

Type Inference with Expansion Variables and Intersection Types in System E and an Exact Correspondence with β -Reduction*

Sébastien Carlier
Heriot-Watt University

<http://www.macs.hw.ac.uk/~sebc/>

J. B. Wells
Heriot-Watt University

<http://www.macs.hw.ac.uk/~jbw/>

ABSTRACT

System E is a recently designed type system for the λ -calculus with intersection types and *expansion variables*. During automatic type inference, expansion variables allow postponing decisions about which non-syntax-driven typing rules to use until the right information is available and allow implementing the choices via substitution.

This paper uses expansion variables in a unification-based automatic type inference algorithm for System E that succeeds for every β -normalizable λ -term. We have implemented and tested our algorithm and released our implementation publicly. Each step of our unification algorithm corresponds to exactly one β -reduction step, and *vice versa*. This formally verifies and makes precise a step-for-step correspondence between type inference and β -reduction. This also shows that type inference with intersection types and expansion variables can, in effect, carry out an arbitrary amount of partial evaluation of the program being analyzed.

Categories and Subject Descriptors

D.3.1 [Programming Languages]: Formal Definitions and Theory; F.4.1 [Theory of Computation]: Mathematical Logic—*Lambda calculus and related systems*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*Program analysis*.

General Terms

Algorithms, languages, theory.

Keywords

Lambda-calculus, type inference, intersection types, expansion variables.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP '04, 2004-08-24/---26, Verona, Italy.

Copyright 2004 ACM 1-58113-819-9/04/0008 ...\$5.00.

1. DISCUSSION

1.1 Background and Motivation

1.1.1 Types for Programs and Type Inference

Types have been used extensively to analyze computer program properties without executing the programs, for purposes such as detecting programming errors, enforcing abstract inter-module interfaces, justifying compiler optimizations, and enforcing security properties.

In the *type assignment* style, a type system associates each untyped (i.e., free of type annotations) term (e.g., computer program fragment) with 0 or more *typings*, where each typing is a pair of a result type and a type environment for free term variables. *Type inference* is finding a typing to assign to an untyped term, if possible. Type inference provides the benefits of types while relieving programmers from having to manually supply the types.

Type inference algorithms generally solve constraints, often by *unification*, the process of computing a *solution* (if one exists) that assigns values to variables so that constraints become solved. Unification-based type inference is usually understandable and often efficiently implementable. The most widely used type inference algorithm is Milner’s \mathcal{W} for the well known Hindley/Milner (HM) system [27], used in languages such as SML, OCaml, and Haskell.

The amount of polymorphism provided by HM is equivalent to unfolding all **let**-expressions (a form of partial evaluation) and then doing type inference with simple types. HM-style polymorphism is not implemented by unfolding **let**-expressions because (1) this is difficult to combine with separate compilation and (2) it is very inefficient as it re-analyzes **let**-expressions every time they are used. HM has simple and efficient (for typical cases) type inference and supports separate compilation provided module interfaces are given. However, HM is somewhat inflexible and does not support *compositional analysis*, i.e., analyzing modules without knowledge about other modules.

1.1.2 Intersection Types

Intersection types were introduced for the λ -calculus by Coppo and Dezani [8] and independently by Pottinger [29]. In both cases, a major motivation was the connection between β -reduction and intersection types, which makes intersection types well suited for program analysis.

Intersection type systems have been developed for many kinds of program analysis (e.g., flow [1], strictness [18], dead-code [13, 10], and totality [7]) usable for justifying compiler

optimizations to produce better machine code. Intersection types seem to have the potential to be a general, flexible framework for many program analyses.

In most intersection type systems, every typable term has a *principal typing* [37], one that is stronger than all other typings assignable to that term. Principal typings improve the possibilities of analyzing programs incrementally and minimizing reanalysis cost after program modifications.

1.1.3 Type Inference for Full Intersection Types

The first type inference algorithms for intersection type systems were by Coppo, Dezani and Venneri [9], and Ronchi della Rocca and Venneri [32]. These algorithms give principal typings for β -normal forms, and it is separately proven that a principal typing for a term's β -normal form is also principal for the term. One disadvantage of this approach is that terms must be reduced to β -normal form before types are found for them. Another disadvantage is that getting other typings from principal typings in this approach uses operations such as *substitution*, *expansion*, and *lifting*, and the older expansion definitions are hard to understand.

Ronchi della Rocca [31] devised the first unification-based type inference algorithm for a non-rank-restricted intersection type system. This algorithm yields principal typings when it succeeds. Because typability for the full system is undecidable, the algorithm is of course sometimes non-terminating; to ensure termination, one can restrict type height. This algorithm uses the older, complicated definition of expansion, which has several disadvantages: (1) the unification procedure requires global knowledge in addition to the two types to unify, which makes the technique delicate to extend; (2) expansion is only defined on types and typings, but not on typing derivations (needed for use in compiler optimizations), which are thus not straightforward to obtain via type inference; (3) the older notion of expansion only allows intersection introduction in argument position, and is thus unsuitable for accurate resource tracking under call-by-need and call-by-value evaluation.

Regnier [30] presented an inference algorithm for an intersection type system called *De*. System *De* is similar to that of [31], with the essential addition of *labels* in types to carry out expansion. Using *nets*, an untyped version of proof nets from Linear Logic [14], Regnier showed an exact correspondence between net reduction in Linear Logic and type inference in System *De*. Boudol and Zimmer [4] gave an inference algorithm that implements expansion by attaching a set of type variables to each typing constraint; this appears to be hard to generalize beyond pure λ -calculus. Both the approach of Regnier and that of Boudol and Zimmer are stated to be unification-based, but neither approach produces *unifiers* (substitutions that solve unification constraints); this makes them less practical.

1.1.4 Rank-2 Intersection Types

Leivant [23] introduced a rank-2 intersection type system and remarked that it is similar in typing power to Hindley/Milner. In fact, rank-2 intersection type systems type strictly more terms than Hindley/Milner, although the additional terms are not very significant.

Van Bakel [35] gave the first unification-based type inference algorithm for a rank-2 intersection type system. Jim [19] studied extensions of rank-2 intersection type systems with practical programming features and discussed the impor-

tance of principal typings. Damiani [11] has extended support for conditionals and letrec (mutually recursive bindings) as well as support for a let-binding mechanism that effectively adds some of the power of one more rank to obtain some of the expressiveness of rank-3 intersection types. Damiani [12] has also worked on rank-2 intersection types for symmetric linking of modules.

Restricting intersection types to rank-2 does not use their full potential to analyze programs as precisely as needed, so we do not pursue this idea.

1.1.5 Expansion Variables

Kfoury and Wells [20, 22] gave a type inference algorithm for System I, a full, non-rank-restricted intersection type system. System I introduced *expansion variables* and a single operation integrating expansion and substitution; together, these features vastly simplified expansion.

Unfortunately, System I has several technical limitations. The substitution operation (which contains expansion) has a built-in renaming (needed for complete type inference) of type and expansion variables that prevents substitutions from being composable. An awkward workaround for this problem called *safe composition* was developed, but safe composition is hard to understand and people have been error-prone when working with it. System I has other technical limitations such as non-associative and non-commutative intersections and no weakening; together, these limitations prevent it from having subject reduction.

The recently developed System E [6] improves in many ways on System I. The full System E (only a fraction of its power is needed for this paper) is more flexible than the extremely flexible intersection type system of Barendregt, Coppo, and Dezani [2]. Contrary to Van Bakel's advice to make intersection type systems as lean as possible [36], System E does not restrict where intersection type constructors (and thus also expansion variables) can be used. Flexible expansion variable placement allows expansion variables to establish namespaces. In turn, this allows a substantially simpler way of integrating expansion and substitution. In particular, System E does not need the automatic built-in fresh-renaming done during expansion in System I. As a result, substitution and expansion in System E are more robust than they are in System I and composition works for substitutions and expansions. These improvements make it much simpler to design type inference methods for System E, leading to the new results in this paper.

1.2 Summary of Contributions

1. We present a clear and precise unification-based intersection type inference algorithm in System E and prove that it types any β -normalizing λ -term. We believe our presentation is easier to understand and implement than other presentations of intersection type inference. It is a strength of our algorithm that it needs only the λ -free fragment of System E restricted to trivial subtyping. (Our algorithm works unchanged with these features present.)
2. Our intersection type inference approach provides a simple and clean *step-for-step* β -reduction/unification correspondence. In the main algorithm phase, each step of our unify- β rule on constraints corresponds to exactly one β -reduction step on λ -terms, and *vice versa*. Any β -reduction strategy can be simulated by a strategy of using unify- β , and *vice versa*. Our proof that our algo-

algorithm succeeds for all β -normalizing λ -terms takes advantage of this correspondence and uses the well known β -normalizing leftmost/outermost strategy. This correspondence is impossible for systems without the flexibility of intersection types, because they can not type even all strongly β -normalizing terms [34]. Our correspondence improves over that of Regnier [30] both by being more formal and also by being direct instead of being the composition of two correspondences with untyped *nets* as an intermediate notion.

