System E: Expansion Variables for Flexible Typing with Linear and Non-linear Types and Intersection Types*

Sébastien Carlier[1], Jeff Polakow[1], J. B. Wells[1], and A. J. Kfoury[2]

[1] Heriot-Watt University, `http://www.macs.hw.ac.uk/ultra/`
[2] Boston University, `http://www.cs.bu.edu/~kfoury/`

Abstract. Types are often used to control and analyze computer programs. Intersection types give great flexibility, but have been difficult to implement. The ! operator, used to distinguish between linear and non-linear types, has good potential for better resource-usage tracking, but has not been as flexible as one might want and has been difficult to use in compositional analysis. We introduce System E, a type system with expansion variables, linear intersection types, and the ! type constructor for creating non-linear types. System E is designed for maximum flexibility in automatic type inference and for ease of automatic manipulation of type information. Expansion variables allow postponing the choice of which typing rules to use until later constraint solving gives enough information to allow making a good choice. System E removes many difficulties that expansion variables had in the earlier System I and extends expansion variables to work with ! in addition to the intersection type constructor. We present subject reduction for call-by-need evaluation and discuss program analysis in System E.

1 Discussion

1.1 Background and Motivation

Many current forms of program analysis, including many type-based analyses, work best when given the entire program to be analyzed [21, 7]. However, by their very nature, large software systems are assembled from components that are designed separately and updated at different times. Hence, for large software systems, a program analysis methodology will benefit greatly from being *compositional*, and thereby usable in a *modular* and *incremental* fashion.

Type systems for programming languages that are flexible enough to allow safe code reuse and abstract datatypes must support some kind of polymorphic types. Theoretical models for type polymorphism in existing programming languages (starting in the 1980s through now) have generally obtained type polymorphism via $\forall$ ("for all") [15, 6] and $\exists$ ("there exists") quantifiers [13] or closely related methods. Type systems with $\forall$ and $\exists$ quantifiers alone tend to

* Supported by grants: EC FP5 IST-2001-33477, EPSRC GR/L 41545/01, NATO CRG 971607, NSF 0113193 (ITR), Sun Microsystems EDUD-7826-990410-US.

be inadequate for representing modular program analysis results, because such systems fail to have *principal typings* [22, 7] where each typable term has a best typing that logically implies all of its other typings. (Do not confuse this with the much weaker property often (mis)named "principal types" associated with the Hindley/Milner type system [12, 5] used by Haskell, OCaml, Standard ML, etc.) In contrast, intersection type systems often have principal typings (see [8] for a discussion), leading to our interest in them.

Beyond basic type safety, type-based analyses can find information useful for other purposes such as optimization or security analysis. Linear type systems, with a ! operator for distinguishing between linear and non-linear types, are good for more accurate tracking of resource usage, but have not been as flexible as one might want [19, 20, 17, 16, 11]. Also, polymorphic linear type systems usually rely on quantifiers and thus are not suited for compositional analysis.

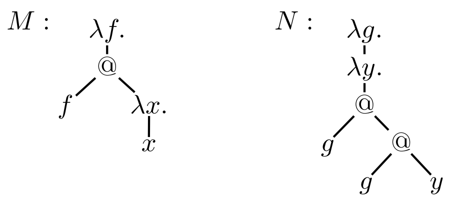
Several years ago, we developed a polymorphic type system for the λ-calculus called System I [8]. System I uses intersection types together with the new technology of *expansion variables* and has principal typings. Although the resulting program analysis can be done in a fully modular manner, there are many drawbacks to System I. The types are nearly linear (actually *affine*, i.e., used once or discarded) and multiple use of anything requires having intersection types with one branch for each use. An implication is that for any k there are simply-typable λ-terms that are not typable at rank k in System I. In contrast, rank 0 usually has the power of simple types and rank 2 usually contains the typing power of the Hindley/Milner system. System I does not have subject reduction, a basic property desired for any type system. And also, quite painfully, the substitutions used in type inference for System I do not support composition.

Some of System I's problems seemed solvable by allowing non-linear types. Also, we wanted analysis systems that can track resource usage. Toward these goals, we investigated combining the (nearly) linear intersection types of System I with the ! operator of linear type systems for controlled introduction of non-linear types. Because implementing intersection types has historically been hard, an additional goal was easier implementation. Our investigation led to System E.

1.2 Expansion and Expansion Variables

We solve the problems mentioned above with a new type system named System E that improves on previous work in the way it uses *expansion variables* (E-variables) to support *expansion.* This section gives a gentle, informal introduction to the need for expansion and how E-variables help.

Consider two typing derivations structured as follows, where λ and @ represent uses of the appropriate typing rules:



In a standard intersection type system, the derived result types could be these:

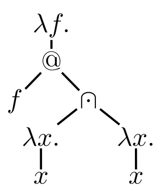
$$M : \underbrace{((\alpha \to \alpha) \to \beta)}_{\tau_1} \to \beta \qquad\qquad N : \underbrace{(\beta' \to \gamma') \cap (\alpha' \to \beta') \to \alpha' \to \gamma'}_{\tau_2}$$

In order to type the application M @ N, we must somehow "unify" τ_1 and τ_2. We could first unify the types $(\beta' \to \gamma')$ and $(\alpha' \to \beta')$ to collapse the intersection type $(\beta' \to \gamma') \cap (\alpha' \to \beta')$, but this would lose information and is not the approach we want to follow for full flexibility. Historically, in intersection type systems the solution has been to do *expansion* [4] on the result type of M:

$$M : ((\alpha_1 \to \alpha_1) \cap (\alpha_2 \to \alpha_2) \to \beta) \to \beta \qquad (1)$$

Then, the substitution $(\alpha_1 := \alpha', \beta' := \alpha', \alpha_2 := \alpha', \gamma' := \alpha', \beta := \alpha' \to \alpha')$ makes the application M @ N typable.

What justified the expansion we did to the result type of M? The expansion operation above effectively altered the typing derivation for M by inserting a use of intersection introduction at a nested position in the previous derivation, transforming it into the following new derivation, where $\cap$ marks a use of the intersection-introduction typing rule:



Expansion variables are a new technology for simplifying expansion and making it easier to implement and reason about. Expansion variables are *placeholders* for unknown uses of other typing rules, such as intersection introduction, which are propagated into the types and the type constraints used by type inference algorithms. The E-variable-introduction typing rule works like this (in traditional notation, not as stated later in the paper), where "$e\,A$" sticks the E-variable e on top of every type in the type environment A:

$$\textsf{(E-variable)}\,\frac{A \vdash M : \tau}{e\,A \vdash M : e\,\tau}$$

Type-level substitutions in System E substitute types for type variables, and *expansions* for expansion variables. An expansion is a piece of syntax standing for some number of uses of typing rules that act uniformly on every type in a judgement. The most trivial expansion, $\boxdot$, is the identity. Intersection types can also be introduced by expansion; for example, given $M : (e\,(\alpha \to \alpha) \to \beta) \to \beta$, applying the substitution $e{:=}\boxdot \cap \boxdot$ to this typing yields $M : ((\alpha \to \alpha) \cap (\alpha \to \alpha) \to \beta) \to \beta$. Substitutions are also a form of expansion; if we apply the substitution $e := (\alpha := \alpha_1) \cap (\alpha := \alpha_2)$ to M, we get the expanded typing given for M in (1) above.

