

Inferring Intersection Typings that are Equivalent to Call-by-Name and Call-by-Value Evaluations^{*}

Adam Bakewell¹, Sébastien Carlier², A. J. Kfoury¹, and J. B. Wells²

¹ Boston University, {bake,kfoury}@cs.bu.edu

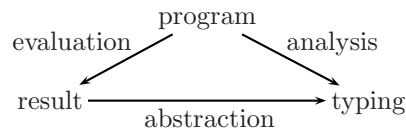
² Heriot-Watt University, {sebc,jbw}@macs.hw.ac.uk

Abstract. We present a procedure to infer a typing for an arbitrary λ -term M in an intersection-type system that translates into exactly the call-by-name (resp., call-by-value) evaluation of M . Our framework is the recently developed System E which augments intersection types with *expansion variables*. The inferred typing for M is obtained by setting up a unification problem involving both type variables and expansion variables, which we solve with a confluent rewrite system. The inference procedure is *compositional* in the sense that typings for different program components can be inferred in *any* order, and without knowledge of the definition of other program components.³ Using expansion variables lets us achieve a compositional inference procedure easily. Termination of the procedure is generally undecidable. The procedure terminates and returns a typing iff the input M is normalizing according to call-by-name (resp., call-by-value). The inferred typing is *exact* in the sense that the exact call-by-name (resp., call-by-value) behaviour of M can be obtained by a (polynomial) transformation of the typing. The inferred typing is also *principal* in the sense that any other typing that translates the call-by-name (resp., call-by-value) evaluation of M can be obtained from the inferred typing for M using a substitution-based transformation.

1 Background and Motivation

1.1 Exact Intersection Typings

This paper is about a method for typing inference centering around the following relationships:



^{*} Work partly funded by NSF grant CCR-0113193 *Implementing Modular Program Analysis via Intersection and Union Types*.

³ Type inference, let alone *typing* inference (which is a stronger notion), in Standard ML is *not* compositional: for a program $\text{let } x = M \text{ in } N$, the type-checker first infers a type for M , based on which it infers a type for N , then a type for the whole program. Parallel separate analysis, i.e., typing inference, for M and N is impossible.

A *typing* combines a result *type* with a type *environment* to specify logical restrictions on the input variables, output and free variables of the program.

Typings viewed as program abstractions have many uses in program validation and compilation. For example, “non-standard” type systems derive typings that support compiler optimisations such as *dead-code analysis* (by detecting parts that will definitely not be evaluated) and as *strictness analysis* (by detecting parts that will definitely be evaluated). The accuracy of these typing analyses, and hence the accuracy of the validation or efficacy of the optimisation, depends on the formalism of types and the capabilities of its inference procedures. Another important factor is the ability of a type system to model the behaviour of the program evaluation strategy.

Intersection type systems (since [10, 4]) offer polymorphism by listing all the usage types of subterms. (Rather than abstracting over different types as in quantifier-based type systems). This can be interpreted to give precisely the sort of usage information sought by the aforementioned analyses. They can also assign types to programs beyond the reach of the most powerful quantifier-based systems such as System F_ω . Systems of quantifier-based polymorphic types (e.g. Hindley-Milner and System F) by contrast lose the information needed to make fine distinctions between the different ways subprograms are used.

Many existing intersection type inference techniques only support analysis to a modest (albeit useful) degree of accuracy. A typical restriction is the *rank-2* types, e.g., [12] for strictness and [11] for deadcode (the rank hierarchy stratifies types according to the depth of nesting of the intersection constructor). Higher rank types indicate “higher order sharing”, so features of programs with a more sophisticated structure are undetectable to many analyses. It is the sheer difficulty of defining simple and highly accurate procedures for intersection typing analysis has held back the use of intersection typings in practical systems.

We present a typing inference procedure that provides a perfectly accurate (“exact”) analysis of evaluation behaviour — the abstraction arrow in our program analysis triangle is really a “lossless” translation. Naturally, program behaviour depends on the evaluation strategy; here, programs are terms of the λ -calculus and we infer typings for their behaviour under the call-by-name (CBN hereafter) and call-by-value (CBV hereafter) evaluation strategies. Our typings are expressed in a system of intersection types, augmented with *expansion variables*. We claim that our approach is straightforward, and show that it enjoys many useful properties.

1.2 Running Example

To illustrate the benefits of intersection typings, the differences between exact analyses for different strategies, and compositional analysis, we consider the following λ -terms (the visible application constructor @ aids clarity and graphical presentation).

$$M_a = (\lambda x. x @ x) @ M_b \qquad M_b = (\lambda y. \lambda z. V_c) @ V_d$$

The two unspecified subterms V_c and V_d could be any *values* not containing y or z as free variables. Neither CBN nor CBV evaluate subterms inside V_c

and V_d . The key evaluation difference is that CBN does not evaluate M_b before substituting it into $\lambda x. x @ x$, and CBV does.

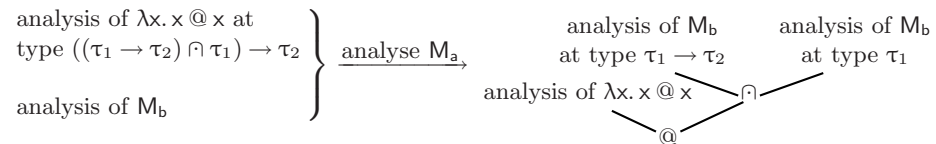
1.3 Compositional Typing Inference

Type systems can aid separate compilation by supporting *smartest recompilation* [20]: code whose source does not change should not need re-analysis. This demands *compositional* type inference. Following the designations of [6], a subprogram should be *intra-checked* based on its definition alone (as noted in footnote 3, SML type inference fails to meet this standard); then *inter-checking* — i.e., link-time, or even “live update-time” analysis — validates whole programs based on subprogram analysis results only. Recent research applies these notions to Java-like languages to solve problems such as minimal recompilation [1].

We do not separate intra and inter-checking, but we do provide a truly compositional analysis that produces its results without any knowledge of unavailable subprograms, can be specialised using the analysis results of missing subprograms when they become available, and permits analysis of subprograms (always leading to the same result) in any order. A key point is that compositional typing analyses are built on the notion of *principal typings* [22]. In essence, such analyses infer a most general typing from which all other typings can be obtained by a substitution-based transformation.

1.4 Expansion Variables

Our inference technique is based around the concepts of *expansions* and *expansion variables*. With reference to our example, the central task in compositional intersection type inference is in making the transformation depicted below.

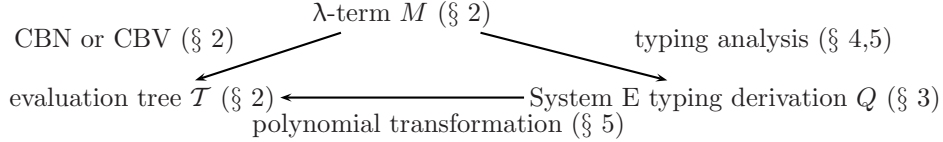


Recognising that the term M_b is to be used at two types, the final analysis shape is formed by: duplicating the analysis of M_b , with each copy distinctly renamed to allow separate treatment, specialising each copy to match the usage types in M_a , and joining them with the \cap constructor. This action was called expansion in early work on intersection types [19, 21].

In System E [7] we conceive typing inference as a transformation from an “unsolved initial analysis” to a “solved final analysis”. Expansion *variables* are introduced in the unsolved analysis (which has the same shape as the term under analysis) at any position where the expansion operation might be needed. The expansion operation itself becomes a special kind of term-cum-substitution. The job of typing inference amounts to replacing expansion variables with the appropriate expansion term to produce an analysis shape that describes how every subterm is used.

Kfoury, Wells and their collaborators have developed the use of expansion variables for type inference over recent years in work that began with the idea of β -unification [14], which introduced a primitive form of expansion variables. An early formulation in System I [18] enabled the inference of principal typings (and thus compositional type inference) for the β -SN terms of the λ -calculus. Expansion in the sense of System E [7] improves, simplifies and extends the capabilities of these older versions in many ways.

1.5 Organization of the Paper and Contributions



- The CBN and CBV λ -calculi, and the System E framework, are summarised.
- A compositional, unification-based, typing inference procedure is presented.
- A polynomial translation mapping inferred typing derivations to CBN or CBV evaluation trees is given, demonstrating how the typing derivations serve as precise analyses of program behaviour in CBN and CBV languages.
- The complete CBN (resp., CBV) typing inference procedure generates a typing derivation for term M that encodes a forest of evaluation trees: the CBN (resp., CBV) tree of M plus the forest of CBN (resp., CBV) evaluation trees of the immediate sub-terms of the result of M .
- The unification procedure and translation are combined to provide an *exact* typing inference procedure that types precisely the terms that are normalizing according to CBN or CBV.
- The inferred CBN (resp., CBV) typing derivations are principal in that any other typing derivation that translates into the CBN (resp., CBV) evaluation tree is a substitution instance of the inferred exact typing derivation.
- Areas for further development are identified. More detail and examples and proofs are available in the accompanying report [2].

2 The Call-by-Name and Call-by-Value λ -Calculi

This section introduces our notations, recalls basic definitions of the λ -calculus and defines CBN and CBV evaluation trees.