The clear presentation of this correspondence helps to share the understanding that any intersection type inference approach will be equivalent to *partial evaluation* followed by a monovariant analysis. In effect, intersection type inference can do an arbitrary amount of partial evaluation via type unification, i.e., it can do the equivalent of any amount of β -reduction of the analyzed term.

3. Our intersection type inference approach is the only one to provide a notion of *readback* that allows extracting β -reduced λ -terms from partially solved constraints *during* type inference. This makes it easy to prove the step-for-step β -reduction/unification correspondence and makes the correspondence easier to understand. The use of readback makes our approach clearer than the approaches of Boudol and Zimmer [4] and also Regnier [30].
4. Unlike early algorithms from before 1990, our inference algorithm can build and use expansions based solely on local matching of constraint solving rules without needing global knowledge of the typing inferred so far, because expansion in System E is guided by E-variables.
5. The statements and proofs of properties of our inference algorithm are made clearer and more precise by the use of *skeletons*, compact syntactic representations of typing derivations. Unlike System E, nearly all other intersection type systems do not have skeletons.
6. Our inference algorithm builds a *solution* that solves the input constraints, i.e., a substitution of types for type variables and expansions for E-variables that when applied to the input constraints yields satisfied constraints. This is only possible because the composition of expansions and substitutions is straightforward in System E, which is not true for other systems.
7. Our algorithm can easily be used to get different forms of output such as a full typing derivation or just a typing (result type and type environment) because expansion is defined on all of the mathematical entities in System E, including skeletons. Constructing typing derivations is vital for use in compilers with typed intermediate representations. In contrast, many presentations of other algorithms do not clearly document how to construct typing derivations; this knowledge is either buried deep inside proofs or simply omitted.
8. Our inference algorithm manages analysis polyvariance by a scheme of properly arranging the nesting of a finite number of E-variables. This is simpler than renaming schemes used in other approaches and greatly eases implementation. This is only possible because, unlike previous intersection type systems, System E has no restrictions on where expansion can occur. Expansion in System E can splice intersection constructors into type positions that are to the right of arrows and into typing derivation positions that are not function application arguments.
9. As explained below in sec. 1.3, our algorithm is more

suitable to be extended to have flexible analysis precision and to analyze call-by-need and call-by-value behavior.

10. We have implemented all of the algorithms described in this paper and made them available via a web interface [5] and for downloading (<http://www.macs.hw.ac.uk/DART/software/system-e/>). Our implementation generated the example output found in appendix A.
11. A technical report with full proofs will be made available on the web pages of the authors by 2004-08.

1.3 Ongoing Future Work

The full System E (only part is presented in this paper) has the ! type constructor which allows non-exact analysis. The step-for-step β -reduction/unification correspondence means that type inference can obtain polyvariance by doing the equivalent of any amount of partial evaluation of the analyzed program followed by a cruder, more traditional monovariant analysis that uses ! and subtyping to collapse the analysis. Because of the careful way ! is integrated into the full System E, the analysis results precisely indicate where the information is exact and where it is approximate. This ability for type inference to partially evaluate leads to the potential of analysis that simultaneously is compositional and has easily adjustable cost and precision. (The algorithm presented in this paper always has exact precision and the same cost as normalization.) In contrast, the widely used algorithm \mathcal{W} for Hindley/Milner is non-compositional and does the equivalent of a fixed amount of partial evaluation (unfolding all let-expressions) followed by the standard (monovariant) first-order unification of simple type inference [25]; all information in the results must be assumed to be approximate. Although we have left the full exploration of this promising possibility to future work, this motivation helps justify the significance of this work. Investigating this is ongoing work with our colleague Makhholm.

Because in System E expansion can occur in non-function-argument positions, type inference will be able to do resource-aware analysis of call-by-need and call-by-value evaluation, rather than being applicable only to call-by-name evaluation (which is unused in practice).

We believe the algorithm *Infer* (definition 4.22) finds solutions that yield *principal typings* [37]. Proving this is ongoing work with our colleagues Kfoury and Bakewell.

1.4 Other Future Work

Because types are often exposed to programmers, a major design goal for many type systems has been making types suitable for human comprehension. Unfortunately, this conflicts with making types suitable for accurate and flexible program analysis. Intersection types are good for accurate analysis. However, inferred intersection types may be extremely detailed and thus are likely to be unsuitable for presenting to humans. Alternative type error reporting methods such as *type error slicing* [15, 16] can avoid presenting these types directly to programmers. Investigation is needed to combine type error slicing with System E.

Module boundary interfaces are generally intended to abstract away from the actual software on either side of the interface, so that implementations can be switched. Also, module boundary interfaces must be compact and easily understandable by humans. For these reasons, \forall and \exists quantifiers are appropriate for use in module boundary types. An open problem is how to use very flexible and accurate

types such as intersection types for analysis and then check whether they imply types using \forall and \exists quantifiers.

We (Carrier, Kfoury, and Wells) and also independently Mairson and Neergaard [28] have noticed a correspondence between *solving of type inference constraints* in System I and *reduction of nets*. Intersection types correspond to contraction nodes, ω (introduced in \mathcal{L}_ω by Carrier) to 0-ary contraction nodes (weakening), E-variables to boxes, T-variables to axiom links, and constraints to cut links. Each type inference constraint solving step in System I (using the precise version of the rules of [20]) corresponds to a net reduction step. Expansion describes net transformations, and our unification algorithm does the equivalent of cut-elimination on nets via substitution for E-variables. In System E (but not in System I), the equalities imposed on types and constraints in sec. 3.2 correspond to the flexibility of nets.

This connection with nets has several possible implications. First, the expansion variables of System E offer a syntactic alternative to nets that may be easier for precise reasoning. Second, System E could lead to a Linear Logic extension with intersection as a proof-functional connective [24] (unlike the usual truth-functional connectives). In this new system, Regnier's nets could be annotated with formulas. (Although Mairson [26] earlier suggested that net edges can be annotated with Linear Logic formulas, a counter-example, due to Urzyczyn, is the net for 22K, with $2 = \lambda f. \lambda x. f(fx)$ and $K = \lambda y. \lambda z. y.$)

1.5 Other Related Work

Sayag and Mauny [33] characterize principal typings in intersection type systems and show they are isomorphic to β -normal forms. The correspondence is limited to normal forms and does not directly show the step-for-step correspondence between β -reduction and type inference.

1.6 Acknowledgements

This paper benefited from detailed comments by Adam Bakewell, Assaf Kfoury, Henning Makhholm, and Jeff Polakow and from helpful discussions with Harry Mairson and Peter Neergaard on correspondences between proof nets and type inference with expansion variables.

2. PRELIMINARY DEFINITIONS

This section defines generic mathematical notions. Let i, j, m, n, p , and q range over $\{0, 1, 2, \dots\}$ (the natural numbers). Let $\pi_1(\langle a, b \rangle) = a$ and $\pi_2(\langle a, b \rangle) = b$. Given a function f , let $f[a \mapsto b] = (f \setminus \{(a, c) \mid (a, c) \in f\}) \cup \{(a, b)\}$. Let r range over binary relations. Let \xrightarrow{r} be alternate infix notation for r . Let \xrightarrow{r} be the transitive and reflexive (w.r.t. the intended carrier set) closure of r . Let $r; r'$ be the composition of r and r' , i.e., $r; r' = \{(a, c) \mid \exists b. r(a, b) \wedge r'(b, c)\}$. Given a context C , let $C[U]$ stand for C with the single occurrence of \square replaced by U , e.g., $(\lambda x. \square)[x] = \lambda x. x$.

If S names a set and φ is defined as a metavariable ranging over S , let S^* be the set of *sequences* over S as per the following grammar, quotiented by the subsequent equalities, and let $\vec{\varphi}$ be a metavariable ranging over S^* :

$$\begin{aligned} \vec{\varphi} \in S^* &::= \epsilon \mid \varphi \mid \vec{\varphi}_1 \cdot \vec{\varphi}_2 \\ \epsilon \cdot \vec{\varphi} &= \vec{\varphi}, \quad \vec{\varphi} \cdot \epsilon = \vec{\varphi}, \quad (\vec{\varphi}_1 \cdot \vec{\varphi}_2) \cdot \vec{\varphi}_3 = \vec{\varphi}_1 \cdot (\vec{\varphi}_2 \cdot \vec{\varphi}_3) \end{aligned}$$

For example, \vec{n} ranges over $\{0, 1, 2, \dots\}^*$ (sequences of natural numbers). Length 1 sequences are equal to their sole member; this requires taking some care.

Given an order $<$ on a set X , let the *lexicographic-extension order* $<_{\text{lex}}$ of $<$ be the least relation s.t. for any $\vec{x}, \vec{z} \in X^*$ and $y, y' \in X$ where $y < y'$ both $\vec{x} \cdot y \cdot \vec{z}$ and $\vec{x} \cdot y' \cdot \vec{z}$ hold.

