The Algebra of Expansion*

Sébastien Carlier and J. B. Wells

2008-03-25T04:51

Abstract

Expansion is an operation on typings (pairs of type environments and result types) in type systems for the λ-calculus. Expansion was originally introduced for calculating possible typings of a term in systems with intersection types. Unfortunately, definitions of expansion (even the most modern ones) have been difficult for outsiders to absorb. This paper aims to clarify expansion and make it more accessible to non-specialists by isolating the pure notion of expansion on its own, independent of type systems and types. We show how expansion can be seen as a simple algebra on terms with variables, substitutions, composition, and miscellaneous constructors such that the algebra satisfies 8 simple axioms and axiom schemas: the 3 standard axioms of a monoid, 4 standard axioms or axiom schemas of substitutions (including one that corresponds to the usual "substitution lemma"), and 1 axiom schema for expansion itself. This presentation should help make more accessible to a wider community theory and techniques involving intersection types and type inference with flexible precision.

1 Discussion

Expansion was introduced nearly 3 decades ago by Coppo and Dezani [6] as an operation on *typings* (pairs of type environments and result types) in order to obtain *principal typings* in type systems for the λ-calculus with *intersection types*. Expansion transforms a typing in a way that corresponds to the effects of introducing uses of typing rules (such as intersection-introduction) at positions inside a corresponding proof of the typing, but without needing any access to the proof.

Early definitions of expansion [6, 11, 10, 13, 14, 12] were quite difficult for outsiders to follow. Kfoury and Wells attempted to make things easier with *expansion variables* (*E-variables*) in System I [7, 9], which combined substitutions with expansions and allowed in some cases composing expansions, but System I's definition of expansion suffered from a confusing scheme of renaming type variables. Later, Carlier et al. made further improvements with System E [2], which made expansions fully composable, extended expansion to introducing other typing rules than intersection-introduction (such as the rules for introducing the ω and ! type constructors), and made expansion a uniform operation on types and typing proof terms[1].

Carlier and Wells give a survey of the history of expansion [5]. In that survey, the following simple example is given of a use of expansion. (See the survey for more details on why expansion is needed for this example.) In a typical system with intersection types, the λ-term $M = (\lambda x.x(\lambda y.yz))$ can be assigned the typing $\Theta_1 = \langle (z : a) \vdash (((a \to b) \to b) \to c) \to c \rangle$, which happens to be its principal typing.[2] The term M can also be assigned the typing $\Theta_2 = \langle (z : a_1 \cap a_2) \vdash (((a_1 \to b_1) \to b_1) \cap ((a_2 \to b_2) \to b_2) \to c) \to c \rangle$, and an expansion operation can obtain Θ_2 from Θ_1 (using the early definitions of expansion). Because the early definitions of expansion were complicated, E-variables were introduced in order to make the calculations easier to mechanize and reason about. For example, in System E (using this paper's newer notation), the typing Θ_1 from above is replaced by $\Theta_3 = \langle (z : e \circ a) \vdash (e \circ ((a \to b) \to b) \to c) \rangle$, which differs from Θ_1 by the insertion of the E-variable e at two places, and Θ_2 can be obtained from Θ_3 by substituting for e the *expansion term* (often just called an *expansion*) $E_1 = (a := a_1, b := b_1, \boxdot) \cap (a := a_2, b := b_2, \boxdot)$, which is done by applying the expansion $E_2 = (e := E_1, \boxdot)$ to the typing Θ_3.

Unfortunately, even System E is sufficiently complex that it is difficult for outsiders to absorb. This paper clarifies things by isolating the pure notion of expansion on its own, independent of type systems

*This is a draft which corresponds roughly to the contents of a talk given at the ITRS 2008 workshop.

[1] System E also uses expansion on subtyping proof terms and subtyping constraint sets, but this paper will not discuss this.