Convention 1 (Notation) *Literals (names in sans serif fonts, e.g., M_a, x_1, e, λ and ω) are carefully distinguished from metavariables (names in italic fonts, e.g., M, x, e, \mathcal{D} and τ). Variables h, \dots, q range over the natural numbers. New names may be formed by adding (literal or variable) sub- or super-scripts. \square*

Terms of the λ -calculus are defined in Def. 2. The sort of values are the terms that may result from CBN or CBV evaluation; they are the *weak-head normal forms* plus the *head normal forms* that begin with a variable — for compositional analysis we need to be able to handle *open* programs, so we do not choose to make the popular convenience of the restriction to closed terms.

Def. 2 Term and evaluation tree metavariables, sorts and grammars; result projection.

$x, y, z \in \text{Term-Var} ::= x \mid y \mid z$	$V \in \text{Value} ::= U \mid \lambda x. M$
$U \in \text{Var-HNF} ::= x \mid U @ M$	$M, N \in \text{Term} ::= V \mid M @ N$
$\mathcal{T} \in \text{Eval-Tree} ::= \frac{\mathcal{T}_1 \cdots \mathcal{T}_n}{M \downarrow V}$	$\text{result} \left(\frac{\mathcal{T}_1 \cdots \mathcal{T}_n}{M \downarrow V} \right) = V$

Def. 2 also defines an evaluation tree \mathcal{T} as a tree of evaluation *judgements*, with a *conclusion* judgement at the root (if \mathcal{T} has no sub-evaluations we omit the horizontal line). Each judgement $M \downarrow V$ relates term M to value V . The result projection extracts the conclusion result of an evaluation tree. The purpose of using a *syntax* for evaluation trees is to allow them to be manipulated directly, simplifying our proofs.

The CBN (resp., CBV) evaluation tree of every CBN-normalizing (resp., CBV) term is given by CBN (resp., CBV) in Def. 3. Values have trees consisting of a single judgement: there is no evaluation under λ 's or in the arguments of terms in Var-HNF. (for terms of the form $x @ M_1 @ \cdots @ M_n$ the results of CBx is non-deterministic).

Def. 3 Term CBN and CBV evaluation trees, where $\mathcal{T}_i = \text{CBx}(M_i)$ and $V_i = \text{result}(\mathcal{T}_i)$.

$\text{CBx}(V)$	$= V \downarrow V$
$\text{CBx}(M_1 @ M_2)$	$= \frac{\mathcal{T}_1}{M_1 @ M_2 \downarrow V_1 @ M_2}$ if $V_1 \in \text{Var-HNF}$
$\text{CBN}(M_1 @ M_2)$	$= \frac{\mathcal{T}_1 \quad \mathcal{T}_3}{M_1 @ M_2 \downarrow V_3}$ if $V_1 = \lambda x. M_1$ and $\mathcal{T}_3 = \text{CBN}(M'_1[x := M_2])$
$\text{CBV}(M_1 @ M_2)$	$= \frac{\mathcal{T}_1 \quad \mathcal{T}_2 \quad \mathcal{T}_3}{M_1 @ M_2 \downarrow V_3}$ if $\{V_i = \lambda x. M'_i\}_{i=1}^2$ and $\mathcal{T}_3 = \text{CBV}(M'_1[x := V_2])$

For applications, if the evaluation tree of applied term M does not result in a λ -abstraction then the tree result is a variable head normal form for CBN and CBV. Otherwise, the CBN tree is formed by adding the evaluation tree of the abstraction body specialised with the argument term (resp., the CBV tree adds the evaluation tree of the argument and of the abstraction body specialised with the argument result value; the argument must evaluate to a λ -value because the typing skeleton translation presented later does not work with other values). We assume substitution is defined to avoid variable *capture*, i.e., $(\lambda x. M)[y := M_1] = \lambda x. M$ if $x = y$ and $\lambda x. M[y := M_1[x := z]]$ where z is fresh otherwise, and take term equality to be defined modulo α -conversion as usual. The evaluation trees for our example terms:

$$\begin{aligned} T_{Nb} = \text{CBN}(M_b) &= \frac{\lambda y. \lambda z. V_c \downarrow \lambda y. \lambda z. V_c \quad \lambda z. V_c \downarrow \lambda z. V_c}{M_b \downarrow \lambda z. V_c} \quad V_c \downarrow V_c \\ T_{Na} = \text{CBN}(M_a) &= \frac{\lambda x. x @ x \downarrow \lambda x. x @ x \quad \frac{M_b \downarrow \lambda z. V_c \quad V_c \downarrow V_c}{M_b @ M_b \downarrow V_c}}{M_a \downarrow V_c} \\ T_{Vb} = \text{CBV}(M_b) &= \frac{\lambda y. \lambda z. V_c \downarrow \lambda y. \lambda z. V_c \quad V_d \downarrow V_d \quad \lambda z. V_c \downarrow \lambda z. V_c}{M_b \downarrow \lambda z. V_c} \\ T_{Va} = \text{CBV}(M_a) &= \frac{\lambda x. x @ x \downarrow \lambda x. x @ x \quad T_{Vb} \quad \frac{\lambda z. V_c \downarrow \lambda z. V_c \quad \lambda z. V_c \downarrow \lambda z. V_c \quad V_c \downarrow V_c}{(\lambda z. V_c) @ (\lambda z. V_c) \downarrow V_c}}{M_a \downarrow V_c} \end{aligned}$$

3 Summary of System E

We give a concise formulation of the System E intersection type system framework⁴. Section 3.1 presents the syntax. Section 3.2 presents the key expansion application operation. Section 3.3 presents the typing rules.

3.1 Syntax, Precedence and Equality

Def. 4 defines the syntax. Members of many sorts (on the RHS of the figure) may include pieces of *structure*, these are terms built from the binary intersection constructor and applications of unary *expansion variables* (E-variables). Using the same structure notation in many sorts greatly reduces the definitional overhead. A structure $\mathbb{C}((X_1, \dots, X_n))$ is the result of applying a linear structure context \mathbb{C} with n holes (the \bullet 's) to the n entities X_i .

Def. 4 System E metavariables, sorts and grammars; structure application.

$e, v \in \text{E-Variable} ::= e_n$	$\mathbb{C} \in \text{Structure-Context} ::= \bullet \mid e\mathbb{C} \mid \mathbb{C} \cap \mathbb{C}_1$
$\alpha, v \in \text{T-Variable} ::= a_n$	
$\bar{\tau} \in \text{Type}^{\rightarrow} ::= \alpha \mid \tau \rightarrow \tau_1$	$\tau, \Phi, Y, X \in \text{Type} ::= \mathbb{C}((\tau_1, \dots, \tau_n)) \mid \omega \mid \bar{\tau}$
$S \in \text{ET-Subst} ::= \square \mid v := \Phi, S$	$E, \Phi, Y, X \in \text{Expansion} ::= \mathbb{C}((E_1, \dots, E_n)) \mid \omega \mid S$
$\bar{\Delta} \in \text{Single-Cstr} ::= \tau \leq \tau_1$	$\Delta, Y, X \in \text{Constraint} ::= \mathbb{C}((\Delta_1, \dots, \Delta_n)) \mid \omega \mid \bar{\Delta}$
$Q, X \in \text{Skeleton} ::= x^{i\tau} \mid \lambda x. Q \mid Q @ Q_1 \mid Q^{i\tau} \mid \mathbb{C}((Q_1, \dots, Q_n)) \mid \omega^M$	
For convenience, some of the metavariables range over multiple sorts:	
$v ::= \alpha \mid e$	$\Phi ::= \tau \mid E$
$Y ::= \tau \mid E \mid \Delta$	$X ::= \tau \mid E \mid \Delta \mid Q$
$(\mathbb{C} \cap \mathbb{C}_1)((X_1, \dots, X_{i+j})) = X \cap X_1$ if $X = \mathbb{C}((X_1, \dots, X_i))$ and $X_1 = \mathbb{C}_1((X_{i+1}, \dots, X_{i+j}))$	
$(e\mathbb{C})((X_1, \dots, X_n)) = e\mathbb{C}((X_1, \dots, X_n))$	
$\bullet((X)) = X$	

Types use the binary function type constructor \rightarrow , the empty intersection constructor ω , structure, and type variables (T-variables). *Constraints* are multisets of single constraints encoded as structures containing the constant ω or *single constraints* (these are required equalities in this paper, but \leq is not commutative so we do not use a symbol more suggestive of equality); the empty constraint is ω . *Substitutions* map T-variables to types and E-variables to expansions. *Expansions* are structures whose leaves are all ω or substitutions.

We defined evaluation tree terms to encode term evaluation derivations. *Skeletons* are a similar kind of term that encode System E *typing derivations*. These explicit derivation terms are a key part of our proof techniques.

Convention 5 (Associativity and precedence) *Application* ($@$) is left-associative; constructors (\rightarrow, \cap) are right-associative. Function application ($f(x)$) binds tightest; E-variable application (eX) and expansion application ($[E]X$, Def. 7) bind tighter than (\cap) and substitution in \bar{e} (\bar{e}/S , Convention 8); which bind tighter than $(\rightarrow, @)$; which bind tighter than (\leq, λ) . \square

⁴ Aspects that deal with non-linearity and arbitrary subtyping are omitted.