Diagrams illustrate formal statements. A diagram means that for all entities linked to solid lines satisfying the relations attached to the solid lines, the additional entities linked to dashed lines exist satisfying the relations attached to the dashed lines. For example, the following diagram means $\forall m, n, q. (m < n) \wedge (n \leq q) \Rightarrow \exists p. (m \leq p) \wedge (p < q)$:

$$\begin{array}{ccc} m & \xrightarrow{\quad} & n \\ \vdots & & \vdots \\ p & \xrightarrow{\quad} & q \end{array} \leq$$

3. SYSTEM E

This section presents System E. See [6] for full details.

3.1 Syntax

Fig. 1 defines the syntactic entities used in this paper. Note the distinction between the metavariables x, α , and e and concrete variables like x_0, a_1 , and e_2 . The main difference from the original System E definition [6] is that the ! operator is omitted in this paper.

We use @ for application in terms and skeletons; this is non-standard, but we do so to have some syntactic marker for application nodes when terms and skeletons are pretty-printed or drawn as tree.

We define operator precedence, including for ordinary function application ($f(a)$) and modification ($f[a \mapsto b]$), and for later-defined operations like expansion application ($[E]X$) and term-variable substitution ($M_1[x := M_2]$). The precedence groups follow, from highest to lowest:

- group 1: $Q^\tau, f(a), f[a \mapsto b], M_1[x := M_2], v := \Phi, C[M], D[Q], \vec{\varphi}_1 \cdot \vec{\varphi}_2$
- group 2: $eX, [E]X, (\phi, S)$
- group 3: $X_1 \cap X_2, e/S$
- group 4: $\tau_1 \rightarrow \tau_2, M @ N, Q_1 @ Q_2, S_1; S_2$
- group 5: $\tau_1 \leq \tau_2, \lambda x. M, \lambda x. Q,$

As examples of binding conventions, $e\alpha_1 \cap \alpha_2 \rightarrow \alpha_3 = ((e\alpha_1) \cap \alpha_2) \rightarrow \alpha_3$, and $(e\alpha_1 \leq \alpha_2) = ((e\alpha_1) \leq \alpha_2)$, and $\lambda x. x^{\alpha_1} @ y^{\alpha_2} = \lambda x. (x^{\alpha_1} @ y^{\alpha_2})$. As is usual, application is left-associative so that $M_1 @ M_2 @ M_3 = (M_1 @ M_2) @ M_3$ (similarly for skeletons) and function types are right-associative so that $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Let the expression $\tau_1 \cap \dots \cap \tau_n$ denote $\tau_1 \cap (\tau_2 \cap (\dots \cap \tau_n))$ when $n \geq 1$ and ω when $n = 0$. Extending E-variable application to sequences, let $\epsilon X = X$ and $(\vec{e} \cdot e)X = \vec{e}(eX)$.

3.2 Equalities

Terms and skeletons are quotiented by α -conversion as usual [3], where $\lambda x. M$ and $\lambda x. Q$ bind the variable x .

For types and constraints, the definitions of fig. 1 are modified by imposing equalities for E-variable application, the \cap operator, and the ω constant. The \cap operator is associative and commutative with ω as its unit. E-variable application distributes over \cap and ω . (The constant ω is a 0-ary version of \cap .) Formally, these rules hold:

$$\begin{aligned} T_1 \cap (T_2 \cap T_3) &= (T_1 \cap T_2) \cap T_3 \\ T_1 \cap T_2 &= T_2 \cap T_1 & \omega \cap T &= T \\ e(T_1 \cap T_2) &= eT_1 \cap eT_2 & e\omega &= \omega \end{aligned}$$

The sorts and their abstract syntax grammars and metavariables:			
$x \in$	Term-Variable $::= x_i$	$M, N \in$	Term $::= x \mid \lambda x. M \mid M_1 @ M_2$
$\alpha \in$	T-Variable $::= a_i$	$C \in$	Term-Context $::= \square \mid \lambda x. C \mid C @ M \mid M @ C$
$e \in$	E-Variable $::= e_i$	$\tau \in$	Type $::= \tau_1 \cap \tau_2 \mid e \tau \mid \omega \mid \alpha \mid \tau_1 \rightarrow \tau_2$
$\phi \in$	ET-Assignment $::= \alpha := \tau \mid e := E$	$E \in$	Expansion $::= E_1 \cap E_2 \mid e E \mid \omega \mid S$
$S \in$	ET-Substitution $::= \square \mid \phi, S$	$\Delta \in$	Constraint $::= \Delta_1 \cap \Delta_2 \mid e \Delta \mid \omega \mid \tau_1 \leq \tau_2$
		$Q \in$	Skeleton $::= Q_1 \cap Q_2 \mid e Q \mid \omega^M \mid x^{:\tau} \mid \lambda x. Q \mid Q_1 @ Q_2 \mid Q^{:\tau}$
		$D \in$	Skel-Context $::= \square \mid D \cap Q \mid Q \cap D \mid e D \mid \lambda x. D \mid D @ Q \mid Q @ D \mid D^{:\tau}$
Metavariables ranging over subsets or multiple sorts:			
$v ::= e \mid \alpha$	$\Phi ::= E \mid \tau$	$T ::= \tau \mid \Delta$	$X ::= \tau \mid E \mid \Delta \mid Q$
		$Y ::= \tau \mid E \mid \Delta$	$\dot{\Delta} ::= e \dot{\Delta} \mid \bar{\Delta}$
			$\bar{\Delta} ::= \tau_1 \leq \tau_2$

Figure 1: Syntax grammars and metavariable conventions.

$[E_1 \cap E_2] X$	$= [E_1] X \cap [E_2] X$	$[\square] \alpha$	$= \alpha$	$[S] e X$	$= [[S] e] X$
$[e E] X$	$= e [E] X$	$[\square] e$	$= e \square$	$[S] \square$	$= S$
$[\omega] Y$	$= \omega$	$[v := \Phi, S] v'$	$= [S] v'$ if $v \neq v'$	$[S] (v := \Phi, S')$	$= (v := [S] \Phi, [S] S')$
$[\omega] Q$	$= \omega^{\text{term}(Q)}$	$[v := \Phi, S] v$	$= \Phi$	$[S] x^{:\tau}$	$= x^{:[S] \tau}$
$[S] (X_1 \cap X_2)$	$= [S] X_1 \cap [S] X_2$	$[S] \omega$	$= \omega$	$[S] \lambda x. Q$	$= \lambda x. [S] Q$
$[S] (\tau_1 \leq \tau_2)$	$= [S] \tau_1 \leq [S] \tau_2$	$[S] \omega^M$	$= \omega^M$	$[S] (Q_1 @ Q_2)$	$= [S] Q_1 @ [S] Q_2$
$[S] (\tau_1 \rightarrow \tau_2)$	$= [S] \tau_1 \rightarrow [S] \tau_2$	$[S] Q^{:\tau}$	$= ([S] Q)^{:[S] \tau}$		

Figure 2: Expansion application.

Both α -conversion and the additional rules for types and constraints are imposed as *equalities*, where “=” is mathematical equality (as it should be). For example, $\omega \cap \alpha = \alpha$. After this modification, the syntactic sorts no longer form an initial algebra, so care must be taken.

3.3 Operations on Syntax

Let $T \sqsubseteq T'$ iff $T' = T \cap T''$ and let $T \sqsupseteq T'$ iff $T' \sqsubseteq T$.

Let $M[x := N]$ and respectively $M \xrightarrow{\beta} N$ denote the usual notions [3] for untyped λ -terms of term-variable substitution and respectively β -reduction.

Let **term** be the least-defined function such that:

$$\begin{aligned}
\text{term}(x^{:\tau}) &= x \\
\text{term}(\lambda x. Q) &= \lambda x. \text{term}(Q) \\
\text{term}(Q_1 @ Q_2) &= \text{term}(Q_1) @ \text{term}(Q_2) \\
\text{term}(Q^{:\tau}) &= \text{term}(e Q) = \text{term}(Q) \\
\text{term}(\omega^M) &= M \\
\text{term}(Q_1 \cap Q_2) &= \text{term}(Q_1) \quad \text{if } \text{term}(Q_1) = \text{term}(Q_2)
\end{aligned}$$

A skeleton Q is *well formed* iff $\text{term}(Q)$ is defined. E.g., $Q = x^{:\tau_1} \cap y^{:\tau_2}$ is ill-formed if $x \neq y$ since $\text{term}(Q)$ is undefined.