[2] We write typing judgments in the form $M \blacktriangleright \langle A \vdash T \rangle$ and call $\langle A \vdash T \rangle$ a *typing*.

and types, and thereby makes the following contributions:

i) We show how expansion can be seen as a simple algebra on terms with variables, substitutions, composition, and miscellaneous constructors such that the algebra satisfies 8 simple axioms or axiom schemas: the 3 standard axioms of a monoid, 4 standard axioms or axiom schemas of substitutions (including one that corresponds to the usual "substitution lemma"), and 1 axiom schema for expansion itself.

ii) We show how the expressions written using the operators of the algebra can be put into canonical forms via a confluent and terminating rewriting system whose equational theory corresponds to the axioms required by the algebra.

iii) In addition to human-oriented proofs, the key properties of the canonical forms, the term rewriting system (TRS), and the TRS's correspondence to the algebra are also formalized in the Coq proof assistant with details provided in appendix B.

iv) We extend the expansion algebra with a notion of *sorts* allowing the algebra to be tailored to particular uses and modify the definition so that sorts are preserved.

v) We then explain how System E can be redefined with its types and typing proof terms (*skeletons*) as instances of the entities of the expansion algebra.

As a result of these contributions, this paper should make it easier for others to learn how to use expansion with E-variables to implement and reason about type systems with intersection types and other advanced type features.

2 Mathematical Preliminaries

This section presents mathematical definitions which are not particular to our work.

Let h, i, j, m, n, p, and q range over $\mathbb{N} = \{0, 1, 2, \ldots\}$, the set of natural numbers.

We rely on an operator $(\cdot, \cdot)$ for building *ordered pairs* and corresponding *projection* operators fst and snd, such that if $Z = (X, Y)$, then $\mathsf{fst}(Z) = X$ and $\mathsf{snd}(Z) = Y$. Let a *relation* be a set of pairs and let $\mathcal{R}$ range over relations. We let the statement $(X, Y) \in \mathcal{R}$ be written with the alternate notation $X \xrightarrow{\mathcal{R}} Y$. Let $\mathcal{R}^*$ be the reflexive and transitive closure of $\mathcal{R}$ and let $\mathcal{R}^=$ be the reflexive, symmetric, and transitive closure of $\mathcal{R}$; in both cases we use the convention that the reader must infer the set $\mathcal{S}$ w.r.t. which to take the reflexive closure. Let $X \xrightarrow[\mathcal{R}]{} Y$ mean $X \xrightarrow{\mathcal{R}^*} Y$, and let $X \xleftrightarrow[\mathcal{R}]{} Y$ mean $X \xrightarrow{\mathcal{R}^=} Y$. We say that a relation $\mathcal{R}$ is *terminating* iff there is no infinite sequence $X_1 \xrightarrow{\mathcal{R}} X_2 \xrightarrow{\mathcal{R}} \cdots$. If $X \xrightarrow[\mathcal{R}]{} Y$, and there exists no Z such that $Y \xrightarrow{\mathcal{R}} Z$, then we call Y *a $\mathcal{R}$-normal form of X*. A *function* is a relation f such that for all X, Y, and Z, if $\{(X, Y), (X, Z)\} \subseteq f$ then $Y = Z$. If $(X, Y) \in f$ for some Y, then $f(X)$ denotes Y, otherwise $f(X)$ is undefined. Given a function f, let $f[X \mapsto Y] = (f \setminus \{ Z \in f \mid \mathsf{fst}(Z) = X \}) \cup \{(X, Y)\}$.

If $\mathcal{R}$ is a relation on syntactic entities from set $\mathcal{S}$ (the reader must infer which $\mathcal{S}$), then we write $[\mathcal{R}]$ for the *compatible closure of $\mathcal{R}$ w.r.t. $\mathcal{S}$*, defined as follows: if $\boxed{X}$ is a context over $\mathcal{S}$ (i.e., a term from $\mathcal{S}$ with one subterm replaced by a hole), and $X_1 \xrightarrow{\mathcal{R}} X_2$, then $\boxed{X}[X_1] \xrightarrow{[\mathcal{R}]} \boxed{X}[X_2]$ where $\boxed{X}[Y]$ denotes the term in $\mathcal{S}$ resulting from filling the hole in $\boxed{X}$ by a term Y. Given two syntactic entities X and Y, let $Y \sqsubseteq X$ mean that Y *occurs in* X and let $Y \sqsubset X$ mean that Y occurs in X and $Y \neq X$.