Examples appear throughout. Here we introduce some substitutions and constraints that are involved in the CBN and CBV typing inference for the example terms. Let τ_c and τ_d stand for the types of the values V_c and V_c under CBN (τ_{V_c} and τ_{V_d} under CBV). Let R' denote a substitution that renames every variable v in Δ_{UNa} to v' . Similarly, R''' renames variables v to v''' .

$$\begin{aligned}
S_{Nb} &= e_3 := \omega, a_4 := \omega \rightarrow \tau_c \\
S_{Na} &= a_1 := \omega \rightarrow \tau'_c, e_2 := R' \cap \omega, a_3 := \tau'_c, a_4 := \omega \rightarrow \tau'_c, \\
&\quad e_1 := \omega, e_3 := \omega, a_5 := \tau'_c \\
S_{Vb} &= e_8 := \omega, a_9 := e_7 (\omega \rightarrow e_6 \tau_{V_c}) \\
S_{Va} &= a_6 := e_4 a_7 \rightarrow a_8, a_{10} := e_5 e_6''' \tau_{V_c}, e_8 := \omega, a_9 := e_5 (\omega \rightarrow e_6''' \tau_{V_c}) \cap \omega, \\
&\quad e_7 := e_5 (R''' \cap \omega), e_5 := e_5 (e_4 := \omega, a_8 := e_6''' \tau_{V_c}) \\
\Delta_{UNb} &= \omega \rightarrow (\omega \rightarrow \tau_c) \leq e_3 \tau_d \rightarrow a_4 \\
\Delta_{UNa} &= (a_1 \leq e_1 a_2 \rightarrow a_3) \cap (a_1 \cap e_1 a_2 \rightarrow a_3 \leq e_2 a_4 \rightarrow a_5) \cap e_2 \Delta_{UNb} \\
\Delta_{UVb} &= \omega \rightarrow e_7 (\omega \rightarrow e_6 \tau_{V_c}) \leq e_8 \tau_{V_d} \rightarrow a_9 \\
\Delta_{UVa} &= (a_6 \leq e_4 a_7 \rightarrow a_8) \cap (e_5 (a_6 \cap e_4 a_7) \rightarrow e_5 a_8 \leq a_9 \rightarrow a_{10}) \cap \Delta_{UVb}
\end{aligned}$$

Skeletons are equal under α -conversion, as are terms. The laws in Def. 6 make \cap associative and commutative (but *not* idempotent as our typings are *exact*) and absorb ω ; E-variables distribute over \cap and ω only.^{5 6}

Def. 6 System E algebraic laws.

$e(X \cap X_1) = eX \cap eX_1$	$X \cap X_1 = X_1 \cap X$	$e\omega = \omega$	$e\omega^M = \omega^M$
$X \cap (X_1 \cap X_2) = (X \cap X_1) \cap X_2$	$\omega \cap Y = Y$	$\omega^M \cap Q = Q$	

For example, $e_5 (a_6 \cap e_4 a_7) = e_5 e_4 a_7 \cap e_5 (e_1 \omega \cap a_6) \cap \omega$.

3.2 Expansion Application

Def. 7 Expansion application, $[-]_-$.

$\boxed{\alpha} \alpha = \alpha$	$[v := \Phi, S] v = \Phi$
$\boxed{\alpha} e = e \boxed{\alpha}$	$[v := \Phi, S] v_1 = [S] v_1$ if $v \neq v_1$
$[S] (e X) = [[S] e] X$	$[e E] X = e [E] X$
$[S] (X \cap X_1) = [S] X \cap [S] X_1$	$[E \cap E_1] X = [E] X \cap [E_1] X$
$[S] \omega = \omega$	$[\omega] Y = \omega$
$[S] \omega^M = \omega^M$	$[\omega] Q = \omega^{\text{term}(Q)}$
$[S] (\tau \rightarrow \tau_1) = [S] \tau \rightarrow [S] \tau_1$	$[S] (\tau \leq \tau_1) = [S] \tau \leq [S] \tau_1$
$[S] x^{\tau} = x^{[S] \tau}$	$[S] \boxed{\alpha} = S$
$[S] \lambda x. Q = \lambda x. [S] Q$	$[S] (v := \Phi, S_1) = (v := [S] \Phi, [S] S_1)$
$[S] (Q @ Q_1) = [S] Q @ [S] Q_1$	$[S] Q^{\tau} = ([S] Q)^{[S] \tau}$

Applying expansion E to entity X , denoted $[E] X$ and defined in Def. 7, gives the “structure-part” of E , including any ω leaves, with each substitution leaf S in E

⁵ These laws are omitted for expansions and skeletons in the original presentation of System E to simplify some proofs. For this work it is better to allow them.

⁶ Allowing E-variables to distribute over \rightarrow or \leq is unsound in the sense of making some entities lose their equivalence after expansion.

replaced by $[S]X$. Applying substitution S to X distributes S over constructors and applies it at variables as usual; at E-variables this either results in no change if $S = \square$, or a new expansion application if S contains a mapping for x . For example, $[S_{\text{Nb}}] \Delta_{\text{UNb}} = [e_3 := \omega, a_4 := \omega \rightarrow \tau_c] (\omega \rightarrow (\omega \rightarrow \tau_c) \leq e_3 \tau_d \rightarrow a_4) = \omega \rightarrow (\omega \rightarrow \tau_c) \leq \omega \rightarrow (\omega \rightarrow \tau_c)$.

Thus E-variables serve as namespace separators: an occurrence of variable v below an E-variable is treated differently to occurrences of v elsewhere, i.e., $e[S]\alpha \neq [S]e\alpha$ in general. Expanding a skeleton Q preserves its *underlying term*; even in the $[\omega]Q$ case. The *expansion composition* operator ‘;’ is defined $E_1; E_2 = [E_2]E_1$ (i.e., E_1 then E_2).

Convention 8 (Substitution abbreviations)

- $v_1 := \Phi_1, \dots, v_n := \Phi_n$ abbreviates $(v_1 := \Phi_1, (\dots, (v_n := \Phi_n, \square) \dots))$;
- e/S abbreviates $(e := eS)$, i.e., S in namespace e ;
- \vec{e}/S abbreviates $e_1/\dots/e_n/S$, i.e., S in namespace $\vec{e} = e_1 \dots e_n$. □

3.3 Type Environments, Skeletons and Typing Rules

- *Type environment* $A : \text{Term-Var} \rightarrow \text{Type}$ maps finitely many variables to non- ω types. $(x_1 : \tau_1, \dots, x_n : \tau_n)$ abbreviates $\{(x_i, \tau_i)\}_{i=1}^n \cup \{(x, \omega) \mid x \notin \{x_i\}_{i=1}^n\}$.
- *Environment intersection* is $A \cap A' = \{(x, A(x) \cap A'(x)) \mid x \in \text{Term-Var}\}$.
- *Expansion application* is $[E]A = \{(x, [E]A(x)) \mid x \in \text{Term-Var}\}$.
- *E-variable application* is $eA = [e\square]A$.
- *Typing* $\langle A \vdash \tau \rangle$ contains type environment A and *result type* τ .
- *Judgement* $M \triangleright Q : \langle A \vdash \tau \rangle / \Delta$ contains term, skeleton, typing and constraint.
- *Valid judgements* are derivable by the rules in Def. 9.

Def. 9 Typing judgement derivation rules.

$\text{(variable)} \frac{}{x \triangleright x^\tau : \langle (x : \tau) \vdash \tau \rangle / \omega} \quad \text{(abstraction)} \frac{M \triangleright Q : \langle A \vdash \tau \rangle / \Delta}{\lambda x. M \triangleright \lambda x. Q : \langle A[x \mapsto \omega] \vdash A(x) \rightarrow \tau \rangle / \Delta}$
$\text{(application)} \frac{M \triangleright Q : \langle A \vdash \tau_1 \rightarrow \tau \rangle / \Delta \quad M_1 \triangleright Q_1 : \langle A_1 \vdash \tau_1 \rangle / \Delta_1}{M @ M_1 \triangleright Q @ Q_1 : \langle A \cap A_1 \vdash \tau \rangle / \Delta \cap \Delta_1}$
$\text{(omega)} \frac{}{M \triangleright \omega^M : \langle () \vdash \omega \rangle / \omega} \quad \text{(result subtyping)} \frac{M \triangleright Q : \langle A \vdash \tau \rangle / \Delta}{M \triangleright Q^{\tau_1} : \langle A \vdash \tau_1 \rangle / \Delta \cap (\tau \leq \tau_1)}$
$\text{(structure)} \frac{\{M \triangleright Q_i : \langle A_i \vdash \tau_i \rangle / \Delta_i\}_{i=1}^n}{M \triangleright \mathbb{C}((Q_1, \dots, Q_n)) : \langle \mathbb{C}((A_1, \dots, A_n)) \vdash \mathbb{C}((\tau_1, \dots, \tau_n)) \rangle / \mathbb{C}((\Delta_1, \dots, \Delta_n))}$

A *valid skeleton* is any Q such that $M \triangleright Q : \langle A \vdash \tau \rangle / \Delta$ is valid; such a Q includes the information contained in the other elements of the judgement, so the following “projections” are defined:

- *term* : Skeleton \rightarrow Term gives the underlying term, $\text{term}(Q) = M$;
- *tenv* : Skeleton \rightarrow Environment gives the typing environment, $\text{tenv}(Q) = A$;
- *rtype* : Skeleton \rightarrow Type gives the result type, $\text{rtype}(Q) = \tau$;
- *constraint* : Skeleton \rightarrow Constraint gives the constraint, $\text{constraint}(Q) = \Delta$.

Term M has typing $\langle A \vdash \tau \rangle$ if there is a valid judgement $M \triangleright Q : \langle A \vdash \tau \rangle / \Delta$ such that $\text{solved}(Q)$. Where solved is defined as follows.