Convention 3.1. Henceforth, only well formed skeletons are considered. \square

Definition 3.2. E-path If $\dot{\Delta} = \bar{e}\bar{\Delta}$, then **E-path**($\dot{\Delta}$) = \bar{e} is the *E-path* of $\dot{\Delta}$. If D is a skeleton context, we define **E-path**(D), the E-path of the context hole in D , by induction, as follows:

$$\begin{aligned}
\text{E-path}(\square) &= \epsilon \\
\text{E-path}(e D) &= e \cdot \text{E-path}(D) \\
\text{E-path}(\lambda x. D) &= \text{E-path}(D^{:\tau}) = \text{E-path}(D @ Q) \\
&= \text{E-path}(Q @ D) = \text{E-path}(D \cap Q) \\
&= \text{E-path}(Q \cap D) = \text{E-path}(D) \quad \square
\end{aligned}$$

3.4 Expansion Application

The essential new notion of System E is the way it uses *expansion variables* (E-variables) to implement expansion.

Expansion is an operation that calculates for a typing judgement the changed judgement that would result from inserting additional typing rule uses at nested positions in the judgement’s derivation. E-variables are a new technology that makes expansion simpler and easier to implement and reason about. E-variables are *placeholders* for unknown uses of other typing rules like \cap -introduction. E-variables are propagated into the types and the type constraints used by type inference algorithms. In System E, expansion operations are described by syntactic terms. The use of E-variables and expansion terms allows defining expansion application in a precise, uniform, and syntax-directed way.

Definition 3.3 (Expansion application). Fig. 2 defines the application of substitutions to E-variables, and of expansions to types, expansions, constraints, and skeletons. \square

EXAMPLE 3.4. E-variables effectively establish namespaces and substituting an expansion for an E-variable can merge namespaces. Define the following:

$$\tau_1 = e_1 a_0 \rightarrow a_0 \quad S_1 = (e_1 := \square, \square) \quad S_2 = (a_0 := \tau_2, \square)$$

Then these facts hold:

$$\begin{aligned}
[S_2] \tau_1 &= e_1 a_0 \rightarrow \tau_2 \\
[S_1] \tau_1 &= a_0 \rightarrow a_0 \\
[S_2] [S_1] \tau_1 &= \tau_2 \rightarrow \tau_2
\end{aligned}$$

In $[S_2] \tau_1$, the T-variable a_0 inside the E-variable e_1 is effectively distinct from the T-variable a_0 outside e_1 , so the substitution only replaces the outer a_0 . The operation $[S_1] \tau_1$ replaces e_1 by the empty expansion (which is actually the identity substitution), and this effectively lifts the inner a_0 into the root namespace, so that $[S_2] [S_1] \tau_1$ replaces both occurrences of a_0 . \square

Lemma 3.5 (Expansion application composition). Given any E_1, E_2, X , $[[E_1] E_2] X = [E_1] [E_2] X$. \square

Let $E_1; E_2 = [E_2] E_1$ (composition of expansions). By lem. 3.5, the “;” operator is associative. Although $E_1; E_2$ is not much shorter than $[E_2] E_1$, it is easier to follow.

$\text{(abstraction)} \frac{(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta}{(\lambda x. M \triangleright \lambda x. Q) : \langle A[x \mapsto \omega] \vdash A(x) \rightarrow \tau \rangle / \Delta}$	$\text{(variable)} \frac{}{(x \triangleright x^\tau) : \langle (x : \tau) \vdash \tau \rangle / \omega}$
$\text{(application)} \frac{(M_1 \triangleright Q_1) : \langle A_1 \vdash \tau_2 \rightarrow \tau_1 \rangle / \Delta_1; \quad (M_2 \triangleright Q_2) : \langle A_2 \vdash \tau_2 \rangle / \Delta_2}{(M_1 @ M_2 \triangleright Q_1 @ Q_2) : \langle A_1 \cap A_2 \vdash \tau_1 \rangle / \Delta_1 \cap \Delta_2}$	$\text{(omega)} \frac{}{(M \triangleright \omega^M) : \langle \text{env}_\omega \vdash \omega \rangle / \omega}$
$\text{(intersection)} \frac{(M \triangleright Q_1) : \langle A_1 \vdash \tau_1 \rangle / \Delta_1; \quad (M \triangleright Q_2) : \langle A_2 \vdash \tau_2 \rangle / \Delta_2}{(M \triangleright Q_1 \cap Q_2) : \langle A_1 \cap A_2 \vdash \tau_1 \cap \tau_2 \rangle / \Delta_1 \cap \Delta_2}$	$\text{(E-variable)} \frac{(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta}{(M \triangleright e Q) : \langle e A \vdash e \tau \rangle / e \Delta}$
$\text{(result subtyping)} \frac{(M \triangleright Q) : \langle A \vdash \tau_1 \rangle / \Delta}{(M \triangleright Q^{\tau_2}) : \langle A \vdash \tau_2 \rangle / \Delta \cap (\tau_1 \leq \tau_2)}$	

Note: When these rules derive judgements of the form $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$ such that $\text{solved}(\Delta)$ holds, then these rules act as typing rules in the traditional sense. (See also remark 3.10.)

Figure 3: Typing rules.

An assignment ϕ may stand for a substitution (ϕ, \square) and is to be interpreted that way if necessary. The higher precedence of $(v := \Phi)$ over (ϕ, S) also applies here. For example, as a substitution $(e_1 := e_2 := S_2, e_3 := S_3)$ stands for $((e_1 := (e_2 := S_2)), e_3 := S_3)$ which can be written in full as $((e_1 := ((e_2 := S_2), \square)), e_3 := S_3)$.

Let e/S stand for $(e := e S)$. Thus, when necessary, e/S stands for $((e := e S), \square)$. The “/” notation builds a substitution that affects variables underneath an E-variable, because $[e/S] e X = e [S] X$. E.g., $S = (e_0 / (a_1 := \tau_1), a_0 := \tau_0)$ stands for $S = (e_0 := e_0 (a_1 := \tau_1, \square), a_0 := \tau_0, \square)$ and in this case it holds that $[S] (e_0 a_1 \rightarrow a_0) = e_0 \tau_1 \rightarrow \tau_0$.

We extend this notation to E-variable sequences so that $\vec{e} \cdot e/S$ stands for $\vec{e}/e/S$ and e/S stands for S .

3.5 Type Environments and Typing Rules

Type environments, ranged over by A and B , are total functions from Term-Variable to Type that map only a finite number of variables to non- ω types.

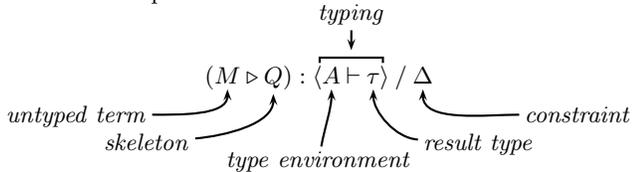
Definition 3.6 (Operations on type environments).

$$\begin{aligned} [E] A &= \{ (x, [E] A(x)) \mid x \in \text{Term-Variable} \} \\ A \cap B &= \{ (x, A(x) \cap B(x)) \mid x \in \text{Term-Variable} \} \\ e A &= \{ (x, e A(x)) \mid x \in \text{Term-Variable} \} \\ \text{env}_\omega &= \{ (x, \omega) \mid x \in \text{Term-Variable} \} \quad \square \end{aligned}$$

Let $(x_1 : \tau_1, \dots, x_n : \tau_n)$ be $\text{env}_\omega[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n]$. Observe, for every e, E_1, E_2, A , and x , that (1) $[E_1 \cap E_2] A = [E_1] A \cap [E_2] A$, (2) $[E] (A \cap B) = [E] A \cap [E] B$, (3) $[e \square] A = e A$, (4) $(e A)(x) = e A(x)$, (5) $[\omega] A = \text{env}_\omega$, and finally (6) $[E] A[x \mapsto \tau] = ([E] A)[x \mapsto [E] \tau]$.

Definition 3.7 (Typing judgements and typing rules).

Fig. 3 gives the typing rules of System E used in this paper. The rules derive *judgements* of the following form with the indicated components:



A pair $\langle A \vdash \tau \rangle$ of a type environment A and a *result type* τ is a *typing*. \square

A skeleton Q is just a *proof term*, a compact notation representing an entire typing derivation. A skeleton Q syntactically represents a tree of typing rule uses that derives

a judgement for the untyped term $\text{term}(Q)$. Using skeletons avoids needing gigantic judgement trees in formal statements. Many type systems use type-annotated λ -terms for this role, but this fails for typing rules like our intersection (\cap -introduction) rule. (See [38, 39] for a discussion.)

The intended meaning of $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$ is that Q is a proof that the untyped term M has the typing $\langle A \vdash \tau \rangle$, provided the constraint Δ is *solved* w.r.t. some subtyping relation. The typing rules do not check whether a constraint Δ is solved to allow (1) using different subtyping relations and (2) using the same rules to generate constraints to be solved by type inference. For a subtyping relation, this paper needs and will use only the weakest, namely equality on types, but other papers on System E use other relations (e.g., $\leq_{\text{nl}}^{\text{lin}}$ and \leq_{flex} from [6]). The results in this paper extend to all subtyping relations that include equality.

