c})\,@\,((y^{:a\to b}\,@\,z^{:a}) \cap (y^{:a\to c}\,@\,z^{:a})) \\ Q_2 = (\lambda x.\, f^{:b\to c\to d}\,@\,x^{:b}\,@\,x^{:c})\,@\,(y^{:a\to b\cap c}\,@\,z^{:a}) \\ Q_1 \triangleright M : \langle (f : b \to c \to d, y : (a \to b) \cap (a \to c), z : a \cap a) \vdash e \rangle \\ Q_2 \triangleright M : \langle (f : b \to c \to d, y : a \to b \cap c, z : a) \vdash e \rangle \end{array}$$

If we interpret $a \to b \cap c$ as the type of a function being called once, and $(a \to b) \cap (a \to c)$ as the type of a function being called twice, then Q_1 corresponds to a call-by-name analysis of M (during its evaluation, y is applied twice to z), while Q_2 corresponds to a call-by-need analysis of M (y is applied once to z, and the result of the computation is used twice). If all types are fully non-linear, as they are in the BCD system, then $a \to b \cap c = (a \to b) \cap (a \to c)$ and the distinction is lost. Note also that if strict types were used, the type $a \to c \cap d$ would not be allowed, and only the call-by-name typing could be expressed.

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 [46], so type inference has the same cost as evaluation. This is illustrated by a type inference algorithm of Carlier and Wells [8] 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 $\mathtt{2}$ has no linear typings below rank 2, though it is simply typable. Clearly, this is unsatisfactory.

System E [7] 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 [33].

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, constraints, 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:

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

7 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 [30, 32], which contains neither E-variables nor types, but a germ of the later idea. Expansion variables first appeared in "Beta-reduction as unification" [31], but were still restricted to "type schemes" used during unification and did not yet appear officially in the type system, or even in "expansions".[8]

Kfoury and Wells later proposed System I [36, 38], 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 [6] to add ω as a type and an expansion. Kfoury, Washburn and Wells [34] 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 [41], 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 [7] 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 typing constraints used during type inference. 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 is 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.

8 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 [6] with ω, the nullary case of intersection, and later by Carlier, Polakow, Wells and Kfoury [7] with substitution application and non-linearity. *Expansion variables* can be used to implement expansion in a simple, clean and flexible way.

[8] At this point, the connection between expansion variables and the earlier concept of "expansion" was not yet understood, as illustrated by this (mistaken) quote [31, footnote 3]: " 'Expansions' in this paper are unrelated to 'expansions' as defined in various articles by researchers at the University of Turin ..."

8.1 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 [7], 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 [8], 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.

8.2 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 $\cap$.

In earlier work, there seems to have been an implicit understanding that, even if $\cap$ is not idempotent in a particular type system, it was acceptable to use a set-based model that assumed $\cap$ was idempotent. This was not completely unreasonable, because the type systems where $\cap$ is not idempotent were being used mainly for type inference for systems with idempotent $\cap$ of which they were restrictions, in the sense that every derivation in the non-idempotent system could be mapped cleanly into the idempotent system. However, this understanding is no longer reasonable when ! is added, because the typing derivations in systems with ! and non-idempotent $\cap$ can no longer be mapped cleanly into the type systems for which models have been built, except by completely erasing !. Given that one of the main points of adding ! is to make distinctions between terms with different resource usage behaviors, it seems very important that the denotational semantics should also make these distinctions, in order to justify practical use of the analysis results obtained with systems like System E.

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

9 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] A. Bakewell, A. J. Kfoury. Unification with expansion variables. Technical report, Department of Computer Science, Boston University, 2004.

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

[3] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4), 1983.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[5] G. Boudol, P. Zimmer. On type inference in the intersection type discipline. Draft available from 1st author's web page, 2004.

[6] S. Carlier. Polar type inference with intersection types and ω. In ITRS '02 [25].

[7] S. Carlier, J. Polakow, J. B. Wells, 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*, vol. 2986 of *LNCS*. Springer-Verlag, 2004.

[8] S. Carlier, 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 [9].

[9] S. Carlier, 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., 2004. Completely superseded by [8].

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

[11] M. Coppo, M. Dezani-Ciancaglini. A new type-assignment for lambda terms. *Archiv für Mathematische Logik*, 19, 1978.