Def. 10 Solved constraints and solved skeletons.

For constraints, $\text{solved}(\mathbb{C}(\tau_1 \leq \tau'_1, \dots, \tau_n \leq \tau'_n))$ iff $1 \leq i \leq n$ implies $\tau_i = \tau'_i$.
 For skeletons, $\text{solved}(Q)$ iff $\text{solved}(\text{constraint}(Q))$.

For example, the following derivation shows that M_b has the typing $\langle () \vdash \omega \rightarrow \tau_c \rangle$. Note that the derived skeleton is isomorphic to the overall derivation, and encodes the same information as all the judgements above it as well as its four projections. Hence skeletons “are” typing derivations.

$$\frac{\frac{\frac{\frac{Q_c \triangleright V_c : \langle () \vdash \tau_c \rangle / \omega}{\lambda z. Q_c \triangleright \lambda z. V_c : \langle () \vdash \omega \rightarrow \tau_c \rangle / \omega}}{\lambda y. \lambda z. Q_c \triangleright \lambda y. \lambda z. V_c : \langle () \vdash \omega \rightarrow (\omega \rightarrow \tau_c) \rangle / \omega}}{(\lambda y. \lambda z. Q_c)^{:\omega \rightarrow \omega \rightarrow \tau_c} \triangleright \lambda y. \lambda z. V_c : \langle () \vdash \omega \rightarrow \omega \rightarrow \tau_c \rangle / \omega} \quad \omega^{V_d} \triangleright V_d : \langle () \vdash \omega \rangle / \omega}}{(\lambda y. \lambda z. Q_c)^{:\omega \rightarrow \omega \rightarrow \tau_c} @ \omega^{V_d} \triangleright M_b : \langle () \vdash \omega \rightarrow \tau_c \rangle / \omega}$$

4 Typing Inference

Typing inference in System E amounts to a unification (constraint solving) problem. Section 4.1 constructs initial, unsolved, skeletons. A rewrite system $\xRightarrow{\text{u}}$ solves the constraints by interleaving simplification (Section 4.2) and part-unifier inference (Section 4.3). The procedure is summarised in Section 4.4.

4.1 Unsolved Typing Skeletons

The relations in Def. 11 specify initial unsolved CBN and CBV skeletons for each term. The intuition for this definition is very simple: in CBN all arguments are wrapped in an E-variable because they will be evaluated once for each usage in the applied function. For example:

$$\begin{aligned} M_b &\xrightarrow{\text{uSkN}} Q_{UNb} = (\lambda y. \lambda z. Q_c)^{:e_3 \tau_d \rightarrow a_4} @ e_3 Q_d \\ M_a &\xrightarrow{\text{uSkN}} Q_{UNa} = (\lambda x. x^{:a_1 : e_1 a_2 \rightarrow a_3} @ e_1 x^{:a_2})^{:e_2 a_4 \rightarrow a_5} @ e_2 Q_{UNb} \end{aligned}$$

$Q_{UNb} =$

$Q_{UNa} =$

In CBV all value arguments and value λ -bodies are wrapped in an E-variable because the values are evaluated once for each usage in the applied function (but

the evaluation work leading up to exposing a value is shared). For example:

$$\begin{aligned}
M_b &\xrightarrow{\text{uSkV}} Q_{UVb} = (\lambda y. e_7 \lambda z. e_6 Q_c)^{e_8 \tau_d \rightarrow a_9} @ e_8 Q_d \\
M_a &\xrightarrow{\text{uSkV}} Q_{UVa} = (\lambda x. e_5 (x^{a_6:e_4} a_7 \rightarrow a_8 @ e_4 x^{a_7}))^{a_9 \rightarrow a_{10}} @ Q_{UVb}
\end{aligned}$$

$Q_{UVb} =$

$Q_{UVa} =$

The side-conditions ensure that introduced variables do not clash with any in the outer namespaces of sub-skeletons.

Def. 11 Relations uSkN and uSkV map terms to unsolved skeletons, where $M_i \xrightarrow{\text{uSk}x} Q_i$.

$x \xrightarrow{\text{uSkN}} x^\alpha$	$x \xrightarrow{\text{uSkV}} x^\alpha$
$\lambda x. M_1 \xrightarrow{\text{uSkN}} \lambda x. Q_1$	$\lambda x. M_1 \xrightarrow{\text{uSkV}} \lambda x. e Q_1$ if $M_1 \in \text{Value}$
$M_1 @ M_2 \xrightarrow{\text{uSkN}} Q_1^{e \text{ rtype}(Q_2) \rightarrow \alpha} @ e Q_2$ if $\alpha, e \notin \text{ovs}(Q_1)$	$\lambda x. M_1 \xrightarrow{\text{uSkV}} \lambda x. Q_1$ if $M_1 \notin \text{Value}$
<i>where the set of variables in the outer namespace of X is given by:</i> $\text{ovs}(X) = \{v \mid \exists \Phi. [v := \Phi] X \neq X\}$	$M_1 @ M_2 \xrightarrow{\text{uSkV}} Q_1^{e \text{ rtype}(Q_2) \rightarrow \alpha} @ e Q_2$ if $\alpha, e \notin \text{ovs}(Q_1)$ and $M_2 \in \text{Value}$
	$M_1 @ M_2 \xrightarrow{\text{uSkV}} Q_1^{\text{rtype}(Q_2) \rightarrow \alpha} @ Q_2$ if $\{\{\alpha\}, \text{ovs}(Q_1), \text{ovs}(Q_2)\}$ are pairwise disjoint and $M_2 \notin \text{Value}$

Unsolved skeleton Q is a valid skeletons of $\text{term}(Q)$. The corollary is that if applying some expansion E solves such a Q then $\text{term}(Q)$ has the typing $\langle \text{tenv}[E] Q \vdash \text{rtype}[E] Q \rangle$.

Proposition 1 ($\text{uSk}x$). If $M \xrightarrow{\text{uSk}x} Q$ then Q is valid and $\text{term}(Q) = M$.

Proof. By structural induction on M . □

4.2 Constraint Simplification

The factor relation (Def. 12) simplifies constraints, distributing the \leq constructors down to be immediately above differing constructors. Common structure is factored when it has the same shape on both sides of a single constraint, if all the types at its leaves are in Type^\rightarrow . For example:

$$\text{factor}(a'_4 \cap e_1 a''_4 \leq (e_1 a_2 \rightarrow a_5) \cap e_1 a_2) = (a'_4 \leq e_1 a_2 \rightarrow a_5) \cap e_1 (a''_4 \leq a_2)$$

Def. 12 Constraint factorisation, factor

$$\begin{array}{l} \text{factor}(\omega) = \omega \\ \text{factor}(\mathbb{C}(\langle \Delta_1, \dots, \Delta_n \rangle)) = \mathbb{C}(\langle \text{factor}(\Delta_1), \dots, \text{factor}(\Delta_n) \rangle) \\ \text{factor}(\bar{\Delta}) = \begin{cases} \text{factor}(\mathbb{C}(\langle \bar{\tau}_1 \leq \bar{\tau}'_1, \dots, \bar{\tau}_n \leq \bar{\tau}'_n \rangle)) & \text{if } \bar{\Delta} \text{ equals } \mathbb{C}(\langle \bar{\tau}_1, \dots, \bar{\tau}_n \rangle) \leq \mathbb{C}(\langle \bar{\tau}'_1, \dots, \bar{\tau}'_n \rangle) \\ \text{factor}(\langle \tau_3 \leq \tau_1 \rangle \cap \langle \tau_2 \leq \tau_4 \rangle) & \text{if } \bar{\Delta} \text{ equals } \tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4 \\ \bar{\Delta} & \text{otherwise} \end{cases} \end{array}$$

The definition is non-deterministic, but in our usage it is immaterial which factorisation is chosen (because all the $\bar{\tau}_i$ are isomorphic — see the discussion of rule (U E-unify) in the next sub-section).

Common \rightarrow constructors are factored by creating two single constraints. The domain types are reversed, consistent with the usual definition of function subtyping. Subtyping is not used in this paper, so the reversal is not essential per se, however, reversal does improve the unification rules (rules (U O-unify) and (U E-unify) in Def. 13 only need to expanded e 's on the left). Solved constraints persist to keep track of all the variable names used in the whole constraint.

Proposition 2 (Solvedness invariant under factoring). *If $\Delta \xrightarrow{\text{factor}} \Delta_1$ then $\text{solved}([E] \Delta)$ iff $\text{solved}([E] \Delta_1)$.*

Proof. By structural induction on Δ . □

4.3 Unification Rules

Def. 13 defines the rewrite system for constraint solving. A direct reduction $\Delta \xrightarrow[S]{\text{u}} \Delta_1$ picks a single constraint in a factorisation of Δ ; then uses a unifier rule to generate the part-unifying substitution S ; factoring $[S] \Delta$ gives the reduct Δ_1 .