Definition 3.8 (Valid skeleton). A skeleton Q is *valid* iff there exist M, A, τ , and Δ s.t. $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$. \square

Convention 3.9. Only valid skeletons are considered. \square

A skeleton Q uniquely determines all components in its judgement. Let *typing*, *constraint*, *tenv*, and *rtype* be functions s.t. $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$ implies $\text{typing}(Q) = \langle A \vdash \tau \rangle$, $\text{constraint}(Q) = \Delta$, $\text{tenv}(Q) = A$, and $\text{rtype}(Q) = \tau$.

A constraint Δ is *solved*, written $\text{solved}(\Delta)$, iff Δ is of the form $\vec{e}_1 (\tau_1 \leq \tau_1) \cap \cdots \cap \vec{e}_n (\tau_n \leq \tau_n)$. Given a judgement $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$, the entire judgement and its skeleton Q are *solved* iff Δ is solved. The *unsolved part* of a constraint Δ , written $\text{unsolved}(\Delta)$, is the smallest constraint Δ_1 such that $\Delta = \Delta_1 \cap \Delta_2$ and $\text{solved}(\Delta_2)$ for some Δ_2 .

REMARK 3.10 (RELATION TO TRADITIONAL TYPING RULES). Solved judgements and skeletons correspond respectively to traditional typing judgements and derivations. Traditional typing rules are merely the special case of the typing rules in fig. 3 where all constraints are solved. When constraints are solved, our use of “typing” for a pair $\langle A \vdash \tau \rangle$ matches the definition in [37]. The rules in fig. 3 can also be used to generate unsolved constraints to be solved by a type inference algorithm, as is done in sec. 4. Our presentation not only saves space by combining the two roles of the rules, but also guarantees in a simple way (stated formally in lem. 3.11) that type inference yields valid typing derivations. \square

Lemma 3.11 (Admissibility of expansion). If $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$, then $(M \triangleright [E] Q) : \langle [E] A \vdash [E] \tau \rangle / [E] \Delta$. \square

4. TYPE INFERENCE

This section presents a type inference algorithm for System E. We show how to infer a typing for any normalizing untyped λ -term M , where each constraint-solving step exactly corresponds to one β -reduction step, starting from M . We also give an algorithm that reconstructs the β -reduced term from each intermediate stage of type inference.

The approach to type inference is as follows. Given an untyped term M as input, (1) pick an initial skeleton Q such that $\text{term}(Q) = M$, (2) do unification to find a substitution S solving $\text{constraint}(Q)$, and (3) use Q and S to calculate a solved skeleton (typing derivation) or a typing for M .

If run on λ -terms which have no normal form, our inference algorithm is non-terminating. This also makes it in some sense incomplete because any term, even if it is non-normalizable, can be typed using the ω typing rule in System E.

4.1 Initial Skeletons

We pick an initial skeleton $\text{initial}(M)$ using three distinct E-variables e_0, e_1 , and e_2 , and one T-variable a_0 , as follows:

$$\begin{aligned} \text{initial}(x) &= x^{:a_0} \\ \text{initial}(\lambda x. M) &= \left(\text{let } Q = e_0 \text{ initial}(M) \right. \\ &\quad \left. \text{in } \lambda x. Q \right) \\ \text{initial}(M @ N) &= \left(\text{let } Q_1 = e_1 \text{ initial}(M) \right. \\ &\quad \left. \begin{array}{l} Q_2 = e_2 \text{ initial}(N) \\ \text{in } Q_1 : \text{rtype}(Q_2) \rightarrow a_0 @ Q_2 \end{array} \right) \end{aligned}$$

We extend this definition to contexts, taking $\text{initial}(\square) = \square$.

EXAMPLE 4.1 (INITIAL SKELETON). Let $M = (\lambda x. x @ x) @ y$, $Q = \text{initial}(M)$:

$$Q = \left(\begin{array}{l} e_1 (\lambda x. e_0 ((e_1 (x^{:a_0}) : e_2 a_0 \rightarrow a_0) @ e_2 (x^{:a_0}))) \\ : e_2 a_0 \rightarrow a_0 \\ @ e_2 (y^{:a_0}) \end{array} \right)$$

We have $(M \triangleright Q) : \langle A \vdash \tau \rangle / \Delta$ where $A = (y : e_2 a_0)$, $\tau = a_0$ and $\Delta = (e_1 (e_0 e_1 a_0 \sqcap e_0 e_2 a_0 \rightarrow e_0 a_0) \leq e_2 a_0 \rightarrow a_0) \sqcap (e_1 e_0 (e_1 a_0 \leq e_2 a_0 \rightarrow a_0))$. \square

We now define *pleasant* skeletons, which syntactically characterize skeletons produced by initial .

Definition 4.2 (Pleasant skeleton). A skeleton Q is *pleasant* iff $Q = P$ for some P defined as follows:

$$P ::= x^{:a_0} \mid \lambda x. e_0 P \mid (e_1 P_1)^{:e_2 \tau \rightarrow a_0} @ e_2 P_2$$

Note that by convention 3.9, if $P = (e_1 P_1)^{:e_2 \tau \rightarrow a_0} @ e_2 P_2$, then $\tau = \text{rtype}(P_2)$. Note that this definition uses 4 specific, fixed concrete E- and T-variables (e_0, e_1, e_2 , and a_0). \square

All constraints of a pleasant skeleton are unsolved, and for all x , $\text{tenv}(P)(x) = \vec{e}_1 a_0 \sqcap \dots \sqcap \vec{e}_n a_0$ for some $n \geq 0$.

Lemma 4.3 (Properties of initial).

1. $\text{initial}(M)$ is a pleasant skeleton.
2. If $P = \text{initial}(M)$ then $\text{term}(P) = M$.
3. If $M = \text{term}(P)$ then $\text{initial}(M) = P$. \square

4.2 Readback

To show an exact correspondence with β -reduction, we define a *readback* function to reconstruct a pleasant skeleton from a typing and a constraint.

Definition 4.4 (Readback). Let readback be the least-defined function where $\text{readback}(A, \tau, \Delta)$ is given by case analysis on τ and subsequently on Δ if $\tau = a_0$, such that:

$$\begin{aligned} \text{readback}((x : a_0), a_0, \omega) &= x^{:a_0} \\ \text{readback}(e_0 A, e_0 \tau_1 \rightarrow e_0 \tau_2, e_0 \Delta) &= \lambda x. e_0 Q \\ &\quad \text{if } A(x) = \omega \\ &\quad \text{and } Q = \text{readback}(A[x \mapsto \tau_1], \tau_2, \Delta) \\ \text{readback}(e_1 A_1 \sqcap e_2 A_2, a_0, \\ &\quad e_1 \Delta_1 \sqcap e_2 \Delta_2 \sqcap (e_1 \tau_1 \leq e_2 \tau_2 \rightarrow a_0)) \\ &= (e_1 Q_1)^{:e_2 \tau_2 \rightarrow a_0} @ e_2 Q_2 \\ &\quad \text{if } Q_1 = \text{readback}(A_1, \tau_1, \Delta_1) \\ &\quad \text{and } Q_2 = \text{readback}(A_2, \tau_2, \Delta_2) \end{aligned}$$

For convenience, we let $\text{readback}(Q) = \text{readback}(\text{tenv}(Q), \text{rtype}(Q), \text{unsolved}(\text{constraint}(Q)))$. \square

EXAMPLE 4.5 (READBACK). Let $A = (z : e_2 a_0)$, $\tau = a_0$ and:

$$\Delta = \left(\begin{array}{l} (e_1 (e_0 a_0 \rightarrow e_0 a_0) \leq e_2 a_0 \rightarrow a_0) \\ \sqcap (e_1 ((e_0 a_0 \rightarrow e_0 a_0) \rightarrow e_0 a_0 \rightarrow e_0 a_0 \\ \leq (e_0 a_0 \rightarrow e_0 a_0) \rightarrow e_0 a_0 \rightarrow e_0 a_0)) \end{array} \right)$$

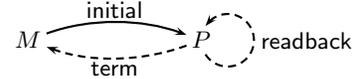
We have:

$$\text{readback}(A, \tau, \Delta) = \left(\begin{array}{l} (e_1 (\lambda x. e_0 (x^{:a_0}) : e_2 a_0 \rightarrow a_0) \\ @ e_2 (z^{:a_0}) \end{array} \right) \quad \square$$

Lemma 4.6 (Readback builds pleasant skeletons). If $\text{readback}(A, \tau, \Delta) = Q$, then $Q = P$ for some P . \square

Lemma 4.7 (Readback is identity on pleasant skeletons). If $\text{readback}(P) = P'$, then $P' = P$. \square

Combining lemmas 4.3 and 4.7, we obtain this diagram:



Lemma 4.8 (Readback preserves typings). Suppose that $\text{readback}(A, \tau, \Delta) = P$ and $\text{term}(P) = M$. Then it holds that $(M \triangleright P) : \langle A \vdash \tau \rangle / \text{unsolved}(\Delta)$. \square

4.3 Unification Rules

From fig. 1, the metavariable $\dot{\Delta}$ ranges over *singular* constraints, those that contain exactly one type inequality, and $\bar{\Delta}$ ranges over singular constraints that have an empty E-path. Each unification step solves one singular constraint.