We use the standard pure λ-calculus defined as follows

$$\begin{aligned} x, y, z \in \mathsf{Term\text{-}Variable} &::= \mathsf{x}_i \\ M, N \in \mathsf{Term} &::= x \mid \lambda x.M \mid M \mathbin{@} N \end{aligned}$$

A term of the form $\lambda x.M$ is an *abstraction* and a term of the form $M \mathbin{@} N$ is an *application*. Using an infix constructor @ for application instead of simply adjoining the operands is slightly unusual, but helps when reasoning explicitly about the application syntax constructor independently of its subterms.

Let $\equiv$ be the smallest compatible equivalence relation on Entity satisfying all instances of these axioms:

(E1)	$\boxdot \circ X \equiv X$	(E4)	$(v := X_1, X_2) \circ v \equiv X_1$
(E2)	$X \circ \boxdot \equiv X$	(E5)	$(v := X, X_2) \circ v' \equiv X_2 \circ v'$ if $v \neq v'$
(E3)	$(X_1 \circ X_2) \circ X_3 \equiv X_1 \circ (X_2 \circ X_3)$	(E6)	$S \circ (v := X_1, X_2) \equiv (v := S \circ X_1, S \circ X_2)$
(E8)	$c(X_1, \ldots, X_n) \circ X \equiv c(X_1 \circ X, \ldots, X_n \circ X)$	(E7)	$S \circ c(X_1, \ldots, X_n) \equiv c(S \circ X_1, \ldots, S \circ X_n)$

Figure 1: Axioms of Expansion Algebra

3 Expansion as an Algebra

This section presents *the algebra of expansion* which is a more abstract and general presentation of expansion than all earlier versions. Although expansion was originally defined in the context of intersection types, the algebra given here abstracts away from all details of type systems. Expansion is presented here as a fundamental operation on syntax objects related to *substitution*, and substitution is presented as a special case of expansion.

The expansion algebra is parameterized over (1) a set Variable of (object-level) *variables* ranged over by v, and (2) a set Constructor of *constructors* ranged over by c. The expressions of the expansion algebra are called *entities* and are generated by the following pseudo-grammar:

$$X \in \mathsf{Entity} ::= X_1 \circ X_2 \mid \boxdot \mid v \mid v := X_1, X_2 \mid c(X_1, \ldots, X_n)$$

Whenever we use the algebra, we will use each constructor c with a fixed arity, but there is nothing in the definition that requires this.

The expression $X_1 \circ X_2$ is the operation of *expansion application* which applies the entity X_1 to the entity X_2. We call an entity X an *expansion term* (sometimes called just an *expansion*) if it is intended to be used as the left argument of $\circ$; this distinction is informal here but is made formal via the sort Ex in the sorted expansion algebra presented in section 5. In the expression $X_1 \circ X_2$, if X_2 is an expansion term then the operation may be called *expansion composition* (that this name is reasonable is justified by axiom E3, as discussed below). The nullary operator $\boxdot$ is the *null expansion*. Entities of the form $v := X_1, X_2$ are *substitutions* ranged over by S; these are a special case of expansion terms. When X is not an expansion term, the expression $S \circ X$ acts like ordinary substitution which replaces variables in X as determined by S. The expression $S_1 \circ S_2$ composes the two substitutions S_1 and S_2.

In addition to their use in forming *expansion variables* (*E-variables*) and expansion terms, the meaning of variables v and constructed expressions $c(X_1, \ldots, X_n)$ can depend on the particular use of expansion algebra. In the case of using expansion algebra to underlie the definition of System E (given in section 6), the variables will include *type variables* and the constructed expressions will include both *types* and typing proof terms.

Figure 1 defines the equivalence relation $\equiv$ on Entity which defines the meaning of expansion. The axioms E1, E2, and E3 characterize a *monoid* (ensuring that composition works smoothly). The axioms E4, E5, E6, and E7 axiomatize simultaneous substitutions (in particular, axiom E6 composes simultaneous substitutions by stating as an axiom what is often called the "substitution lemma"). Finally, axiom E8 handles expansion.

The *expansion algebra* is the algebra that uses as its carrier the set $\mathsf{Entity}_{\equiv}$ which results from quotienting Entity by $\equiv$ and that has the binary operator $\diamond \circ \diamond$, the nullary operator $\boxdot$, the binary operators of the form $v := \diamond, \diamond$ (one for each v), and the k-ary operators $c(\diamond, \ldots, \diamond)$ (one for each c and arity k). The subalgebra $(\mathsf{Entity}_{\equiv}, \diamond \circ \diamond, \boxdot)$ is a monoid due to axioms E1, E2, and E3.