Def. 13 Constraint part unification, $\xrightarrow{\text{unifier}}$, direct reduction, $\xrightarrow[S]{\text{u}}$, and reduction, $\xrightarrow[S]{\text{u}} \gg$.

$$\begin{array}{l} \Delta \xrightarrow[S]{\text{u}} \text{factor}([\bar{e}/S] \Delta) \text{ if } \bar{e} \bar{\Delta} \cap \Delta_1 = \text{factor}(\Delta) \text{ and } \bar{\Delta} \xrightarrow{\text{unifier}} S \text{ where:} \\ \tau_1 \leq \tau_2 \xrightarrow{\text{unifier}} \alpha := \tau \quad \text{if } \{\tau_1, \tau_2\} = \{\alpha, \tau\} \quad \text{(U T-unify)} \\ e \tau \leq \omega \xrightarrow{\text{unifier}} e := \omega \quad \text{(U O-unify)} \\ e \tau \leq \tau_1 \xrightarrow{\text{unifier}} e := \mathbb{C}(\langle S_1, \dots, S_n \rangle) \text{ if } \tau_1 \notin \text{T-Variable and } \tau_1 = \mathbb{C}(\langle \bar{\tau}_1, \dots, \bar{\tau}_n \rangle) \text{ (U E-unify)} \\ \text{and } \mathcal{V} \text{ is the set of variables in } \Delta \\ \text{and } f : (\mathcal{V} \times \{1, \dots, n\}) \rightarrow (\text{T-Variable} \setminus \mathcal{V}) \text{ is injective and } S_i(\alpha) = f(\alpha, i) \text{ if } \alpha \in \mathcal{V} \\ \text{and } g : (\mathcal{V} \times \{1, \dots, n\}) \rightarrow (\text{E-Variable} \setminus \mathcal{V}) \text{ is injective and } S_i(e) = g(e, i) \quad \square \text{ if } e \in \mathcal{V} \\ \text{A constraint reduction composes many direct reductions:} \\ \Delta \xrightarrow[S]{\text{u}} \gg \Delta \text{ and } \Delta_1 \xrightarrow[S_1; S_2]{\text{u}} \gg \Delta_3 \text{ if } \Delta_1 \xrightarrow[S_1]{\text{u}} \Delta_2 \text{ and } \Delta_2 \xrightarrow[S_2]{\text{u}} \gg \Delta_3. \end{array}$$

- (U T-unify) substitutions map the T-variable α on one side to the type τ_1 on the other, as in first-order unification. Both directions are needed. No occur check is used or needed (see [3]).
- (U E-unify) substitutions map e to the expansion formed by taking the largest tree of expansion constructors on the right side, with a fresh variable-

Proposition 3 (Reduction). *If $\Delta \xrightarrow[u]{S} \Delta_1$ and $\text{solved}(\Delta_1)$ then $\text{solved}([S]\Delta)$.*

Proof. By induction on the reduction.

1. Case $\Delta \xrightarrow[u]{\square} \Delta$ is immediate.
2. Case $\Delta \xrightarrow[u]{S_1} \Delta_2 \xrightarrow[u]{S_2} \Delta_1$. By I.H., $\text{solved}([S_2] \Delta_2)$. There are $\Delta_3 = \text{simplify}(\Delta)$ and $\Delta_4 = [S_1] \Delta_3$ such that $\text{simplify}(\Delta_4) = \Delta_2$. By Proposition 2, $\text{solved}([S_2] \Delta_2)$ implies $\text{solved}([S_2] \Delta_4)$. Therefore $\text{solved}([S_1; S_2] \Delta_3)$ And by Proposition 2, $\text{solved}([S_1; S_2] \Delta)$. \square

An important detail is that constraint rewriting is global, so when the partly solved constraints of two sub-terms are combined into the constraint of a whole term there must not be any overlap in variable names.

4.4 Typing Inference

Def. 14 CBN and CBV typing inference, inferN and inferV .

$$M \xrightarrow{\text{infer}_x} [S] Q \text{ if } M \xrightarrow{\text{uSk}_x} Q \text{ and } \text{constraint}(Q) \xrightarrow[u]{S} \Delta \text{ and } \text{solved}(\Delta).$$

The typing inference procedure is summarised in Def. 14. We enumerate some of its basic properties. The exact situation regarding completeness, principality and decidability are explained in Section 5.

Proposition 4 (Properties of typing inference).

1. Soundness. *If $M \xrightarrow{\text{infer}_x} Q$ then Q is valid, $\text{term}(Q) = M$ and $\text{solved}(Q)$.*
2. Uniqueness. *If $M \xrightarrow{\text{infer}_x} Q$ and $M \xrightarrow{\text{infer}_x} Q_1$ then Q and Q_1 are isomorphic.*
3. Incompleteness. *The set $\text{Term} \setminus \{ M \mid \exists Q. M \xrightarrow{\text{infer}_x} Q \}$ is not empty.*

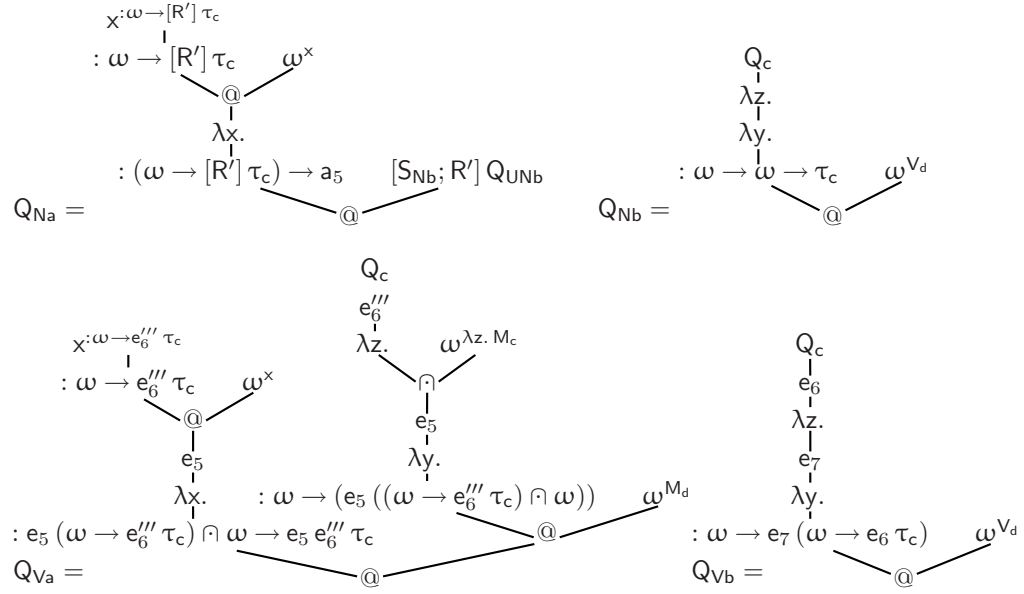
Proof.

1. *Soundness.* Implied by Proposition 1 and Proposition 3.
2. *Uniqueness.* Implied by the confluence result for initial constraints and their reducts in [3].
3. *Incompleteness.* By Proposition 8, if $M \xrightarrow{\text{uSk}_x} Q$ and $M \xrightarrow{\text{infer}_x} Q$ then $\text{toCBx}(Q) = \text{CBx}(M) \bullet Q'$ for some Q' . $\text{CBx}(M)$ cannot exist if M is not normalizing. There are terms that are not normalizing. So there are non-normalising constraints and typing inference is incomplete. \square

For the examples:

$$\begin{aligned} M_b \xrightarrow{\text{inferN}} Q_{Nb} &= [S_{Nb}] Q_{UNb} = (\lambda y. \lambda z. Q_c) : \omega \rightarrow \omega \rightarrow \tau_c @ \omega^{V_d} \\ M_a \xrightarrow{\text{inferN}} Q_{Na} &= [S_{Na}] Q_{UNa} = (\lambda x. x : \omega \rightarrow [R'] \tau_c : \omega \rightarrow [R'] \tau_c @ \omega^x) : (\omega \rightarrow [R'] \tau_c) \cap \omega \rightarrow a_5 \\ &\quad @ [S_{Nb}; R'] Q_{UNb} \cap \omega^{M_b} \\ M_b \xrightarrow{\text{inferN}} Q_{Vb} &= [S_{Na}] Q_{UNa} = (\lambda x. x : \omega \rightarrow [R'] \tau_c : \omega \rightarrow [R'] \tau_c @ \omega^x) : (\omega \rightarrow [R'] \tau_c) \rightarrow a_5 \\ &\quad @ [S_{Nb}; R'] Q_{UNb} \\ M_a \xrightarrow{\text{inferV}} Q_{Va} &= [S_{Vb}] Q_{UVb} = (\lambda y. e_7 \lambda z. e_6 Q_c) : \omega \rightarrow e_7 (\omega \rightarrow e_6 \tau_c) @ \omega^{V_d} \\ M_a \xrightarrow{\text{inferV}} Q_{Va} &= [S_{Va}] Q_{UVa} = (\lambda x. e_5 (x : \omega \rightarrow e_6''' \tau_c : \omega \rightarrow e_6''' \tau_c @ \omega^x)) : e_5 (\omega \rightarrow e_6''' \tau_c) \cap \omega \rightarrow e_5 e_6''' \tau_c \\ &\quad @ (\lambda y. e_5 (\lambda z. e_6''' Q_c \cap \omega \lambda z. M_c)) : \omega \rightarrow (e_5 ((\omega \rightarrow e_6''' \tau_c) \cap \omega)) @ \omega^{M_d} \end{aligned}$$

Pictorially:



5 From Analysis to Evaluation

This section completes the term-evaluation-typing triangle. Section 5.1 presents a notion of substitution used in Section 5.2 to transform solved skeletons into evaluation trees. Section 5.3 considers further properties.

5.1 Linear Skeleton Substitution

Skeleton translation uses the following notion of skeleton substitution to rearrange skeletons from their expanded term shape into the shape of parts of the evaluation trees that corresponds to evaluating a specialised λ -body.