Including substitutions as a form of expansion makes expansion variables effectively establish "namespaces" which can be manipulated separately. For example, applying the substitution $(e_1 := (\alpha := \alpha_1),\ e_2 := (\alpha := \alpha_2))$ to the type $(e_1\,\alpha) \cap (e_2\,\alpha)$ yields the type $\alpha_1 \cap \alpha_2$.

The syntactic expansion ω (introduced as expansion syntax in [3]) is also included, along with a typing rule that assigns the type ω (introduced in [4]) to any term. If we apply the substitution $e := \omega$ to the typing of M above, inside the typing derivation the result type for $\lambda x.x$ becomes ω. Operationally, a term which has type ω can only be passed around and must eventually be discarded.

The types in System E are by default *linear.* For example, $(\alpha_1 \to \alpha_1) \cap (\alpha_2 \to \alpha_2)$ is the type of a function to be used exactly twice, once at type $\alpha_1 \to \alpha_1$, and once at $\alpha_2 \to \alpha_2$. The ! operator creates a *non-linear* type allowing any number of uses, including 0. Subtyping rules for weakening (discarding), dereliction (using once), and contraction (duplicating) give this meaning to non-linear types. Introduction of ! is another possible case of expansion in System E.

The structure of expansions pervades every part of the design of System E. The main sorts of syntactic entities are types, typing constraints, skeletons (System E's proof terms), and expansions. Each of these sorts has cases for E-variable application, intersection, ω, and the ! operator. Other derived notions such as type environments also effectively support each of these operations. This common shared structure is the key to why System E works.

1.3 Summary of Contributions

We introduce System E and present its important properties. The proofs are in a separate long paper. System E improves on previous work as follows:

1. System E is the first type system to combine expansion variables, intersection types, the ! operator, and subtyping. Although not formally proved, we are confident that nearly every type system from the intersection type literature (without $\forall$ quantifiers) can be embedded in System E by putting enough !s in the types. System E has the polyvariant analysis power of intersection types together with the resource tracking power of linear types.
2. System E more cleanly integrates the notion of expansion from the intersection type literature (see [18] for an overview) with substitution. Unlike in System I, expansion is interleaved with substitution and, as a result, both expansions and substitutions are composable. Non-composable substitutions in System I made both proofs and implementations difficult.
3. System E cleanly supports associativity and commutativity of the intersection type constructor together with related laws that smoothly integrate intersection types with the ! type constructor.
4. System E generalizes previous notions of *expansion.* In System E's approach, expansion variables stand for any use of typing rules that operate uniformly on the result and environment types and do not change the untyped λ-term in the judgement. In System E, such rules include not only the intersection introduction rule but also rules for ! and type-level substitution. System E is the first to support such flexible expansion.

5. System E removes other difficulties of its predecessor System I. There are no restrictions on placement of expansion variables or intersection type constructors. System E has subject reduction for call-by-need evaluation. (It does not have subject reduction for call-by-name evaluation because the type system can track resource usage precisely enough to distinguish.)
6. The uniform handling of expansion variables throughout System E makes implementation simple. Demonstrating this, we present output in this paper from our type inference implementation, which can be used at this URL: `http://www.macs.hw.ac.uk/ultra/compositional-analysis/system-E/`
7. System E is parameterized on its subtyping relation. We present 3 different subtyping relations relations with different amounts of typing power.
8. System E has additional features for flexible use. Its typing judgements have separate subtyping constraints to support either eager or lazy constraint solving together with its suspended expansion rule. To help both proofs and implementations, System E's judgements have two different kind of proof terms, one for the analyzed program and one for the proof structure.

2 Preliminary Definitions

This section defines generic mathematical notions. Let h, i, j, m, n, p, and q range over $\{0, 1, 2, \ldots\}$ (the natural numbers). Given a function f, let $f[a \mapsto b] = (f \setminus \{\, (a, c) \,|\, (a, c) \in f \,\}) \cup \{(a, b)\}$. Given a relation $\xrightarrow{r}$, let $\xrightarrow{r}\!\!\!\!\rightarrow$ be its transitive, reflexive closure, w.r.t. the right carrier set. Given a context C, let $C[U]$ stand for C with the single occurrence of $\Box$ replaced by U. For example, $(\lambda x.\Box)[x] = \lambda x.x$.

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^*:

$$\vec{\varphi} \in S^* ::= \epsilon \mid \varphi \mid \vec{\varphi}_1 \cdot \vec{\varphi}_2$$
$$\epsilon \cdot \vec{\varphi} = \vec{\varphi} \qquad \vec{\varphi} \cdot \epsilon = \vec{\varphi} \qquad (\vec{\varphi}_1 \cdot \vec{\varphi}_2) \cdot \vec{\varphi}_3 = \vec{\varphi}_1 \cdot (\vec{\varphi}_2 \cdot \vec{\varphi}_3)$$

For example, $\vec{n}$ ranges over $\{0, 1, 2, \ldots\}^*$ (sequences of natural numbers). Length 1 sequences are equal to their sole member.

3 Syntax

This section defines the syntactic entities by starting from the abstract syntax grammars and metavariable conventions in fig. 1 and then modifying those definitions by the equalities and well-formedness conditions that follow.

Operator precedence is defined here, including for ordinary function application ($f(a)$) and modification ($f[a \mapsto b]$), and for operations defined later such as expansion application ($[E]\,X$) and term-variable substitution ($M_1[x{:=}M_2]$). The precedence groups are as follows, from highest to lowest:

The sorts and their abstract syntax grammars and metavariables:

$x, y, z \in$	Term-Variable	$::= \mathsf{x}_i$
$V \in$	Value	$::= x \mid \lambda x.M$
$M, N, P \in$	Term	$::= V \mid M \mathbin{@} N$
$C^{\mathrm{t}} \in$	Term-Context	$::= \Box \mid \lambda x.C^{\mathrm{t}} \mid C^{\mathrm{t}} \mathbin{@} M \mid M \mathbin{@} C^{\mathrm{t}}$
$e, f, g \in$	E-Variable	$::= \mathsf{e}_i$
$\alpha \in$	T-Variable	$::= \mathsf{a}_i$
$\nu \in$	Env-Subtyping	$::= x : \tau$
$\phi \in$	ET-Assignment	$::= \alpha := \tau \mid e := E$
$S \in$	ET-Substitution	$::= \boxdot \mid \phi,\ S$
$\tau \in$	Type	$::= \tau_1 \mathbin{\dot\cap} \tau_2 \mid e\,\tau \mid !\,\tau \mid \omega \mid \alpha \mid \tau_1 \rightarrow \tau_2$
$E \in$	Expansion	$::= E_1 \mathbin{\dot\cap} E_2 \mid e\,E \mid !\,E \mid \omega \mid S$
$\Delta \in$	Constraint	$::= \Delta_1 \mathbin{\dot\cap} \Delta_2 \mid e\,\Delta \mid !\,\Delta \mid \omega \mid \tau_1 \mathrel{\dot\leq} \tau_2$
$Q \in$	Skeleton	$::= Q_1 \mathbin{\dot\cap} Q_2 \mid e\,Q \mid !\,Q \mid \omega^M \mid \bar{Q}$
$\bar{Q} \in$	SimpleSkeleton	$::= x^{:\tau} \mid \lambda x.\,Q \mid Q_1 \mathbin{@} Q_2 \mid Q^{:\tau} \mid Q^{\nu} \mid \langle Q, E\rangle$