Observe that the complexity of the definition of expansion is contained in the one-line definition of Entity and the axioms in figure 1. In comparison, the complexity in previous definitions was somewhat awkwardly distributed:

Let e-canon be the smallest relation on Entity satisfying the following rewriting rules:

$$
\begin{array}{rcll}
\boxdot \circ X & \xrightarrow{\text{e-canon}}_{(1)} & X & \\
X \circ \boxdot & \xrightarrow{\text{e-canon}}_{(2)} & X & \\
(v \circ X_1) \circ X_2 & \xrightarrow{\text{e-canon}}_{(3\cdot v)} & v \circ (X_1 \circ X_2) & \\
(v := X_1, X_2) \circ (v \circ X) & \xrightarrow{\text{e-canon}}_{(3\cdot 4)} & X_1 \circ X & \\
(v := X_1, X_2) \circ (v' \circ X) & \xrightarrow{\text{e-canon}}_{(3\cdot 5)} & X_2 \circ (v' \circ X) & \text{if } v \neq v' \\
(v := X_1, X_2) \circ v & \xrightarrow{\text{e-canon}}_{(4)} & X_1 & \\
(v := X_1, X_2) \circ v' & \xrightarrow{\text{e-canon}}_{(5)} & X_2 \circ v' & \text{if } v \neq v' \\
S \circ (v := X_1, X_2) & \xrightarrow{\text{e-canon}}_{(6)} & v := S \circ X_1, S \circ X_2 & \\
S \circ c(X_1, \ldots, X_n) & \xrightarrow{\text{e-canon}}_{(7)} & c(S \circ X_1, \ldots, S \circ X_n) & \\
c(X_1, \ldots, X_n) \circ X & \xrightarrow{\text{e-canon}}_{(8)} & c(X_1 \circ X, \ldots, X_n \circ X) &
\end{array}
$$

Figure 2: Rewriting Relation for Canonicalization

i) Early definitions of expansion (before E-variables) [6, 11, 10, 13, 14, 12] as well as System I (the first system with E-variables) [7, 9] were missing the equivalent of axioms E1, E2, and E3, because these systems did not possess expansions that represented the composition of arbitrary pairs of expansions. Some of the pre-E-variable systems had a notion of chains of expansions which represented the composition of expansions but were not themselves expansions, so the overall complexity was at least as high but less well integrated. System I could compose some expansions to form new expansions, but could not correctly compose arbitrary pairs of expansions. System E [2] had E3 only as a lemma rather than an axiom (the equivalent of lemma A.11).

ii) All pre-E-variable definitions of expansion were missing the equivalent of axioms E4, E5, E6, and E7, because these systems treated substitution separately from expansion, and so the complexity of these axioms was represented in the definitions of substitution. When present, axiom E6 has often been proven as a lemma instead. System I also treated substitution separately, and in effect duplicated axioms E4, E5, and E7; only some cases of E6 were handled.

iii) The pre-E-variable definitions of expansion and also System I had complicated notions of automatic renaming of type variables integrated into the equivalent of axiom E8, while in the definition presented here (and therefore in System E) this can be handled via the integrated substitution machinery.

iv) All definitions of expansion prior to System E only defined expansion as an operation on types and typings. System E was the first to define expansion on skeletons (proof terms which represent typing derivations) and other kinds of entities like subtyping constraint sets; in previous systems the notion of expansion on typing derivations was buried inside proofs, where details were sometimes left implicit and not even written. In contrast, the definition here encompasses all these kinds of expansion.

v) As already mentioned, the definition here abstracts away from all details of type systems, while all earlier definitions had type-system-specific details mixed in with their definitions of expansion.

4 Canonical Forms

This section defines a rewriting relation $\xrightarrow{\text{[e-canon]}}$ operating on terms in Entity that can be used to put any entity X into a *canonical form* $\overline{X}$ such that $X \equiv \overline{X}$. The contributions of this section have some important consequences. First, in a sense, $\xrightarrow{\text{[e-canon]}}$ *implements* expansion: while the relation $\equiv$ is sufficient to define expansion, $\xrightarrow{\text{[e-canon]}}$ gives an effective procedure for deciding which terms in Entity are equivalent by $\equiv$. Second, the characterization of the canonical forms which we give allows seeing that $\equiv$ is sensible in that it does not relate too many entities, i.e., expansion algebra is *consistent*.