Def. 15 Linear skeleton substitution, $[- := _]$.

$\omega^N[x := \omega^M]$	$= \omega^{N[x:=M]}$	
$x^\tau[x := Q]$	$= Q$	if $\text{rtype}(Q) = \tau$
$y^\tau[x := \omega^M]$	$= y^\tau$	if $y \neq x$
$(\lambda y. Q_1)[x := Q]$	$= \lambda y. (Q_1[x := Q])$	
$(Q_1 \textcircled{\wedge} Q_2)[x := Q]$	$= (Q_1[x := Q'_1])^\tau \textcircled{\wedge} (Q_2[x := Q'_2])$	if $Q = Q'_1 \cap Q'_2$ and $\text{tenv}(Q_i)(x) = \text{rtype}(Q'_i)$
$(\mathbb{C}((Q_1, \dots, Q_n)))[x := Q]$	$= \mathbb{C}((Q_1[x := Q'_1], \dots, Q_n[x := Q'_n]))$	if $Q = \mathbb{C}((Q'_1, \dots, Q'_n))$ and $\text{tenv}(Q_i)(x) = \text{rtype}(Q'_i)$

Skeleton substitution $Q_1[x := Q_2]$ is ill-defined for arbitrary Q_1 and Q_2 . Well-defined instances are identified by Proposition 5; in our usage, Q_1 is the

skeleton of an applied term and Q_2 the skeleton of the argument (or its value).
For example:

$$x:\omega \rightarrow [R'] \tau_c : \omega \rightarrow [R'] \tau_c @ \omega^x [x := [S_{Nb}; R'] Q_{UNb} \cap \omega^{M_b}] = [S_{Nb}; R'] Q_{UNb} : \omega \rightarrow [R'] \tau_c @ \omega^{M_b}$$

$$\begin{array}{c}
x:\omega \rightarrow e_6''' \tau_c \\
\omega \rightarrow e_6''' \tau_c \quad \omega^x \\
\textcircled{\omega} \\
e_5
\end{array}
\left[\begin{array}{c}
Q_c \\
e_6''' \\
x := \lambda z. \omega^{\lambda z}. M_c \\
\textcircled{\omega} \\
e_5
\end{array} \right] =
\begin{array}{c}
Q_c \\
e_6''' \\
\lambda z. \omega^{\lambda z}. M_c \\
\textcircled{\omega} \\
e_5
\end{array}$$

$$\begin{array}{c}
x:\omega \rightarrow e_6''' \tau_c [x := \lambda z. e_6''' Q_c] \\
\omega \rightarrow e_6''' \tau_c \quad \omega x [x := \omega \lambda z. M_c] \\
\textcircled{\omega} \\
e_5
\end{array}
=
\begin{array}{c}
Q_c \\
e_6''' \\
\lambda z. \omega^{\lambda z}. M_c \\
\textcircled{\omega} \\
e_5
\end{array}$$

A key property of skeleton substitution is that — unlike term substitution — it preserves the overall application count; this is why we call it linear. The $\text{app}\#$ function counts the applications in its argument.

Def. 16 Application size of skeleton, $\text{app}\#$.

$\text{app}\#(Q_1 : \tau @ Q_2) = 1 + \sum_{i=1}^2 \text{app}\#(Q_i)$	$\text{app}\#(\lambda x. Q) = \text{app}\#(Q)$
$\text{app}\#(\mathbb{C}((Q_1, \dots, Q_n))) = \sum_{i=1}^n \text{app}\#(Q_i)$	$\text{app}\#(\omega^M) = 0$
$\text{app}\#(x : \tau) = 0$	

Proposition 5 (Skeleton substitution).

If $\text{tenv}(Q_a)(x) = \mathbb{C}(\text{rtype}(Q_1), \dots, \text{rtype}(Q_n))$ and $Q_b = \mathbb{C}((Q_1, \dots, Q_n))$ is valid then:

1. there is a Q such that $Q = Q_a[x := Q_b]$;
2. $\text{term}(Q) = \text{term}(Q_a)[x := \text{term}(Q_b)]$;
3. $\text{app}\#(Q) = \text{app}\#(Q_a) + \text{app}\#(Q_b)$.

Proof. By structural induction on Q_a .

- Case ω^N .
Then $Q_b = \omega^M$ and 1. $Q = \omega^{N[x:=M]}$; 2. $\text{term}(Q) = N[x := M]$; 3. $\text{app}\#(Q_b) = \text{app}\#(Q) = \text{app}\#(Q_a) = 0$.
- Case $x : \tau$.
1. $Q = Q_b$; 2. $\text{term}(Q) = x[x := \text{term}(Q_b)]$; 3. $\text{app}\#(Q_a) = 0$ and $\text{app}\#(Q) = 0 + \text{app}\#(Q_b)$.
- Case $y : \tau$ and $y \neq x$.
Then $Q_b = \omega M$ and 1. $Q = Q_a$; 2. $\text{term}(Q) = y[x := \text{term}(Q_b)]$; 3. $\text{app}\#(Q_b) = 0$ and $\text{app}\#(Q) = \text{app}\#(Q_a) + 0$.

- Case $\lambda y. Q'_a$.
Then $\text{tenv}(Q_a)(x) = \text{tenv}(Q_1)$ by the assumption of capture-free substitution and the proposition holds for $Q' = Q'_a[x := Q_b]$ by I.H. And 1. $Q = \lambda y. Q'$; 2. $\text{term}(Q) = \lambda y. \text{term}(Q') = \text{term}(Q_a)[x := \text{term}(Q_b)]$; 3. $\text{app}\#(Q) = 0 + \text{app}\#(Q') = 0 + \text{app}\#(Q'_a) + \text{app}\#(Q_b) = \text{app}\#(Q_a) + \text{app}\#(Q_b)$.
- Case $Q'_a{}^{:\tau} @ Q''_a$. Then $\text{tenv}(Q_a) = \text{tenv}(Q'_a) \cap \text{tenv}(Q''_a)$ by the definition of tenv . Thus $Q_b = Q'_b \cap Q''_b$ and $\text{tenv}(Q'_a) = \text{rtype}(Q''_b)$ and the proposition holds for $Q' = Q'_a[x := Q'_b]$ by I.H.; similarly for Q'', Q''_a, Q''_b . And 1. $Q = Q'^{:\tau} @ Q''$; 2. $\text{term}(Q) = \text{term}(Q'_a[x := Q'_b]^{:\tau}) @ \text{term}(Q''_a[x := Q''_b]) = (\text{term}(Q'_a[x := \text{term}(Q'_b)]) @ (\text{term}(Q''_a[x := \text{term}(Q''_b)]) \text{term}(Q_a)[x := \text{term}(Q_b)])$; 3. $\text{app}\#(Q) = 1 + \text{app}\#(Q') + \text{app}\#(Q'') = 1 + \text{app}\#(Q'_a) + \text{app}\#(Q'_b) + \text{app}\#(Q''_a) + \text{app}\#(Q''_b) = \text{app}\#(Q_a) + \text{app}\#(Q_b)$.
- Case $\mathbb{C}((Q_a^1, \dots, Q_a^m))$ is similar to the previous case. \square

5.2 Translation of Skeletons

A solved CBN (or CBV) skeleton for M is actually a linear encoding of a CBN (or CBV) evaluation \mathcal{T} and a solved skeleton for the result of \mathcal{T} .

- The term $\mathcal{T} \bullet Q$ is a *Derivation-result pair* if $\text{term}(Q) = \text{result}(\mathcal{T})$.

The rules in Def. 17 translate certain part-solved skeletons into derivation-result pairs. The translation of values is immediate. For applications where the result skeleton of Q is an abstraction, the body is specialised by a linear skeleton substitution to replace each free x in Q_3 with the appropriate part of the argument skeleton Q_1 ; the resulting skeleton is translated to form the second premise of the derivation.

Def. 17 Skeleton to derivation-result pair toCBx transformations, where $M_i = \text{term}(Q_i)$ and $\text{toCBx}(Q_i) = \mathcal{T}_i \bullet Q'_i$.

$\text{toCBx}(Q_1)$	$= M_1 \downarrow M_1 \bullet Q_1$	<i>if</i> $M_1 \in \text{Value}$
$\text{toCBx}(Q_1 \text{:rtype}(Q_1) @ Q_2)$	$= \frac{\mathcal{T}_1}{M_1 @ M_2 \downarrow M'_1 @ M_2} \bullet Q'_1 \text{:rtype}(Q'_1) @ Q_2$	<i>if</i> $Q'_1 \neq \lambda x. Q_3$
$\text{toCBN}(Q_1 \text{:rtype}(Q_1) @ Q_2)$	$= \frac{\mathcal{T}_1}{M_1 @ M_2 \downarrow M_4} \bullet Q_4$	$\left\{ \begin{array}{l} \text{if } Q'_1 = \lambda x. Q_3 \text{ and} \\ \text{toCBN}(Q_3[x := Q_2]) = \mathcal{T}_4 \bullet Q_4 \end{array} \right.$
$\text{toCBV}(Q_1 \text{:rtype}(Q_1) @ Q_2)$	$= \frac{\mathcal{T}_1}{M_1 @ M_2 \downarrow M_4} \bullet \vec{e} Q_4$	$\left\{ \begin{array}{l} \text{if } Q'_1 = \lambda x. \vec{e} Q_3 \text{ and } Q'_2 = \vec{e} Q \\ \text{and } \text{toCBV}(Q_3[x := Q]) = \mathcal{T}_4 \bullet Q_4 \end{array} \right.$

Proposition 6 (Soundness). 1. *If* $Q \xrightarrow{\text{toCBx}} \mathcal{T} \bullet Q'$ *and* $M = \text{term}(Q)$ *then* $M \xrightarrow{\text{CBx}} \mathcal{T}$ *and* $\text{term}(Q') = \text{result}(\mathcal{T})$.