[12] M. Coppo, M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4), 1980.

[13] M. Coppo, M. Dezani-Ciancaglini, B. Venneri. Principal type schemes and λ-calculus semantics. In J. R. Hindley, J. P. Seldin, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[14] M. Coppo, P. Giannini. Principal types and unification for simple intersection type systems. *Inform. & Comput.*, 122(1), 1995.

[15] F. Damiani. A conjunctive type system for useless-code elimination. *Math. Structures Comput. Sci.*, 13, 2003.

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

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

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

[19] V. Danos, H. Herbelin, L. Regnier. Game semantics and abstract machines. In *Proc. 11th Ann. IEEE Symp. Logic in Comput. Sci.*, 1996.

[20] M. Dezani, R. Meyer, Y. Motohama. The semantics of entailment omega. *Notre Dame J. Formal Logic*, 43(3), 2002.

[21] P. Giannini, F. Honsell, S. Ronchi Della Rocca. Type inference: Some results, some problems. *Fund. Inform.*, 19(1/2), 1993.

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

[23] C. Haack, J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, vol. 2618 of *LNCS*. Springer-Verlag, 2003. Superseded by [24].

[24] C. Haack, J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50, 2004. Supersedes [23].

[25] *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.*

[26] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[27] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[28] T. Jim. A polar type system. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, J. B. Wells, eds., *ICALP Workshops 2000: Proceedings of the Satellite Workshops of the 27th International Colloquium on Automata, Languages, and Programming*, vol. 8 of *Proceedings in Informatics*, Geneva, Switzerland, 2000. Carleton Scientific.

[29] A. J. Kfoury. Beta-reduction as unification. A refereed extensively edited version is [31]. This preliminary version was presented at the Helena Rasiowa Memorial Conference, 1996.

[30] A. J. Kfoury. A linearization of the lambda-calculus. A refereed version is [32]. This version was presented at the Glasgow Int'l School on Type Theory & Term Rewriting, 1996.

[31] A. J. Kfoury. Beta-reduction as unification. In D. Niwinski, ed., *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*. Springer-Verlag, 1999. Supersedes [29] but omits a few proofs included in the latter.

[32] A. 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).

[33] A. J. Kfoury, H. G. Mairson, F. A. Turbak, J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*. ACM Press, 1999.

[34] A. J. Kfoury, G. Washburn, J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In ITRS '02 [25]. The ITRS '02 proceedings appears as vol. 70, issue 1 of *Elec. Notes in Theoret. Comp. Sci.*

[35] A. J. Kfoury, J. B. Wells. New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. In *Proc. 10th Ann. IEEE Symp. Logic in Comput. Sci.*, 1995.

[36] A. J. Kfoury, 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.*, 1999. Superseded by [38].

[37] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [36], 2003.

[38] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3), 2004. Supersedes [36]. For omitted proofs, see the longer report [37].

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

[40] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17, 1978.

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

[42] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, vol. 19 of *LNCS*. Springer-Verlag, 1974.

[43] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2), 1988.

[44] S. Ronchi Della Rocca, B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2), 1984.

[45] P. Sallé. Une extension de la théorie des types en λ-calcul. In G. Ausiello, C. Böhm, eds., *Fifth International Conference on Automata, Languages and Programming*, vol. 62 of *LNCS*. Springer-Verlag, 1978.

[46] É. Sayag, M. Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, 1996.

[47] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Structures Comput. Sci.*, 7(4), 1997.

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

[49] S. J. van Bakel. Principal type schemes for the strict type assignment system. *J. Logic Comput.*, 3(6), 1993.

[50] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, vol. 2380 of *LNCS*. Springer-Verlag, 2002.

[51] J. B. Wells, A. Dimock, R. Muller, F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, 1997. Superseded by [52].

[52] J. B. Wells, A. Dimock, R. Muller, F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 12(3), 2002. Supersedes [51].

[53] J. B. Wells, C. Haack. Branching types. In *Programming Languages & Systems, 11th European Symp. Programming*, vol. 2305 of *LNCS*. Springer-Verlag, 2002. Completely superseded by [54].

[54] J. B. Wells, C. Haack. Branching types. *Inform. & Comput.*, 200X. Completely supersedes [53]. Accepted subject to revisions.