Metavariables ranging over multiple sorts:

$v ::= e \mid \alpha$	$T ::= \tau \mid \Delta$	$W ::= M \mid E \mid Q$	$U ::= M \mid \tau \mid E \mid \Delta \mid Q$
$\Phi ::= E \mid \tau$	$Y ::= \tau \mid E \mid \Delta$	$X ::= \tau \mid E \mid \Delta \mid Q$	

Fig. 1. Syntax grammars and metavariable conventions.

group 1: $Q^{:\tau}$, Q^{ν}, $f(a)$, $f[a \mapsto b]$, $M_1[x := M_2]$, $v := \Phi$
group 2: $e\,X$, $!\,X$, $[E]\,X$, $(\phi,\ S)$
group 3: $X_1 \mathbin{\dot\cap} X_2$, e/S
group 4: $\tau_1 \rightarrow \tau_2$, $M \mathbin{@} N$, $Q_1 \mathbin{@} Q_2$
group 5: $\tau_1 \mathrel{\dot\leq} \tau_2$, $\lambda x.M$, $\lambda x.Q$

For example, $e\,\alpha_1 \mathbin{\dot\cap} \alpha_2 \rightarrow \alpha_3 = ((e\,\alpha_1) \mathbin{\dot\cap} \alpha_2) \rightarrow \alpha_3$, and $(e\,\alpha_1 \mathrel{\dot\leq} \alpha_2) = ((e\,\alpha_1) \mathrel{\dot\leq} \alpha_2)$, and $\lambda x.\, x^{:\alpha_1} \mathbin{@} y^{:\alpha_2} = \lambda x.\,(x^{:\alpha_1} \mathbin{@} y^{:\alpha_2})$. Application is left-associative so that $M_1 \mathbin{@} M_2 \mathbin{@} M_3 = (M_1 \mathbin{@} M_2) \mathbin{@} M_3$ (similarly for skeletons) and function types are right-associative so that $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

3.1 Equalities

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

For types and constraints, the definition provided by fig. 1 is modified by imposing several equalities for E-variable application, the $\dot\cap$ and ! operators, and the ω constant. The $\dot\cap$ operator is associative and commutative with ω as its unit. E-variable application and ! both distribute over $\dot\cap$ and ω. (The constant ω can be viewed as a 0-ary version of $\dot\cap$.) The ! operator is idempotent and applications of E-variables and ! can be reordered. Formally, these rules hold:

$$\begin{array}{lll}
T_1 \mathbin{\dot\cap} T_2 = T_2 \mathbin{\dot\cap} T_1 & e\,\omega = \omega & e\,(T_1 \mathbin{\dot\cap} T_2) = e\,T_1 \mathbin{\dot\cap} e\,T_2 \\
T_1 \mathbin{\dot\cap} (T_2 \mathbin{\dot\cap} T_3) = (T_1 \mathbin{\dot\cap} T_2) \mathbin{\dot\cap} T_3 & !\,\omega = \omega & !\,(T_1 \mathbin{\dot\cap} T_2) = !\,T_1 \mathbin{\dot\cap} !\,T_2 \\
\omega \mathbin{\dot\cap} T = T & !\,!\,T = !\,T & e\,!\,T = !\,e\,T
\end{array}$$

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, $\omega \sqcap \alpha = \alpha$. After this modification, the syntactic sorts are not initial algebras. The underlying formalism to realize this could be, e.g., that types are equivalence classes closed under the rules. This level of detail is left unspecified.

Because we have imposed equalities (α-conversion and the rules for $\sqcap$, ω, !, and E-variable application in types and constraints), we can not use structural recursion and induction for definitions and proofs. To solve this, we define a size function and prove an induction principle. Details are in the long paper.

3.2 Additional Notions

Extending E-variable application to sequences, let $\epsilon\, X = X$ and $(\vec{e} \cdot e)\, X = \vec{e}\,(e\, X)$. Similarly, for environment subtyping, let $Q^{\epsilon} = Q$ and $Q^{\nu \cdot \vec{\nu}} = (Q^{\nu})^{\vec{\nu}}$.

Let $M[x := N]$ denote the usual notion of term-variable substitution in untyped terms. Let $\mathsf{FV}(M)$ denote the free term variables of M.

Let term be the least-defined function such that:

$$\begin{array}{llll}
\mathsf{term}(x^{:\tau}) & = x & \mathsf{term}(Q^{:\tau}) & = \mathsf{term}(Q) \\
\mathsf{term}(\lambda x.\, Q) & = \lambda x.\mathsf{term}(Q) & \mathsf{term}(Q^{\nu}) & = \mathsf{term}(Q) \\
\mathsf{term}(Q_1 \,@\, Q_2) & = \mathsf{term}(Q_1) \,@\, \mathsf{term}(Q_2) & \mathsf{term}(!\, Q) & = \mathsf{term}(Q) \\
\mathsf{term}(\langle Q, E \rangle) & = \mathsf{term}(Q) & \mathsf{term}(\omega^{M}) & = M \\
\mathsf{term}(e\, Q) & = \mathsf{term}(Q) & & \\
\mathsf{term}(Q_1 \sqcap Q_2) & = \mathsf{term}(Q_1) \quad \text{if } \mathsf{term}(Q_1) = \mathsf{term}(Q_2) & &
\end{array}$$

A skeleton Q is *well formed* iff $\mathsf{term}(Q)$ is defined. For example, $Q = x^{:\tau_1} \sqcap y^{:\tau_2}$ is not well formed if $x \neq y$, because $\mathsf{term}(Q)$ is not defined.

Convention 3.1 *Henceforth, only well formed skeletons are considered.* □

4 Expansion Application and Type-Level Substitution

This section defines the fundamental operation of *expansion application* which is the basis of the key features of System E. Expansion is defined, basic properties of expansion are presented, syntactic sugar for writing substitutions (a special case of expansions) is defined, and then examples are presented.

Definition 4.1 (Expansion Application). *Figure 2 defines the application of an expansion to E-variables, types, expansions, constraints, and skeletons.* □

Lemma 4.2 (Expansion Application Properties).