Fig. 4 introduces rules $\text{unify-}\beta$ and $\text{unify-}@$, which are used to produce substitutions that solve singular constraints. Observe that these rules mention 4 specific, fixed concrete E- and T-variables (e_0, e_1, e_2, a_0), rather than arbitrary metavariables e and α . (Originally, we made the rules match arbitrary E-variables, but this generality was not actually used and the proof is simpler this way.) We now study each of these two rules in detail.

4.3.1 Rule unify- β

This section shows that rule $\text{unify-}\beta$ solves constraints in a way that corresponds exactly with β -reduction on λ -terms.

Rule $\text{unify-}\beta$ matches on singular constraints of the form $\bar{\Delta} = \vec{e}(e_1 (e_0 \tau_0 \rightarrow e_0 \tau_1) \leq e_2 \tau_2 \rightarrow a_0)$ corresponding to a β -redex $(\lambda x. M) @ N$ in the untyped λ -term. Informally, each portion of $\bar{\Delta}$ corresponds to a portion of the redex: τ_0 to the type of x in M , τ_1 to the result type of M , τ_2 to the result type of N , and a_0 to the result type of the whole redex.

$\vec{e}(e_1 (e_0 \tau_0 \rightarrow e_0 \tau_1) \leq e_2 \tau_2 \rightarrow a_0) \xrightarrow{\text{unify-}\beta} \vec{e} / ((e_2 := e_1 e_0 E, e_1 / e_0 / S); (a_0 := [S] \tau_1, e_1 := e_0 := \square))$ <p style="text-align: center; margin: 0;">if $\tau_0 \xrightarrow{\text{extract}} E$ and $\tau_0 \xrightarrow{:=\tau_2} S$</p>	
$\vec{e}(e_1 a_0 \leq e_2 \tau_1 \rightarrow a_0) \xrightarrow{\text{unify-}\textcircled{\alpha}} \vec{e} / (e_1 := (a_0 := e_2 \tau_2 \rightarrow a_0, e_1 := e_1 e_1 \square, e_2 := e_1 e_2 \square))$	
$\alpha \xrightarrow{\text{extract}} \square$	$\alpha \xrightarrow{:=\tau_2} (\alpha := \tau_2)$
$\tau_1 \cap \tau_2 \xrightarrow{\text{extract}} E_1 \cap E_2 \quad \text{if } \tau_1 \xrightarrow{\text{extract}} E_1 \text{ and } \tau_2 \xrightarrow{\text{extract}} E_2$	$\tau_1 \cap \tau_2 \xrightarrow{:=\tau_2} S_1; S_2 \quad \text{if } \tau_1 \xrightarrow{:=\tau_2} S_1 \text{ and } \tau_2 \xrightarrow{:=\tau_2} S_2$
$e \tau \xrightarrow{\text{extract}} e E \quad \text{if } \tau \xrightarrow{\text{extract}} E$	$e \tau \xrightarrow{:=\tau_2} e / S \quad \text{if } \tau \xrightarrow{:=\tau_2} S$
$\omega \xrightarrow{\text{extract}} \omega$	$\omega \xrightarrow{:=\tau_2} \square$

Figure 4: Rules for solving constraints.

EXAMPLE 4.9 (RULE $\text{unify-}\beta$). Consider the following singular constraint $\bar{\Delta}$ which is part of Δ from example 4.1 (for memory, $M = (\lambda x.x @ x) @ y$):

$$\bar{\Delta} = (e_1 (e_0 e_1 a_0 \cap e_0 e_2 a_0 \rightarrow e_0 a_0) \leq e_2 a_0 \rightarrow a_0)$$

This constraint corresponds to the β -redex in M . We have $\bar{\Delta} \xrightarrow{\text{unify-}\beta} S$, where S is the following substitution:

$$\begin{array}{l} \left\{ \begin{array}{l} e_2 := e_1 e_0 (e_1 \square \cap e_2 \square) \\ e_1 := e_1 (\left\{ \begin{array}{l} e_0 := e_0 (\left\{ \begin{array}{l} e_1 := e_1 (a_0 := a_0, \square), \square \\ e_2 := e_2 (a_0 := a_0, \square), \square \end{array} \right\} \\ \square \end{array} \right. \\ \square \end{array} \right. \\ ; a_0 := a_0, e_1 := e_0 := \square, \square, \square \end{array} \right. \end{array}$$

The first part of S (line 1, $e_2 := e_1 e_0 (e_1 \square \cap e_2 \square)$) makes as many copies of argument y of the β -redex as there are occurrences of the bound variable x . Each copy is put underneath a distinct sequence of E-variables (here e_1 and e_2); each sequence of E-variables corresponds to one occurrence of x . The second part of the substitution S (lines 2-4) replaces the type variable associated with each occurrence of the bound variable x with the result type of the corresponding copy of the argument y (this has no effect in this case, because y also has type a_0). The last part of S (line 6) simultaneously replaces the type variable that holds the result type of the application with the result type of the body of the λ -abstraction (having no effect in this case), and then erases E-variables e_1 and e_0 . Only the E-variable e_1 needs to be erased for the constraint to become solved; erasing e_0 is needed to preserve definedness of readback . \square

Lemma 4.10. If $\text{constraint}(P) = \bar{\Delta} \cap \Delta'$ and $\bar{\Delta} \xrightarrow{\text{unify-}\beta} S$, then $\text{solved}([S] \bar{\Delta})$.

$$\begin{array}{ccc} S & \xleftarrow{\pi_1} & \langle S, \bar{\Delta} \rangle \xrightarrow{[\cdot]; \text{unsolved}} \omega \\ \text{unify-}\beta \uparrow & & \\ \bar{\Delta} & \xleftarrow{\pi_2} & \end{array}$$

The previous lemmas show that $\text{unify-}\beta$, when it applies to some constraint $\bar{\Delta}$ of a pleasant skeleton, produces a substitution S that solves $\bar{\Delta}$. $\bar{\Delta} \xrightarrow{\text{unify-}\beta} S$ implies the following:

$$\begin{array}{l} \bar{\Delta} = e_1 (e_0 \tau_0 \rightarrow e_0 \tau_1) \leq e_2 \tau_2 \rightarrow a_0 \\ \tau_0 = \vec{e}_1 a_0 \cap \dots \cap \vec{e}_n a_0 \\ S = \left\{ \begin{array}{l} (e_2 := e_1 e_0 E, e_1 / e_0 / S') \\ ; (a_0 := [S'] \tau_1, e_1 := e_0 := \square) \end{array} \right. \\ \tau_0 \xrightarrow{\text{extract}} E \quad \text{and} \quad \tau_0 \xrightarrow{:=\tau_2} S', \text{ for some } E \text{ and } S'. \end{array}$$

Let $E' = \vec{e}_1 \square \cap \dots \cap \vec{e}_n \square$ and

$$S'' = \vec{e}_1 / (a_0 := \tau_2); \dots; \vec{e}_n / (a_0 := \tau_2).$$

With these particular E' and S'' , it is easy to see that $[E'] \tau_2 = \vec{e}_1 \tau_2 \cap \dots \cap \vec{e}_n \tau_2 = [S''] \tau_0$ holds, provided that

no \vec{e}_i is a proper prefix of \vec{e}_j , with $i, j \in \{1, \dots, n\}$. This is guaranteed by the hypothesis $\text{constraint}(P) = \bar{\Delta} \cap \Delta'$, since for every i , \vec{e}_i is the E-path of some skeleton variable in P . Equalities of sec. 3.2 are imposed on types, but not on expansions and substitutions, so $E = E'$ and $S' = S''$ do not necessarily hold. However, all possible E and S' have the same effect when they are applied to types. By definition 3.3, $[S] \bar{\Delta} = [S'] \tau_0 \rightarrow [S] \tau_1 \leq [E] \tau_2 \rightarrow [S] \tau_1$. Since $[E] \tau_2 = [E'] \tau_2 = [S''] \tau_0 = [S'] \tau_0$, $\text{solved}([S] \bar{\Delta})$ holds.

The next lemma shows that, given a pleasant skeleton P whose term is a β -redex, any substitution generated by $\text{unify-}\beta$ applied to the constraint $\bar{\Delta}$ of P generates a substitution S that, when applied to P , (1) solves $\bar{\Delta}$ and only $\bar{\Delta}$, (2) preserves definedness of readback , and (3) changes the term of the readback by contracting the β -redex considered.