2. *If* $M \xrightarrow{\text{inferx}} Q$ *then there are* \mathcal{T}, Q' *such that* $Q \xrightarrow{\text{toCBx}} \mathcal{T} \bullet Q'$.

Proof.

1. By structural induction on M .
 - Case $M \in \text{Value}$. Then $\mathcal{T} = M \downarrow M$ and $Q' = Q$.

- Case $M = M_1 @ M_2$ and $Q = Q_1 @ Q_2$. For M_i by I.H., $M_i \xrightarrow{\text{CB}x} T_i$ and $\text{term}(Q'_i) = \text{result}(T_i)$.
If $Q'_1 = \lambda x. Q_3$, $\text{term}(Q_2) = M_2$ implies $\text{term}(Q_3[x:=Q_2]) = \text{term}(Q_3)[x:=M_2]$ (Proposition 5) and the result follows by I.H.
If $Q'_1 \neq \lambda x. Q_3$, then $T = \frac{T_1}{M \downarrow M'_1 @ M_2}$ and $Q' = Q'_1 @ Q_2$.
- 2. By structural induction on M . Note that $\text{term}(Q) = M$ as expansion preserves the term projection of all valid skeletons, skeletons generated by $\text{uSk}x$ are valid and Q is a substitution instance of such a skeleton.
 - Case $M \in \text{Value}$. Then $T = M \downarrow M$ and $Q' = Q$.
 - Case $M = M_1 @ M_2$ and $Q = [S](Q_1 @ Q_2)$ where $M_i \xrightarrow{\text{infer}x} Q_i$ (it is valid to assume this order of analysis by the confluence of constraint reduction) and $Q_1 \xrightarrow{\text{toCB}x} T_1 \bullet Q'_1$ by I.H.
 - If $Q'_1 = \lambda x. Q_3$ and $x = \mathbf{N}$, then $Q_2 = e Q'_2$ and $S(e) = E$, there is a $Q_4 = [S] Q_3[x:=E] Q'_2$ (Proposition 5) and $\text{term}(Q_3)[x:=M_2] \xrightarrow{\text{infer}\mathbf{N}} Q_4$ and the result follows by I.H.
 - If $Q'_1 = \lambda x. Q_3$ and $x = \mathbf{V}$, then $Q_2 \xrightarrow{\text{toCBV}} T_2 \bullet e Q'_2$ and $S(e) = E$, there is a $Q_4 = [S] Q_3[x:=E] Q'_2$ (Proposition 5) and $\text{term}(Q_3)[x:=M_2] \xrightarrow{\text{infer}\mathbf{V}} Q_4$ and the result follows by I.H.
 - If $Q'_1 \neq \lambda x. Q_3$, then $T = \frac{T_1}{M \downarrow M'_1 @ M_2}$ and $Q' = [S](Q'_1 @ Q_2)$.

□

Proposition 7 (Complexity). *Computing $\text{toCB}x(Q)$ is polynomial in $\text{app}\#(Q)$.*

Proof. Each translation step involves either constant work; or, it consumes one application in Q and translates a skeleton containing one less application, the work of a linear skeleton substitution is polynomial in the application size of Q . Thus overall the work is bounded by the number of applications multiplied by this polynomial. □

5.3 Completeness and Principality

Here we consider the sense in which $\text{infer}x$ is complete and produces a *most general* typing skeleton. The definition of $\text{toCB}x$ indicates that a solved skeleton for M encodes more than just the evaluation tree of M ; it also includes a solved skeleton for the result value of M . A solved $\text{CB}x$ skeleton encodes a $\text{CB}x$ evaluation “forest” for M : the evaluation tree of M and the evaluation forests of the immediate subterms of the value of M .

We reach this result via consideration of the following weakened inference procedure to give an *exact* CBN or CBV typing skeleton by solving the initial constraint just enough to allow the skeleton to be transformed. Note that $\text{exact}x$ is sound in the sense of producing a skeleton for M that translates to a $\text{CB}x$ evaluation tree for M by definition.

Def. 18 Exact CBN and CBV typing inference, $\text{exact}\mathbf{N}$ and $\text{exact}\mathbf{V}$.

$M \xrightarrow{\text{exact}x} Q \quad \text{if } M \xrightarrow{\text{uSk}x} Q \text{ and } \text{constraint}Q \xrightarrow{S} \text{constraint}(Q')$ $\text{and } \text{toCB}x([S]Q) = T \bullet Q' \text{ and } \text{result}(T) \xrightarrow{\text{uSk}x} Q'$

If M has an evaluation tree \mathcal{T} under $\text{CB}x$ then there is a skeleton Q that translates to \mathcal{T} ; moreover, exact inference with $\text{exact}x$ relates M to a most general translatable skeleton of M .

Proposition 8 (Exact inference complete and most general).

If $M \xrightarrow{\text{CB}x} \mathcal{T}$ and $M \xrightarrow{\text{uSk}x} Q$ then (1) there is a Q_1 such that $M \xrightarrow{\text{exact}x} Q_1$; and (2) $[S]Q \xrightarrow{\text{toCB}x} (\mathcal{T} \bullet Q_2)$ implies there is an S_1 such that $[S]Q = [S_1]Q_1$.

Proof. By structural induction on M . Let e, S_2, Q_3 denote the entities such that by the definition of $\text{exact}x$: $Q_1 = [S_2]Q$; and $\text{result}(\text{derivar}) \xrightarrow{\text{uSk}x} Q_3$; and $\text{constraint}(Q) \xrightarrow[\text{u}]{S_2} \text{constraint}(Q_3)$ if $x = \text{N}$ or $\text{term}(Q_3) \neq \lambda x.M'$, or $\text{constraint}(Q) \xrightarrow[\text{u}]{S_2} e \text{constraint}(Q_3)$ if $x = \text{V}$ and $\text{term}(Q_3) = \lambda x.M'$.

– Case $M \in \text{Value}$.

Then $Q_1 = Q = Q_3$ and $S_1 = S$ and $S_2 = \square$.

– Case $M = M' @ M''$ and $M' \xrightarrow{\text{CB}x} \mathcal{T}'$.

By I.H. there are Q'_1, S'_1 s.t. $M' \xrightarrow{\text{exact}x} Q'_1$ and $[S']Q' \xrightarrow{\text{toCB}x} (\mathcal{T}' \bullet Q'_2) \Rightarrow [S']Q' = [S'_1]Q'_1$.

If $\text{result}(\mathcal{T}') \neq \lambda x.M'''$ and $Q = Q'^{\text{r}} @ Q'b'$ then $Q_1 = [S']Q$ and $S_1 = S'_1$ and $Q_3 = Q_3'^{\text{r}} @ Q_3''$.

If $\text{result}(\mathcal{T}') = \lambda x.M'''$ then by I.H., $\lambda x.M''' \xrightarrow{\text{uSk}x} x @ Q'''$ and $\text{rtype}(x @ Q''') = \mathbb{C}((\tau_1^x, \dots, \tau_n^x)) \rightarrow \tau''''$.

Either CBN / CBV with $M'' \in \text{Value}$ and $Q = Q':e \text{rtype}(Q'') \rightarrow \alpha @ e Q''$

and $\text{constraint}(Q) \xrightarrow[\text{u}]{S'_2} \text{constraint}(Q'_3) \cap e \text{constraint}(Q'') \cap \text{rtype}(x @ Q''') \leq e \text{rtype}(Q'') \rightarrow \alpha = \Delta$ and $S'_2 = \square$ and $S = S'_2; S'_2; S'_1$. (Note that $Q'_3 = \lambda x.Q'''$).

Or CBV with $M'' \notin \text{Value}$ and there are Q''_1, E''_1 s.t. $M'' \xrightarrow{\text{exact}x} Q''_1$ and $[E'']Q'' \xrightarrow{\text{toCB}x} (\mathcal{T}'' \bullet Q''_2) \Rightarrow [E'']Q'' = [E''_1]Q''_1$. and $Q''_1 \xrightarrow{\text{toCB}x} \mathcal{T}'' \bullet e Q''_3$ and $Q = Q':\text{rtype}(Q'') \rightarrow \alpha @ Q''$

and $\text{constraint}(Q) \xrightarrow[\text{u}]{S'_2; S'_2} \text{constraint}(Q'_3) \cap e \text{constraint}(Q''_3) \text{rtype}(x @ Q''') \leq e \text{rtype}(Q''_3) \rightarrow \alpha = \Delta$ and $S = S'_2; S'_2; S'_1$ (in this case we assume that $S'_1(e) = S'_2$).

Let $S'''' = e := \mathbb{C}((R_1, \dots, R_n), \vec{e}_1/\alpha_1 := [R_1] \text{rtype}(Q''_3), \dots, \vec{e}_n/\alpha_n := [R_n] \text{rtype}(Q''_3))$

where $\Delta \xrightarrow[\text{u}]{S''''} \text{constraint}()$

By structural induction on Q'''' , if $\text{toCB}x([S'_1]Q''''[x := [S'_1(e)]Q''_3]) = \mathcal{T}_3 \bullet Q_2$ then $S'_1 = S''''; S''''$ for some S'''' .