1. *X and $[E]\, X$ have the same sort (τ, E, Δ, or Q).*
2. *Expansion application preserves untyped terms, i.e.,* $\mathsf{term}([E]\, Q) = \mathsf{term}(Q)$.
3. *The substitution constant $\boxdot$ acts as the identity on types, expansions, constraints and skeletons, i.e., $[\boxdot]\, X = X$.* □

$$
\begin{array}{llll}
[\boxdot]\,\alpha & = \alpha & [v := \Phi,\ S]\,v & = \Phi \\
[\boxdot]\,e & = e\,\boxdot & [v := \Phi,\ S]\,v' & = [S]\,v' \text{ if } v \neq v' \\
\\
[S]\,(X_1 \cap X_2) & = [S]\,X_1 \cap [S]\,X_2 & [E_1 \cap E_2]\,X & = [E_1]\,X \cap [E_2]\,X \\
[S]\,e\,X & = [[S]\,e]\,X & [e\,E]\,X & = e\,[E]\,X \\
[S]\,!\,X & = !\,[S]\,X & [!\,E]\,X & = !\,[E]\,X \\
[S]\,\omega & = \omega & [\omega]\,Y & = \omega \\
[S]\,\omega^{M} & = \omega^{M} & [\omega]\,Q & = \omega^{\mathsf{term}(Q)} \\
\\
[S]\,(\tau_1 \to \tau_2) & = [S]\,\tau_1 \to [S]\,\tau_2 & [S]\,(\tau_1 \lessdot \tau_2) & = [S]\,\tau_1 \lessdot [S]\,\tau_2 \\
[S]\,x^{:\tau} & = x^{:[S]\,\tau} & [S]\,\boxdot & = S \\
[S]\,\lambda x.\,Q & = \lambda x.\,[S]\,Q & [S]\,(v := \Phi,\ S') & = (v := [S]\,\Phi,\ [S]\,S') \\
[S]\,(Q_1 \mathbin{@} Q_2) & = [S]\,Q_1 \mathbin{@} [S]\,Q_2 & [S]\,Q^{:\tau} & = ([S]\,Q)^{:[S]\,\tau} \\
[S]\,\langle Q, E\rangle & = \langle Q, [S]\,E\rangle & [S]\,Q^{x:\tau} & = ([S]\,Q)^{x:[S]\,\tau}
\end{array}
$$

Fig. 2. Expansion application.

Lemma 4.3 (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. 4.3, the ";" operator is associative. Although $E_1; E_2$ is not much shorter than $[E_2]\,E_1$, it allows writing, e.g., $S_1; S_2; S_3; S_4; S_5$, which is easier to follow than $[S_5]\,[S_4]\,[S_3]\,[S_2]\,S_1$.

An assignment ϕ may stand for $S = (\phi,\ \boxdot)$ and is to be interpreted that way if at all possible. The higher precedence of $(v := \Phi)$ over $(\phi,\ S)$ applies here. For example, $e_1 := e_2 := S_2,\ e_3 := S_3$ stands for $(e_1 := (e_2 := S_2)),\ e_3 := S_3$ which stands for $(e_1 := ((e_2 := S_2),\ \boxdot)),\ (e_3 := S_3),\ \boxdot$.

Let e/S stand for $(e{:=}e\,S)$. Thus, e/S stands for $((e := e\,S), \boxdot)$ when possible. The "/" notation builds a substitution that affects variables underneath an E-variable, because $[e/S]\,e\,X = e\,[S]\,X$ and $[e/S]\,X = X$ if $X \neq e\,X'$. For example, $S = (\mathsf{e}_0/(\mathsf{a}_1 := \tau_1),\ \mathsf{a}_0 := \tau_0)$ stands for $S = (\mathsf{e}_0 := \mathsf{e}_0\,(\mathsf{a}_1 := \tau_1,\ \boxdot),\ \mathsf{a}_0 := \tau_0,\ \boxdot)$ and in this case $[S]\,(\mathsf{e}_0\,\mathsf{a}_1 \to \mathsf{a}_0) = \mathsf{e}_0\,\tau_1 \to \tau_0$. We extend this notation to E-variable sequences so that $\vec{e} \cdot e/S$ stands for $\vec{e}/e/S$ and ϵ/S stands for S.

Example 4.4. E-variables effectively establish namespaces and substituting an expansion for an E-variable can merge namespaces. Define the following:

$$\tau_1 = \mathsf{e}_1\,\mathsf{a}_0 \to \mathsf{a}_0 \qquad S_1 = (\mathsf{e}_1 := \boxdot) \qquad S_2 = (\mathsf{a}_0 := \tau_2)$$

Then these facts hold:

$$
\begin{array}{llll}
[S_2]\,\tau_1 & = \mathsf{e}_1\,\mathsf{a}_0 \to \tau_2 & [S_1]\,\tau_1 & = \mathsf{a}_0 \to \mathsf{a}_0 \\
[S_2]\,[S_1]\,\tau_1 & = \tau_2 \to \tau_2 & S_1; S_2 & = (\mathsf{e}_1 := (\mathsf{a}_0 := \tau_2),\ \mathsf{a}_0 := \tau_2)
\end{array}
$$

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. □

Example 4.5. The composition $S; S'$ may take operations in one namespace in S and duplicate them to multiple namespaces in $S; S'$. Define the following:

$$\tau_1 = \mathsf{e}_1\,\mathsf{a}_0 \to \mathsf{a}_0 \qquad S_3 = (\mathsf{e}_1 := \mathsf{e}_2\,\boxdot) \qquad S_4 = (\mathsf{e}_2/(\mathsf{a}_0 := \tau_1))$$

Then these facts hold:

$$[S_4]\,[S_3]\,\tau_1 = \mathsf{e}_2\,\tau_1 \to \mathsf{a}_0 \qquad S_3; S_4 = (\mathsf{e}_1 := \mathsf{e}_2\,(\mathsf{a}_0 := \tau_1),\ \mathsf{e}_2/(\mathsf{a}_0 := \tau_1))$$

Both S_4 and its assignment $(\mathsf{a}_0{:=}\tau_1)$ appear in $S_3; S_4$. In general, arbitrary pieces of S' may appear in $S; S'$ copied to multiple places. Thus, an implementation should either compose lazily or share common substructures. □

Example 4.6. Substitutions can act differently on distinct namespaces and then merge the namespaces afterward. This is essential for composing substitutions. The key design choice making this work is making substitutions be the leaves of expansions. Define the following:

$$\tau = \mathsf{e}_1\,\mathsf{a}_0 \to \mathsf{a}_0 \qquad S_5 = \mathsf{e}_1/(\mathsf{a}_0 := \tau') \qquad S_6 = (\mathsf{e}_1 := \boxdot)$$

These facts hold:

$$\begin{array}{ll} [S_5]\,\tau\ = \mathsf{e}_1\,\tau' \to \mathsf{a}_0 & \\ [S_6]\,\tau\ = \mathsf{a}_0 \to \mathsf{a}_0 & [S_6]\,[S_5]\,\tau = \tau' \to \mathsf{a}_0 \\ S_5; S_6 = (\mathsf{e}_1 := (\mathsf{a}_0 := \tau'),\ \mathsf{e}_1 := \boxdot) & [S_5; S_6]\,\tau\ = \tau' \to \mathsf{a}_0 \end{array}$$

A "flat" substitution notion (as in System I [8]) which does not interleave expansions and substitutions can not express the composition $S_5; S_6$.

The substitution $S_5; S_6$ has an extra assignment $(\mathsf{e}_1{:=}\boxdot)$ at the end which has no effect (other than uglifying the example), because it follows the assignment $(\mathsf{e}_1 := (\mathsf{a}_0 := \tau'))$. The substitution $S_5; S_6$ is equivalent to the substitution $S_7 = (\mathsf{e}_1 := (\mathsf{a}_0 := \tau'))$, in the sense that $[S_5; S_6]\,X = [S_7]\,X$ for any X other than a skeleton with suspended expansions. Expansion application could have been defined to clean up redundant assignments, but at the cost of complexity. □

5 Type Environments and Typing Rules

This section presents the type environments and typing rules of System E. Also, the role of skeletons is explained.