Lemma 4.11. Let $M = (\lambda x.M_1) @ M_2$, let $P = \text{initial}(M)$, let $\text{constraint}(P) = \bar{\Delta} \cap \Delta'$, and let $\bar{\Delta} \xrightarrow{\text{unify-}\beta} S$. Then $\text{readback}([S] P) = P'$ where $\text{constraint}(P') = [S] \Delta'$ and $\text{term}(P') = M_1[x := M_2]$. \square

The next two lemmas show the step-by-step equivalence between $\text{unify-}\beta$ and β -reduction.

Lemma 4.12 (One step of $\text{unify-}\beta$ corresponds to one step of β -reduction). If $\text{readback}(Q) = P = \text{initial}(M)$, $M = \text{term}(P)$, $\text{constraint}(Q) \sqsupseteq \bar{\Delta} \xrightarrow{\text{unify-}\beta} S$, and $Q' = [S] Q$, then there exist M' and P' s.t. $\text{readback}(Q') = P' = \text{initial}(M')$, and $M \xrightarrow{\beta} M' = \text{term}(P')$.

$$\begin{array}{ccccc} \bar{\Delta} & \xrightarrow{\text{unify-}\beta} & S & & \\ \uparrow \text{constraint}; \square & & \uparrow \pi_1 & & \\ Q & \xleftarrow{\pi_2} & \langle S, Q \rangle & \xrightarrow{[\cdot]} & Q' \\ \text{readback} \downarrow & & & & \downarrow \text{readback} \\ P & & & & P' \\ \text{initial} \uparrow \downarrow \text{term} & & \beta & & \text{initial} \uparrow \downarrow \text{term} \\ M & \xrightarrow{\beta} & M' & & \end{array}$$

Lemma 4.13 (One step of β -reduction corresponds to one step of $\text{unify-}\beta$). If $\text{readback}(Q) = P = \text{initial}(M)$, $\text{term}(P) = M \xrightarrow{\beta} M'$, $\text{initial}(M') = P'$, and $\text{term}(P') = M'$, then there exist $\bar{\Delta}$, S and Q' s.t. $\text{constraint}(Q) \sqsupseteq \bar{\Delta} \xrightarrow{\text{unify-}\beta} S$, $[S] Q = Q'$, and $\text{readback}(Q') = P'$.

$$\begin{array}{ccccc} \bar{\Delta} & \xrightarrow{\text{unify-}\beta} & S & & \\ \uparrow \text{constraint}; \square & & \uparrow \pi_1 & & \\ Q & \xleftarrow{\pi_2} & \langle S, Q \rangle & \xrightarrow{[\cdot]} & Q' \\ \text{readback} \downarrow & & & & \downarrow \text{readback} \\ P & & & & P' \\ \text{initial} \uparrow \downarrow \text{term} & & \beta & & \text{initial} \uparrow \downarrow \text{term} \\ M & \xrightarrow{\beta} & M' & & \end{array}$$

4.3.2 Rule unify-@

Rule `unify-@` is designed to be used after rule `unify-β` has been used as much as possible and can not be used anymore. Because `unify-β` effectively simulates β -reduction, this means when `unify-β` is done, the term resulting from `readback` is a β -normal form. Because a β -normal form may have applications in it, there may still be unsolved constraints. Rule `unify-@` applies to singular constraints of the form $\bar{\Delta} = e_1 a_0 \leq e_2 \tau_2 \rightarrow a_0$. Such constraints are generated by terms that are chains of applications whose head is a variable, i.e., terms of the form $xM_1 \cdots M_n$. For $[S] \bar{\Delta}$ to be solved, S must replace $e_1 a_0$ by an arrow type.

The substitution S produced by `unify-@` is simple, but has been carefully designed. Part of the substitution produced by `unify-@` is $S' = e_1 := (a_0 := e_2 \tau_2 \rightarrow a_0)$ which has the property that $[S'] \bar{\Delta}$ is solved. However, S' erases e_1 , which effectively merges the namespace inside e_1 with the parent namespace. Without care, this has the danger that it could cause formerly distinct variables which should remain distinct to become the same.

There are three variables that we have to be careful about that might live at the top level in the namespace of e_1 just before rule `unify-@` is applied. The strategy we use for applying rules `unify-β` and `unify-@`, discussed in the next section, guarantees that e_0 does not occur at the top-level of the namespace inside e_1 . We do not need to worry about the single T-variable a_0 , because it is completely replaced by S . The rest of the substitution produced by `unify-@` avoids confusing e_1 and e_2 (and also their nested namespaces) in the namespace of the outer e_1 with the same names in the parent namespace by replacing them by $e_1 e_1$ and respectively $e_1 e_2$. This is done at the same time as the outer e_1 is erased, thereby effectively preserving the original namespaces.

Note that if a constraint $\bar{\Delta} = e_1 (e_1 a_0 \leq e_2 \tau_2 \rightarrow a_0)$ exists in the namespace of the outer e_1 as it is erased, this constraint becomes $[S] \bar{\Delta} = e_1 e_1 a_0 \leq e_1 e_2 \tau_2 \rightarrow e_2 \tau_2 \rightarrow a_0$, to which `unify-@` would not apply, and constraint solving would get stuck at some point. This issue is easily solved by applying `unify-@` with a strategy that always selects the singular constraints that have the longer E-path first. Such a strategy is given in the next section.

4.4 Inference Algorithm

This section presents an inference algorithm producing a valid, solved skeleton for any β -normalizable term.

Let M be an arbitrarily chosen λ -term. If M has a β -normal form, then the procedure described below will terminate, otherwise it will go forever. The results of previous sections are used to design a strategy for applying rules `unify-β` and `unify-@` that exactly follows leftmost/outermost β -reduction. This reduction strategy is known to terminate for arbitrary normalizing terms, so by following it we are able to infer typings for any term that has a β -normal form.

Definition 4.14 (Leftmost/outermost redex of an untyped term). Define metavariables and sets as follows:

$$\begin{aligned} M^{\text{nf}} &\in \text{NF-Term} &::= \lambda x. M^{\text{nf}} \mid M^{\text{nf}1} \\ M^{\text{nf}1} &\in \text{NF1-Term} &::= x \mid M^{\text{nf}1} @ M^{\text{nf}} \\ C^{\text{lo}} &\in \text{LO-Context} &::= \lambda x. C^{\text{lo}} \mid C^{\text{lo}1} \\ C^{\text{lo}1} &\in \text{LO1-Context} &::= \square \mid C^{\text{lo}1} @ M \mid M^{\text{nf}1} @ C^{\text{lo}} \end{aligned}$$

For every term M , exactly one of the following conditions holds. (1) $M = M^{\text{nf}}$, meaning M is a β -normal form. (2)

$M = C^{\text{lo}}[(\lambda x. M_1) @ M_2]$, in which case the occurrence of the subterm $(\lambda x. M_1) @ M_2$ in the hole of C^{lo} is the *leftmost/outermost redex* of M . \square

In order to precisely define the complete strategy for type inference, we now define the notions of *leftmost/outermost constraint*, which we use for `unify-β`, and *rightmost/innermost constraint*, which we use for `unify-@`.

Definition 4.15 (Order on sequences of expansion variables). Let $e_i < e_j$ iff $i < j$. Thus, $<$ is a strict total order on expansion variables, and so is its lexicographic-extension order $<_{\text{lex}}$ on sequences of finite length. \square

Definition 4.16 (Leftmost/outermost constraint). If $\Delta = \bar{e}_1 \bar{\Delta}_1 \cap \cdots \cap \bar{e}_n \bar{\Delta}_n$, then the *leftmost/outermost constraint* of Δ , written $\text{LO}(\Delta)$, if it exists, is the singular constraint that has the least E-path, i.e., $\text{LO}(\Delta) = \bar{e}_i \bar{\Delta}_i$ iff $1 \leq i \leq n$ and $\bar{e}_i <_{\text{lex}} \bar{e}_j$ for any $j \in (\{1, \dots, n\} \setminus \{i\})$. \square

Definition 4.17 (Right/innermost constraint). If $\Delta = \bar{e}_1 \bar{\Delta}_1 \cap \cdots \cap \bar{e}_n \bar{\Delta}_n$, then the *rightmost/innermost constraint* of Δ , written $\text{RI}(\Delta)$, if it exists, is the singular constraint that has the greatest E-path, i.e., $\text{RI}(\Delta) = \bar{e}_i \bar{\Delta}_i$ iff $1 \leq i \leq n$ and $\bar{e}_j <_{\text{lex}} \bar{e}_i$ for any $j \in (\{1, \dots, n\} \setminus \{i\})$. \square

EXAMPLE 4.18. Consider the following skeleton:

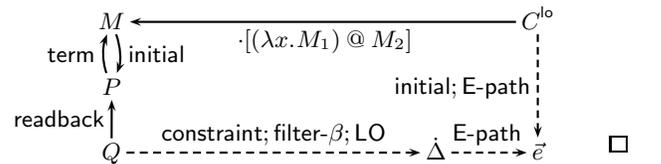
$$(e_1 \lambda x. e_2 x^{:\tau_1})^{:\tau_2} @ e_3 (e_4 y^{:\tau_5} \tau_3 @ e_5 z^{:\tau_4})$$

The leftmost/outermost constraint is $e_1 (e_2 \tau_1 \rightarrow e_2 \tau_1) \leq \tau_2$, with E-path e_1 , and the rightmost/innermost constraint is $e_3 (e_4 \tau_5 \leq \tau_3)$, with E-path e_3 . \square

By design, the leftmost/outermost constraint of a pleasant skeleton P , namely $\text{LO}(\text{constraint}(P))$, corresponds exactly to the leftmost/outermost application in $\text{term}(P)$. The leftmost/outermost constraint in $\text{constraint}(P)$ that also matches rule `unify-β` corresponds exactly to the leftmost/outermost β -redex in $\text{term}(P)$. Thus, the constraint solving strategy that always uses rule `unify-β` on the leftmost/outermost constraint matching `unify-β` corresponds exactly to leftmost/outermost β -reduction.