By Proposition 5, $M'''' = \text{term}(Q'_3)[x := \text{term}(Q''_3)] \xrightarrow{\text{uSk}x} Q'''' = [S''''']Q''''[x := [S'''''](e)]Q''_3$

By I.H., there are $S_1''''', S_2''''', Q_3'''''$ s.t. $M'''' \xrightarrow{\text{exact}x} [S_2''''']Q'''''$; and $[S''''']Q'''' \xrightarrow{\text{toCB}x} (\mathcal{T} \bullet Q_2''''') \Rightarrow [S''''']Q'''' = [S_1''''']Q_1'''''$.

Overall, $S_2 = S'_2; S'_2; S''''; S_2'''''$ and $S_1 = S_1'''''$. \square

Proposition 9 (Undecidability of exact typing inference). *Membership of $\{M \mid \exists Q.M \xrightarrow{\text{exact}x} Q\}$ is undecidable.*

Proof. By reduction of the undecidable problem of term normalization under CBx . The CBx -normalizing terms are precisely the terms typable by $exactx$ by Proposition 6 and Proposition 8. \square

Similarly, typability with $inferN$ corresponds to β -normalization because it is the *normal order* or *leftmost-outermost* reduction strategy, guaranteed to terminate for normalizing terms. A similar result is conjectured for $inferV$, but we do not give a proof because it does not really correspond to any well-studied strategy. In particular, $inferV$ does not correspond to strong normalization because it can ignore certain unused non-normalizing subterms.

Returning to the notion of evaluation forests, the following partial function translates some solved skeletons into sets of evaluation trees.

Def. 19 Skeleton CBN and CBV forest translations, $forestN$ and $forestV$.

$$forestx(Q) = \{T\} \cup \begin{cases} forestx(Q_1) & \text{if } toCBx(Q) = T \bullet \lambda x. Q_1 \\ \bigcup_{i=1}^n forestx(Q_i) & \text{if } toCBx(Q) = T \bullet x @ Q_1 @ \dots @ Q_n \end{cases}$$

For example, if $V_c = \lambda x. x$ and $Q_c = \lambda x. x^{a_{11}}$ so $solved(Q_c)$ and $\tau_{Nc} = a_{11} \rightarrow a_{11}$, then $forestN([S_{Na}] Q_{UNa}) = \{T_{Na}, x \downarrow x\}$.

Proposition 10 (Inferred skeletons most general). *If $forestx(Q) = \{T_i\}_{i=1}^n$ and $term(Q) = M$ there are Q', S such that $M \xrightarrow{inferx} Q'$ and $Q = [S] Q'$.*

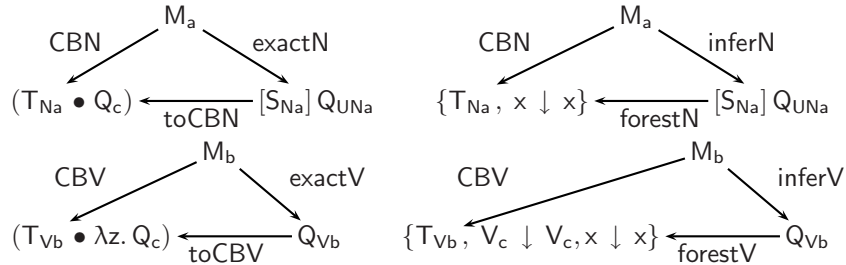
Proof. By structural induction on the sequence of evaluation trees. By the definition of $exactx$ and its completeness (Proposition 8), there are $Q_1, Q_2, Q_3, S_1, S_2, i$ such that $M \xrightarrow{exactx} Q_1$ (where $M \xrightarrow{uSkx} Q_2$ and $Q_1 = [S_1] Q_2$ and $Q_1 \xrightarrow{toCBx} T_i \bullet Q_3$. and $Q = [S_2] Q_2$ and $term(Q_3) = result(CBx(M)) = M_3$.

Case $M_3 = \lambda x. M_4$ and $Q_3 = \lambda x. Q_4$, then $Q_4 \xrightarrow{forestx} \{T_j \mid j \in \{1, \dots, n\} \setminus \{i\}\}$ and by I.H. on Q_4 , there are S'_1, S'_2 such that $S_2 = S'_1; S'_2$; and $S = S'_2$ and $Q = [S] S_1; S'_1 Q_2$.

Case $M_3 \in \text{Var-HNF}$ is similar; a special case is when $n = 1$, $solved(Q_3)$ and $Q' = Q_1$ and $S = S_2$. \square

6 Discussion

Summary. Once the mechanism of expansion is understood, the inference procedure and the relationships between terms, skeletons and evaluations is very simple. The following diagrams summarise the example.



Unsolved skeletons have the shape of the term; applications are interpreted as constraints; constraint solving morphs the skeleton shape by expanding argument (or value) skeletons for each of their uses; the final skeleton is a rearrangement of the evaluation tree and a skeleton for the result, which clearly shows how to interpret a solved skeleton as an analysis of CBN or CBV behaviour, and hence how typing skeletons can be utilized in program analysis. For example, terms typed at ω are deadcode; function terms are strict in their function-typed arguments.

Comparable Approaches. This is the first paper to match exact intersection typing inference so closely to the CBN and CBV strategies. The System I inference procedure [18] gives similar results to *inferN*, but restricted to the β -SN terms and lacking many benefits of the System E approach to expansion. An earlier System E inference procedure [8] offers close results to *inferN*, but lacks the confluence property that makes this work suitable for compositional analysis. The typing inference procedure by Boudol and Zimmer [5] corresponds to a reduction strategy for the β -SN terms in which all discarded sub-terms are evaluated, so it is not CBN or CBV; their method is not based on expansion variables.

Duality of CBN and CBV. Other researchers have studied the differences between the CBN and CBV strategies, the general observation being that they are, in various formal senses, duals. In our case the distinction is expressed in a particularly striking manner. The exact inference procedures can be defined in such a way that the only difference between CBN and CBV is in the choice of *uSkN* or *uSkV*. Thus we can view the whole procedure after unsolved skeleton construction as defining a generic evaluation mechanism — an interpreter, or compiler combined with an abstract machine, which uses the same rules for both strategies. Alternatively, exactly the same analysis procedure produces the same sort of analysis results tailored to CBN or CBV depending only on the outcome of unsolved skeleton construction.

Our encoding of CBV leaves some questions unanswered. Specifically, how to handle applications when the argument evaluates to a $U \in \text{Var-HNF}$.

Exact and Approximate Analysis. Generating exact analyses by a simple unification procedure is an important step that has been enabled by the development of expansion variables. A next step is to develop techniques for compositional *approximate* analysis. This may require the invention of new forms of expansion and there is also the question of how to control the level of detail. Historically, typings have been stratified by *ranks* (e.g., in [15]). Developing this approach will allow our analysis to be used in real implementations to test the benefits that higher-rank typing analysis can bring to compiler optimisations, type safety checking, and type-based security analysis.

Other Evaluation Strategies. System E can express typings that are isomorphic to call-by-need evaluations; an inference procedure is yet to be developed. Extending to more optimal strategies that share under λ 's is an interesting challenge that may go beyond the capabilities of System E.

References

1. D. Ancona and E. Zucca. Principal typings for Java-like languages. In *Conf. Rec. POPL '04: 31th ACM Symp. Princ. of Prog. Langs.*, 2004.
2. A. Bakewell, S. Carlier, A. J. Kfoury, and J. B. Wells. Inferring intersection typings that are equivalent to call-by-name and call-by-value evaluations. Technical report, Church Project, Boston University, Apr. 2005.
3. A. Bakewell and A. J. Kfoury. Properties of a rewrite system for unification with expansion variables. Technical report, Church Project, Boston University, Apr. 2005.
4. F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Inform. & Comput.*, 119:202–230, 1995.
5. G. Boudol and P. Zimmer. On type inference in the intersection type discipline. Draft available from 1st author's web page, Feb. 2004.
6. L. Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pp. 266–277, 1997.
7. S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*, pp. 294–309. Springer-Verlag, 2004.
8. S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β -reduction. In *Proc. 6th Int'l Conf. Principles & Practice Declarative Programming*, 2004. Completely supersedes [9].
9. S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β -reduction. Technical Report HW-MACS-TR-0012, Heriot-Watt Univ., School of Math. & Comput. Sci., Jan. 2004. Completely superseded by [8].
10. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ -calculus semantics. In J. R. Hindley and J. P. Seldin, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 535–560. Academic Press, 1980.
11. F. Damiani and F. Prost. Detecting and removing dead-code using rank 2 intersection. In *TYPES*, pp. 66–87, 1996.
12. T. Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.
13. A. J. Kfoury. Beta-reduction as unification. A refereed extensively edited version is [14]. This preliminary version was presented at the Helena Rasiowa Memorial Conference, July 1996.
14. 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*, pp. 137–158. Springer-Verlag, 1999. Supersedes [13] but omits a few proofs included in the latter.
15. A. J. Kfoury, G. Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In *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.*
16. A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999. Superseded by [18].
17. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [16], Aug. 2003.
18. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. Supersedes [16]. For omitted proofs, see the longer report [17].
19. S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, Jan. 1984.
20. Z. Shao and A. Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.
21. S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. Ph.D. thesis, Catholic University of Nijmegen, 1993.
22. J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, vol. 2380 of *LNCS*, pp. 913–925. Springer-Verlag, 2002.