A *type environment* is a total function from term variables to types which maps only a finite number of variables to types other than ω. Let A and B range over type environments. Make the following definitions:

$$\begin{aligned} [E]\,A &= \{\,(x, [E]\,A(x))\,|\,x \in \mathsf{Term\text{-}Variable}\,\} \\ A \sqcap B &= \{\,(x, A(x) \sqcap B(x))\,|\,x \in \mathsf{Term\text{-}Variable}\,\} \\ e\,A = [e\,\boxdot]\,A &= \{\,(x, e\,A(x))\,|\,x \in \mathsf{Term\text{-}Variable}\,\} \\ !\,A = [!\,\boxdot]\,A &= \{\,(x, !\,A(x))\,|\,x \in \mathsf{Term\text{-}Variable}\,\} \\ \mathsf{env}_\omega &= \{\,(x, \omega)\,|\,x \in \mathsf{Term\text{-}Variable}\,\} \end{aligned}$$

$$\textsf{(abstraction)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta}{(\lambda x.M \rhd \lambda x.\,Q) : \langle A[x \mapsto \omega] \vdash A(x) \to \tau \rangle \ / \ \Delta}$$

$$\textsf{(application)}\quad \frac{(M_1 \rhd Q_1) : \langle A_1 \vdash \tau_2 \to \tau_1 \rangle \ / \ \Delta_1; \quad (M_2 \rhd Q_2) : \langle A_2 \vdash \tau_2 \rangle \ / \ \Delta_2}{(M_1 \mathbin{@} M_2 \rhd Q_1 \mathbin{@} Q_2) : \langle A_1 \cap A_2 \vdash \tau_1 \rangle \ / \ \Delta_1 \cap \Delta_2}$$

$$\textsf{(variable)}\quad \frac{}{(x \rhd x^{:\tau}) : \langle (x : \tau) \vdash \tau \rangle \ / \ \omega} \qquad \textsf{(omega)}\quad \frac{}{(M \rhd \omega^{M}) : \langle \mathsf{env}_\omega \vdash \omega \rangle \ / \ \omega}$$

$$\textsf{(intersection)}\quad \frac{(M \rhd Q_1) : \langle A_1 \vdash \tau_1 \rangle \ / \ \Delta_1; \quad (M \rhd Q_2) : \langle A_2 \vdash \tau_2 \rangle \ / \ \Delta_2}{(M \rhd Q_1 \cap Q_2) : \langle A_1 \cap A_2 \vdash \tau_1 \cap \tau_2 \rangle \ / \ \Delta_1 \cap \Delta_2}$$

$$\textsf{(bang)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta}{(M \rhd \,!\,Q) : \langle !\,A \vdash \,!\,\tau \rangle \ / \ !\,\Delta} \qquad \textsf{(E-variable)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta}{(M \rhd e\,Q) : \langle e\,A \vdash e\,\tau \rangle \ / \ e\,\Delta}$$

$$\textsf{(suspended expansion)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta}{(M \rhd \langle Q, E \rangle) : \langle [E]A \vdash [E]\tau \rangle \ / \ [E]\Delta}$$

$$\textsf{(result subtyping)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau_1 \rangle \ / \ \Delta}{(M \rhd Q^{:\tau_2}) : \langle A \vdash \tau_2 \rangle \ / \ \Delta \cap (\tau_1 \leq \tau_2)}$$

$$\textsf{(environment subtyping)}\quad \frac{(M \rhd Q) : \langle A \vdash \tau_1 \rangle \ / \ \Delta}{(M \rhd Q^{x:\tau_2}) : \langle A[x \mapsto \tau_2] \vdash \tau_1 \rangle \ / \ \Delta \cap (\tau_2 \leq A(x))}$$

Fig. 3. Typing rules.

Let $(x_1 : \tau_1, \cdots, x_n : \tau_t)$ abbreviate $\mathsf{env}_\omega[x_1 \mapsto \tau_1]\cdots[x_n \mapsto \tau_n]$. Observe, for every E_1, E_2, A, and x, that $[E_1 \cap E_2]\,A = [E_1]\,A \cap [E_2]\,A$, that $(e\,A)(x) = e\,A(x)$, that $(!\,A)(x) = !\,A(x)$, that $\mathsf{env}_\omega = [\omega]\,A$, that $[E]\,A[x \mapsto \tau] = ([E]\,A)[x \mapsto [E]\,\tau]$, and that $[E]\,(A \cap B) = [E]\,A \cap [E]\,B$.

The typing rules of System E are given in fig. 3. The typing rules derive *judgements* of the form $(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta$. The pair $\langle A \vdash \tau \rangle$ of a type environment A and a *result type* τ is called a *typing*. The intended meaning of $(M \rhd Q) : \langle A \vdash \tau \rangle \,/\, \Delta$ is that Q is a proof that M has the typing $\langle A \vdash \tau \rangle$, provided that the constraint Δ is *solved* w.r.t. some subtyping relation.

The precise semantic meaning of a typing $\langle A \vdash \tau \rangle$ depends on the subtyping relation that is used. The typing rules avoid specifying whether a constraint Δ is solved to allow the use of different subtyping relations, depending on the user's needs. Subtyping relations for System E are discussed in sec. 6.

A skeleton Q is a special kind of term that compactly represents a tree of typing rule uses that derives a judgement for the untyped term given by $\mathsf{term}(Q)$. Thus, a skeleton is basically a *typing derivation*.

Definition 5.1 (Valid Skeleton). *A skeleton Q is* valid *iff there exist M, A, τ, and Δ such that $(M \rhd Q) : \langle A \vdash \tau \rangle \ / \ \Delta$.* □

Lemma 5.2 (Valid Skeletons Isomorphic to Typing Derivations). *If $(M_1 \rhd Q) : \langle A_1 \vdash \tau_1 \rangle \,/\, \Delta_1$ and $(M_2 \rhd Q) : \langle A_2 \vdash \tau_2 \rangle \,/\, \Delta_2$, then* $\mathsf{term}(Q) = M_1 = M_2$, *$A_1 = A_2$, $\tau_1 = \tau_2$, and $\Delta_1 = \Delta_2$.* □

Subtyping rules for $\preceq_{\mathsf{refl}}, \preceq_{\mathsf{nlin}}, \preceq_{\mathsf{flex}}$:

$$\frac{}{\tau \preceq \tau}\;(\mathsf{refl}\text{-}\preceq) \qquad \frac{\tau_1 \preceq \tau_2 \quad \tau_2 \preceq \tau_3}{\tau_1 \preceq \tau_3}\;(\mathsf{trans}\text{-}\preceq) \qquad \frac{\tau_1 \preceq \tau_2}{e\,\tau_1 \preceq e\,\tau_2}\;(e\text{-}\preceq)$$

$$\frac{\tau_3 \preceq \tau_1 \quad \tau_2 \preceq \tau_4}{\tau_1 \to \tau_2 \preceq \tau_3 \to \tau_4}\;(\to\text{-}\preceq) \qquad \frac{\tau_1 \preceq \tau_3 \quad \tau_2 \preceq \tau_4}{\tau_1 \mathbin{\dot\cap} \tau_2 \preceq \tau_3 \mathbin{\dot\cap} \tau_4}\;(\dot\cap\text{-}\preceq) \qquad \frac{\tau_1 \preceq \tau_2}{!\,\tau_1 \preceq !\,\tau_2}\;(!\text{-}\preceq)$$