Let $\text{filter-}\beta(\Delta)$ contain all the singular constraints of Δ to which rule `unify-β` applies. That is, given $\Delta = \bar{\Delta}_1 \cap \cdots \cap \bar{\Delta}_n$, let $\text{filter-}\beta(\Delta) = \bar{\Delta}_{i_1} \cap \cdots \cap \bar{\Delta}_{i_p}$ where $j \in \{i_1, \dots, i_p\} \subseteq \{1, \dots, n\}$ iff there exists S s.t. $\bar{\Delta}_j \xrightarrow{\text{unify-}\beta} S$.

Lemma 4.19. *If $\text{readback}(Q) = P = \text{initial}(M)$ and $M = C^{\text{lo}}[(\lambda x. M_1) @ M_2]$, then $\hat{\Delta} = \text{LO}(\text{filter-}\beta(\text{constraint}(Q)))$ is defined and $\text{E-path}(\hat{\Delta}) = \text{E-path}(\text{initial}(C^{\text{lo}}))$.*



Similarly, in $\text{constraint}(P)$, the rightmost/innermost constraint corresponds to the rightmost/innermost application in $\text{term}(\text{readback}(Q))$. This precise correspondence breaks down once we start using rule `unify-@`, because `readback` is no longer defined after using `unify-@`. However, the intuition still explains how our strategy of using `unify-@` works, because `unify-@` does not rearrange namespaces. Appendix B presents a modified `readback` which is still defined after uses

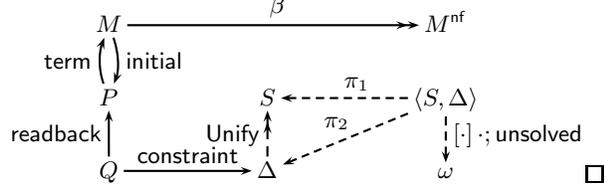
of unify-@ . We do not use it here because its correspondence with pleasant skeletons is not very tight, and this would complicate the proofs.

Definition 4.20 (Unification algorithm). Given a term M , the strategy (which succeeds iff M has a β -normal form) for solving the constraint $\Delta = \text{constraint}(\text{initial}(M))$ is:

- (0) Initial call: $\text{Unify}(\Delta) \longrightarrow \text{Unify}(\Delta, \square)$
- (1) $\text{Unify}(\Delta, S) \longrightarrow \text{Unify}([S'] \Delta, S; S')$
if $\text{LO}(\text{filter-}\beta(\Delta)) \xrightarrow{\text{unify-}\beta} S'$
- (2) $\text{Unify}(\Delta, S) \longrightarrow \text{Unify}([S'] \Delta, S; S')$
if $\text{RI}(\text{unsolved}(\Delta)) \xrightarrow{\text{unify-@}} S'$
and case (1) does not apply
- (3) Final call: $\text{Unify}(\Delta, S) \longrightarrow S$
if $\text{solved}(\Delta)$ □

Lemma 4.21 (Unify succeeds for β -normalizing terms).

If $\text{readback}(Q) = P$, $\text{term}(P) = M$, $M \xrightarrow{\beta} M^{\text{nf}}$ and $\Delta = \text{constraint}(Q)$, then $\text{Unify}(\Delta) \longrightarrow S$ and $\text{solved}([S] \Delta)$.



Definition 4.22 (Type inference algorithm). The overall algorithm is:

$$\text{Infer}(M) \longrightarrow \langle P, S \rangle$$

if $P = \text{initial}(M)$
and $\text{Unify}(\text{constraint}(P)) \longrightarrow S$ □

If $\text{Infer}(M) \longrightarrow \langle P, S \rangle$, a solved skeleton (traditional typing derivation) for M is obtained by computing $[S] P$. If only a typing (type environment, result type) is desired, it is obtained by computing $\langle [S] \text{tenv}(P) \vdash [S] \text{rtype}(P) \rangle$.

Theorem 4.23 (Infer succeeds for β -normalizing terms).

If $M \xrightarrow{\beta} M^{\text{nf}}$, then $\text{Infer}(M) \longrightarrow \langle P, S \rangle$ where $Q = [S] P$ is solved and $\text{term}(Q) = M$. □

5. REFERENCES

- [1] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.
- [2] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4), 1983.
- [3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [4] G. Boudol, P. Zimmer. On type inference in the intersection type discipline. Draft available from 1st author's web page, 2004.
- [5] S. Carlier, J. Polakow. System E experimentation tool. <http://www.macs.hw.ac.uk/ultra/compositional-analysis/system-E>.
- [6] 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.
- [7] M. Coppo, F. Damiani, P. Giannini. Strictness, totality, and non-standard type inference. *Theoret. Comput. Sci.*, 272(1-2), 2002.
- [8] M. Coppo, M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ -calculus. *Notre Dame J. Formal Logic*, 21(4), 1980.
- [9] M. Coppo, M. Dezani-Ciancaglini, B. Venneri. Principal type schemes and λ -calculus semantics. In Hindley and Seldin [17].
- [10] F. Damiani. A conjunctive type system for useless-code elimination. *Math. Structures Comput. Sci.*, 13, 2003.

- [11] F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. on Prog. Langs. & Sys.*, 25(4), 2003.
- [12] F. Damiani. Rank 2 intersection types for modules. In *Proc. 5th Int'l Conf. Principles & Practice Declarative Programming*, 2003.
- [13] F. Damiani, P. Giannini. Automatic useless-code detection and elimination for HOT functional programs. *J. Funct. Programming*, 2000.
- [14] J.-Y. Girard. Linear logic: its syntax and semantics. In J.-Y. Girard, Y. Lafont, L. Regnier, eds., *Advances in Linear Logic, Proceedings of the 1993 Workshop on Linear Logic*, London Mathematical Society Lecture Note Series. Cambridge University Press, 1995.
- [15] 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 [16].
- [16] C. Haack, J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50, 2004. Supersedes [15].
- [17] J. R. Hindley, J. P. Seldin, eds. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.
- [18] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.
- [19] 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.
- [20] 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 [22].
- [21] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [20], 2003.
- [22] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1-3), 2004. Supersedes [20]. For omitted proofs, see the longer report [21].
- [23] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. of Prog. Langs.*, 1983.
- [24] 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.
- [25] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conf. Rec. 17th Ann. ACM Symp. Princ. of Prog. Langs.*, 1990.
- [26] H. G. Mairson. From Hilbert spaces to Dilbert spaces: Context semantics made simple. In *22nd Conference on Foundations of Software Technology and Theoretical Computer Science*, 2002.
- [27] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17, 1978.
- [28] P. Møller Neergaard, H. G. Mairson. Types, potency, and impotency: Why nonlinearity and amnesia make a type system work. In *Proc. 9th Int'l Conf. Functional Programming*. ACM Press, 2004.
- [29] G. Pottinger. A type assignment for the strongly normalizable λ -terms. In Hindley and Seldin [17].
- [30] L. Regnier. *Lambda calcul et réseaux*. PhD thesis, University Paris 7, 1992.
- [31] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1-2), 1988.
- [32] S. Ronchi Della Rocca, B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1-2), 1984.
- [33] É. Sayag, M. Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, 1996.
- [34] P. Urzyczyn. Type reconstruction in \mathbf{F}_ω . *Math. Structures Comput. Sci.*, 7(4), 1997.
- [35] S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993.
- [36] S. J. van Bakel. Intersection type assignment systems. *Theoret. Comput. Sci.*, 151(2), 1995.
- [37] 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.
- [38] J. B. Wells, C. Haack. Branching types. In *Programming Languages & Systems, 11th European Symp. Programming*, vol. 2305 of *LNCS*. Springer-Verlag, 2002.
- [39] J. B. Wells, C. Haack. Branching types. *Inform. & Comput.*, 200X. Supersedes [38]. Accepted subject to revisions.