Subtyping rules for $\preceq_{\mathsf{nlin}}, \preceq_{\mathsf{flex}}$:

$$\frac{}{!\,\tau \preceq \omega}\;(\mathsf{weak}\text{-}\preceq) \qquad \frac{}{!\,\tau \preceq \tau}\;(\mathsf{derel}\text{-}\preceq) \qquad \frac{}{!\,\tau \preceq !\,\tau \mathbin{\dot\cap} !\,\tau}\;(\mathsf{contr}\text{-}\preceq)$$

Subtyping rules for $\preceq_{\mathsf{flex}}$:

$$\frac{}{e\,(\tau_1 \to \tau_2) \preceq e\,\tau_1 \to e\,\tau_2}\;(e\text{-}\to\text{-}\preceq) \qquad \frac{}{!\,(\tau_1 \to \tau_2) \preceq !\,\tau_1 \to !\,\tau_2}\;(!\text{-}\to\text{-}\preceq)$$

$$\frac{}{\omega \preceq \omega \to \omega}\;(\omega\text{-}\to\text{-}\preceq) \qquad \frac{}{(\tau_1 \to \tau_3) \mathbin{\dot\cap} (\tau_2 \to \tau_4) \preceq (\tau_1 \mathbin{\dot\cap} \tau_2) \to (\tau_3 \mathbin{\dot\cap} \tau_4)}\;(\dot\cap\text{-}\to\text{-}\preceq)$$

Fig. 4. Subtyping rules.

Convention 5.3 *Henceforth, only valid skeletons are considered.* □

Let typing, $\mathsf{constraint}$, tenv, and rtype be functions s.t. $(M \triangleright Q) : \langle A \vdash \tau \rangle \,/\, \Delta$ implies $\mathsf{typing}(Q) = \langle A \vdash \tau \rangle$, $\mathsf{constraint}(Q) = \Delta$, $\mathsf{tenv}(Q) = A$, and $\mathsf{rtype}(Q) = \tau$.

6 Subtyping and Solvedness

This section defines whether a constraint Δ is *solved* w.r.t. a subtyping relation $\preceq$, and presents three interesting subtyping relations. Sec. 7 will show that if Δ is solved w.r.t. one of these relations, then the judgement $(M \triangleright Q) : \langle A \vdash \tau \rangle \,/\, \Delta$ is preserved by call-by-need evaluation of M (Δ may change to some solved Δ').

Let $\preceq$ be a metavariable ranging over subtyping relations on types. A constraint Δ is *solved* w.r.t. $\preceq$ iff $\mathsf{solved}(\preceq, \Delta)$ holds by this definition (where the double bar is meant as an equivalence):

$$\frac{\tau_1 \preceq \tau_2}{\mathsf{solved}(\preceq, \tau_1 \mathrel{\dot\leq} \tau_2)} \qquad \frac{\mathsf{solved}(\preceq, \Delta_1) \quad \mathsf{solved}(\preceq, \Delta_2)}{\mathsf{solved}(\preceq, \Delta_1 \mathbin{\dot\cap} \Delta_2)}$$

$$\frac{\mathsf{solved}(\preceq, \Delta)}{\mathsf{solved}(\preceq, e\,\Delta)} \qquad \frac{\mathsf{solved}(\preceq, \Delta)}{\mathsf{solved}(\preceq, !\,\Delta)} \qquad \frac{}{\mathsf{solved}(\preceq, \omega)}$$

A skeleton Q is *solved* iff $\mathsf{constraint}(Q)$ is. Solved skeletons correspond to typing derivations in traditional presentations.

Fig. 4 presents subtyping relations $\preceq_{\mathsf{refl}}$ ("reflexive"), $\preceq_{\mathsf{nlin}}$ ("non-linear"), and $\preceq_{\mathsf{flex}}$ ("flexible") for use with System E.

$$\begin{tikzcd}
M_0 \arrow[r, dashed, "\beta"] \arrow[d, "\mathsf{initial}"'] & M_1 \arrow[r, dashed, "\beta"] \arrow[d, dashed, bend left, "\mathsf{initial}"] & M_2 \arrow[r, dotted, "\beta"] \arrow[d, dashed, bend left, "\mathsf{initial}"] & M_n \arrow[d, dashed, bend left, "\mathsf{initial}"] \\
Q_0 = Q_0' \arrow[loop below, dashed, "\mathsf{readback}"] \arrow[rd, "\mathsf{refl\text{-}1}"'] & Q_1' \arrow[u, dashed, bend left, "\mathsf{term}"] & Q_2' \arrow[u, dashed, bend left, "\mathsf{term}"] & Q_n' \arrow[u, dashed, bend left, "\mathsf{term}"] \\
 & Q_1 \arrow[u, dashed, "\mathsf{readback}"'] \arrow[r, "\mathsf{refl\text{-}1}"] & Q_2 \arrow[u, dashed, "\mathsf{readback}"'] \arrow[r, dotted, "\mathsf{refl\text{-}1}"] & Q_n \arrow[u, dashed, "\mathsf{readback}"']
\end{tikzcd}$$

Fig. 5. Correspondence with β-reduction of a particular type inference process.

The $\preceq_{\mathsf{refl}}$ subtyping relation only allows using subtyping in trivial ways that do not add typing power. When using $\preceq_{\mathsf{refl}}$, System E is similar to System I [8], although it types more terms because it has ω. We have implemented type inference using $\preceq_{\mathsf{refl}}$ that always succeeds for any term M that has a β-normal form and that allows the β-normal form to be reconstructed from the typing.

Fig. 5 illustrates the type inference process. The full details will appear in a later paper. First, we build an initial skeleton Q_0 from the untyped term M_0 by giving every term variable the type a_0 and inserting E-variables under every abstraction (e_0) and application (e_1 for the function, e_2 for the argument). Then, the rule $\mathsf{refl\text{-}1}$ is applied to $\mathsf{constraint}(Q_0)$ to generate a substitution S_0 which is used to calculate $Q_1 = [S_0]\, Q_0$. This is repeated as long as possible.

The dashed edges in the diagram in fig. 5 show a correspondence with untyped β-reduction. The function invocation $\mathsf{readback}(Q)$ looks only at $\mathsf{tenv}(Q)$, $\mathsf{rtype}(Q)$, and $\mathsf{constraint}(Q)$ to produce an initial skeleton. In addition to β-reduction following constraint solving, the converse also holds and constraint solving can follow any β-reduction sequence, e.g., the normalizing leftmost-outermost strategy. Thus, our algorithm types all β-normalizing terms. Once the β-normal form is reached, if it contains applications, an additional constraint solving rule $\mathsf{refl\text{-}2}$ solves the remaining easy constraints.

Example 6.1. Consider $M_0 = (\lambda z.z \mathbin{@} (\lambda x.\lambda y.x) \mathbin{@} ((\lambda x.x \mathbin{@} x) \mathbin{@} z))@(\lambda y.y \mathbin{@} y)$. The normal form of M_0 is $M_5 = \lambda x.\lambda y.x$. Note that M_0 is not strongly normalizing. The initial and final skeletons for M_0 are

$$\begin{array}{rl}
Q_0 = & \left| \begin{array}{l} \mathsf{e}_1\,(\lambda z.\,\mathsf{e}_0\,(\left| \begin{array}{l} \mathsf{e}_1\,(\mathsf{e}_1\,z^{:\mathsf{a}_0} \mathbin{@} \mathsf{e}_2\,(\lambda x.\,\mathsf{e}_0\,\lambda y.\,\mathsf{e}_0\,x^{:\mathsf{a}_0})) \end{array}\right. \\
\qquad\qquad\quad\;\; @ \left| \mathsf{e}_2\,(\mathsf{e}_1\,(\lambda x.\,\mathsf{e}_0\,(\mathsf{e}_1\,x^{:\mathsf{a}_0} \mathbin{@} \mathsf{e}_2\,x^{:\mathsf{a}_0})) \mathbin{@} \mathsf{e}_2\,z^{:\mathsf{a}_0}))) \right. \end{array}\right. \\
@ & \left| \mathsf{e}_2\,(\lambda y.\,\mathsf{e}_0\,(\mathsf{e}_1\,y^{:\mathsf{a}_0} \mathbin{@} \mathsf{e}_2\,y^{:\mathsf{a}_0})) \right.
\end{array}$$

$$\begin{array}{rl}
Q_5 = & \left| (\lambda z.\,(z^{:\tau_2}) \mathbin{@} (\lambda x.\,\lambda y.\,x^{:\tau_1}) \mathbin{\dot{\cap}} (\lambda x.\,\mathsf{e}_0\,(\lambda y.\,\mathsf{e}_0\,(x^{:\mathsf{a}_0}))) \mathbin{@} \omega^{(\lambda x.x@x)@z}) \right. \\
@ & \left| (\lambda y.\,(y^{:\tau_1 \to \omega \to \tau_1}) \mathbin{@} (y^{:\tau_1})) \right.
\end{array}$$

where $\tau_1 = \mathsf{e}_0\,\mathsf{e}_0\,\mathsf{a}_0 \to \mathsf{e}_0\,(\omega \to \mathsf{e}_0\,\mathsf{a}_0)$ and $\tau_2 = (\tau_1 \to \omega \to \tau_1) \mathbin{\dot{\cap}} \tau_1 \to \omega \to \tau_1$. The final judgement is that $(M_0 \triangleright Q_5) : \langle \mathsf{env}_\omega \vdash \tau_1 \rangle \,/\, \Delta$ for some solved Δ. $\square$

The $\preceq_{\mathsf{nlin}}$ subtyping relation adds the power of idempotent intersections:

$$!\,\tau \preceq_{\mathsf{nlin}} (!\,\tau \mathbin{\dot{\cap}} !\,\tau) \preceq_{\mathsf{nlin}} !\,\tau$$

This allows many interesting terms to be typed at lower ranks. In particular, the system of simple types, the Hindley/Milner system, and the rank-k restrictions of traditional intersection type systems can all be embedded into System E when using $\preceq_{\mathsf{nlin}}$ by simply putting ! nearly everywhere in types.

Example 6.2. Using the $\preceq_{\mathsf{nlin}}$ subtyping relation, System E can encode arbitrarily imprecise usage information, unlike with $\preceq_{\mathsf{refl}}$ where it must be exact. For example, consider $\mathsf{twice} = \lambda f.\lambda x.f \mathbin{@} (f \mathbin{@} x)$, and some of its typings:

$$\begin{array}{ll}
(1) & \langle \mathsf{env}_\omega, !\,(\mathsf{a}_1 \to \mathsf{a}_1) \to \mathsf{a}_1 \to \mathsf{a}_1 \rangle \\
(2) & \langle \mathsf{env}_\omega, (\mathsf{a}_2 \to \mathsf{a}_3) \Cap (\mathsf{a}_1 \to \mathsf{a}_2) \to \mathsf{a}_1 \to \mathsf{a}_3 \rangle \\
(3) & \langle \mathsf{env}_\omega, (\mathsf{a}_1 \to \mathsf{a}_1) \Cap (\mathsf{a}_1 \to \mathsf{a}_1) \to \mathsf{a}_1 \to \mathsf{a}_1 \rangle \\
(4) & \langle \mathsf{env}_\omega, (\mathsf{a}_1 \to \mathsf{a}_2) \Cap !\,(\mathsf{a}_1 \to \mathsf{a}_1) \to \mathsf{a}_1 \to \mathsf{a}_2 \rangle
\end{array}$$

Typing (1) is like a typing with simple types; as in Linear Logic, the use of ! erases counting information, i.e., twice may use its first argument *any number of times.* Typing (2) looks like a typing in a traditional intersection type system. However, because System E types are linear by default, the typing gives more information, e.g., this typing states that the first argument is used *exactly twice.* Typing (3) is in a sense between typings (1) and (2): the first argument is used *exactly twice, at the same type.* In System E, even when intersection types are not used for additional flexibility, they can still encode precise usage information. (In an implementation, the linear part of types may of course be represented as a multiset.) Finally, typing (4) contains what we call a "must-use" type. The presence of ! on part of the argument's type erases some counting information. However, there is still one linear use: the first argument is used *at least once.* □

The $\preceq_{\mathsf{flex}}$ subtyping relation allows embedding all type derivations of the very flexible BCD type system [2], again by putting ! operators in nearly every position in types. The BCD system's subtyping rules are not satisfied by $\preceq_{\mathsf{flex}}$, but every BCD rule can be transformed into one satisfied by $\preceq_{\mathsf{flex}}$ by putting ! at all positions mentioned by the rule. The $\preceq_{\mathsf{flex}}$ relation also allows skeletons to hold the information present in Lévy-labeled λ-terms, such that constraint solving simulates labeled reduction. Our experimentation tool implements this.

Example 6.3. Consider the following variant Lévy-labeled reduction where each subterm is marked with an integer sequence. For an initial labeling, we use a distinct length-1 sequence for each subterm. The labeled reduction rule is this:

$$(\lambda x.M)^{\vec{m}} \mathbin{@} N \xrightarrow{\beta\ell} M[x := N]^{\vec{m}}$$

Lévy-labels (which we model with E-variables) track reduction history, and allow information flow for the original term to be extracted from the normal form (in System E, from the typing). Consider this labeled term and its normal form:

$$\begin{array}{l}
M = ((\lambda x.(x^1 \mathbin{@} x^2)^3)^4 \mathbin{@} (\lambda z.(z^5 \mathbin{@} y^6)^7)^8)^9 \\
M \xrightarrow{\beta\ell}\!\!\!\!\rightarrow (y^{6\cdot 5} \mathbin{@} y^6)^{7\cdot 8\cdot 2\cdot 5\cdot 7\cdot 8\cdot 1\cdot 3\cdot 4\cdot 9}
\end{array}$$

If we ask about the original term "what flows to z^5?", we can tell, by collecting labels immediately preceding label 5, that the subterms annotated 6 and 2 both flow to z^5. We can also tell which subterms influence the result. Similar to the way $\mathsf{refl\text{-}1}$ corresponds to ordinary β-reduction, we have a rule $\mathsf{flex\text{-}1}$ that solves constraints in a way corresponding to labeled reduction. Again, the full details will appear in a later paper. By using distinct E-variables throughout the initial skeleton, applying rule $\mathsf{flex\text{-}1}$ until it no longer applies, and doing readback on the typing and constraint at that point, we get this skeleton from our implementation:

$$\mathsf{e}_9\,\mathsf{e}_4\,\mathsf{e}_3\,\mathsf{e}_1\,\mathsf{e}_8\,\mathsf{e}_7\,\mathsf{e}_5\,\mathsf{e}_2\,\mathsf{e}_8\,\mathsf{e}_7\,((\mathsf{e}_5\,\mathsf{e}_6\,(y^{:\mathsf{a}_0})^{:\mathsf{e}_6\,\mathsf{a}_0 \to \mathsf{a}_0})\;@\;\mathsf{e}_6\,(y^{:\mathsf{a}_0}))$$

This skeleton has exactly the information in the reduced labeled term. $\square$

7 Subject Reduction

This section presents subject reduction results for call-by-need reduction for the three subtyping relations presented in sec. 6.

Remark 7.1. In general, subject reduction does not hold for call-by-name reduction in System E. Consider the following example:

$$\begin{array}{l}
M = \mathsf{term}(Q) = (\lambda y.x_1 \;@\; y \;@\; y) \;@\; (x_2 \;@\; z)\\
Q_1 = x_1{}^{:\mathsf{a}_1 \to \mathsf{a}_1 \to \mathsf{a}_3} \;@\; y^{:\mathsf{a}_1} \;@\; y^{:\mathsf{a}_1}\\
Q_2 = x_2{}^{:\mathsf{a}_2 \to (\mathsf{a}_1 \cap \mathsf{a}_1)} \;@\; z^{:\mathsf{a}_2}\\
Q = (\lambda y.\,Q_1) \;@\; Q_2\\
\mathsf{typing}(Q_1) = \langle (x_1 : \mathsf{a}_1 \to \mathsf{a}_1 \to \mathsf{a}_3, y : \mathsf{a}_1 \cap \mathsf{a}_1) \vdash \mathsf{a}_3 \rangle\\
\mathsf{typing}(Q_2) = \langle (x_2 : \mathsf{a}_2 \to (\mathsf{a}_1 \cap \mathsf{a}_1), z : \mathsf{a}_2) \vdash \mathsf{a}_1 \cap \mathsf{a}_1 \rangle\\
\mathsf{typing}(Q) = \langle (x_1 : \mathsf{a}_1 \to \mathsf{a}_1 \to \mathsf{a}_3, x_2 : \mathsf{a}_2 \to (\mathsf{a}_1 \cap \mathsf{a}_1), z : \mathsf{a}_2) \vdash \mathsf{a}_3 \rangle\\
M \xrightarrow{\beta} N = \mathsf{term}(Q_1)[x := \mathsf{term}(Q_2)] = x_1 \;@\; (x_2 \;@\; z) \;@\; (x_2 \;@\; z)
\end{array}$$

The skeleton Q is valid since $\mathsf{rtype}(Q_2) = \mathsf{tenv}(Q_1)(y)$, and it is solved w.r.t. any subtyping relation since its constraint is ω. If subject reduction holds, we expect that there exists some Q' such that $\mathsf{term}(Q') = N$, and that has the same typing as Q. In particular, we expect that $\mathsf{tenv}(Q)(z) = \mathsf{tenv}(Q')(z) = \mathsf{a}_2$. However, the sub-skeletons of Q' corresponding to z must both have type a_2. This makes it impossible to construct Q' since, in general, $\tau \not\preceq \tau \cap \tau$ (i.e., $\mathsf{solved}(\preceq, \tau \leq \tau \cap \tau)$ does not hold for any $\preceq$ we use). Note that if we use $\preceq_{\mathsf{nlin}}$, or $\preceq_{\mathsf{flex}}$, and replace a_1 by $!\,\mathsf{a}_1$ and a_2 by $!\,\mathsf{a}_2$, then we could construct the needed skeleton Q'. $\square$

Call-by-need reduction on untyped terms is performed by these rules:

$$\begin{array}{rcll}
(\lambda x.M) \;@\; V & \xrightarrow{\mathsf{cbn}} & M[x := V] & \\
((\lambda x.M) \;@\; (N_1 \;@\; N_2)) \;@\; P & \xrightarrow{\mathsf{cbn}} & (\lambda x.M \;@\; P) \;@\; (N_1 \;@\; N_2) & \text{if } x \notin \mathsf{FV}(P)\\
M \;@\; ((\lambda x.P) \;@\; (N_1 \;@\; N_2)) & \xrightarrow{\mathsf{cbn}} & (\lambda x.M \;@\; P) \;@\; (N_1 \;@\; N_2) & \text{if } x \notin \mathsf{FV}(M)
\end{array}$$

Let $\xrightarrow{[\mathsf{cbn}]}$ be the smallest relation such that $M \xrightarrow{\mathsf{cbn}} N$ implies $C^{\mathsf{t}}[M] \xrightarrow{[\mathsf{cbn}]} C^{\mathsf{t}}[N]$. Call-by-need evaluation is then a specific strategy of using these rules [1]. We do not include any rule for garbage collection, because it does not affect subject reduction.

Theorem 7.2 (Subject Reduction). *Given* $\preceq \in \{\preceq_{\mathsf{refl}}, \preceq_{\mathsf{nlin}}, \preceq_{\mathsf{flex}}\}$ *and a skeleton* Q_1 *such that* $(M_1 \rhd Q_1) : \langle A \vdash \tau \rangle \;/\; \varDelta_1$, $\mathsf{solved}(\preceq, \varDelta_1)$, *and* $M_1 \xrightarrow{[\mathsf{cbn}]} M_2$, *there exists* Q_2 *s.t.* $(M_2 \rhd Q_2) : \langle A \vdash \tau \rangle \;/\; \varDelta_2$ *and* $\mathsf{solved}(\preceq, \varDelta_2)$. □

Proof. The theorem is proved by induction on Q_1. The proof uses inversion properties for variant subtyping definitions without explicit transitivity. □

References

[1] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, P. Wadler. The call-by-need lambda calculus. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, 1995.

[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] S. Carlier. Polar type inference with intersection types and ω. 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.*

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

[5] L. Damas, R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, 1982.

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

[7] 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.

[8] A. J. Kfoury, J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In POPL '99 [14]. Superseded by [10].

[9] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [8], 2003.

[10] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 200X. To appear. Supersedes [8]. For omitted proofs, see the longer report [9].

[11] N. Kobayashi. Quasi-linear types. In POPL '99 [14].

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

[13] J. C. Mitchell, G. D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Langs. & Systs.*, 10(3), 1988.

[14] *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999.

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

[16] D. N. Turner, P. Wadler. Operational interpretations of linear logic. *Theoret. Comput. Sci.*, 227(1–2), 1999.

[17] D. N. Turner, P. Wadler, C. Mossin. Once upon a type. In *7th International Conference on Functional Programming and Computer Architecture*, San Diego, California, 1995.

[18] S. J. van Bakel. Intersection type assignment systems. *Theoret. Comput. Sci.*, 151(2), 1995.

[19] P. Wadler. Linear types can change the world. In M. Broy, C. B. Jones, eds., *IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

[20] P. Wadler. Is there a use for linear logic? In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM).* ACM Press, 1991.

[21] K. Wansbrough, S. P. Jones. Once upon a polymorphic type. In POPL '99 [14].

[22] 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.