The definition of the rewriting relation $\xrightarrow{\text{[e-canon]}}$ is given in figure 2. Let the *canonical* entities (which will be proven to be the normal forms of $\xrightarrow{\text{[e-canon]}}$) be the subset $\overline{\mathsf{Entity}} \subset \mathsf{Entity}$ defined as follows:

$$\bar{X} \in \overline{\mathsf{Entity}} ::= \boxdot \mid \bar{X}^1$$
$$\bar{X}^1 \in \overline{\mathsf{Entity}}^1 ::= v \circ \bar{X}^1 \mid v \mid v := \bar{X}_1, \bar{X}_2 \mid c(\bar{X}_1, \ldots, \bar{X}_n)$$

Inspecting the definition allows easily seeing that canonical entities have the following properties. First, no occurrence of $\boxdot$ can be removed by $\xrightarrow{\text{[e-canon]}}$. Second, all applications $X_1 \circ X_2$ are such that $X_1 = v$ for some v.

We prove a number of properties of the rewriting relation $\xrightarrow{\text{[e-canon]}}$ and the canonical entities $\overline{\mathsf{Entity}}$. First, we prove that $\xrightarrow{\text{[e-canon]}}$ is terminating and confluent, implying that any rewriting strategy can be safely used to find normal forms. Second, we prove that our syntactic definition of $\overline{\mathsf{Entity}}$ precisely characterizes the normal forms of $\xrightarrow{\text{[e-canon]}}$. Third, we prove (using the previous results) that the equational theory derived from $\xrightarrow{\text{[e-canon]}}$ coincides with $\equiv$, i.e., $\equiv$ and $\xleftrightarrow{\text{[e-canon]}}{}^{*}$ are the same relation. This shows that the declarative definition of expansion given by the axioms for $\equiv$ in figure 1 is equivalent in a sense to the more procedural definition of expansion given by $\xrightarrow{\text{[e-canon]}}$. It follows from this that reasoning modulo $\equiv$ on any entity X can be performed by working on its canonical form $\bar{X}$ (obtained by reducing X into [e-canon]-normal form), and $\bar{X}$ is a useful canonical representative of the $\equiv$-equivalence class containing X. Consequently, if occurrences of expressions of the form $X_1 \circ X_2$ where $X_1 \notin \mathsf{Variable}$ are treated as calls to a function that calculates the [e-canon]-normal form, the axioms of figure 1 will in effect become equalities.

All of the interesting results in this section except confluence are proven with a formally checked proof in Coq in addition to a human-checked proof.

Now we begin the proofs. For proving that $\xrightarrow{\text{[e-canon]}}$ is terminating, we define the metric function $\|\cdot\|$ to be the function mapping all members of Entity to $\mathbb{N}$ such that all of the following hold:

$$\|v\| = \|\boxdot\| = 1 \qquad \|v := X_1, X_2\| = \|X_1\| + \|X_2\| + 1$$
$$\|c(X_1, \ldots, X_n)\| = 1 + n + \textstyle\sum_{i=1}^{n} \|X_i\| \qquad \|X_1 \circ X_2\| = (\|X_1\| \; \|X_2\| + 1)^{\|X_1\|}$$

Lemma 4.1 ([e-canon] strictly decreases the metric $\|\cdot\|$)**.** *If $X \xrightarrow{\text{[e-canon]}} X'$ then $\|X\| > \|X'\|$.* □

Lemma 4.1 is proven in appendix A.1 and is also lemma R2_decreases_size in the Coq proofs.

Theorem 4.2 ([e-canon] is terminating)**.**
Every rewriting sequence $X_1 \xrightarrow{\text{[e-canon]}} X_2 \xrightarrow{\text{[e-canon]}} \cdots$ has a finite number of steps. □

Proof. Easy by lemma 4.1. □

Lemma 4.3 (Local confluence of [e-canon])**.** *If $X \xrightarrow{\text{[e-canon]}} X_1$ and $X \xrightarrow{\text{[e-canon]}} X_2$ then there exists X' such that $X_1 \xrightarrow{\text{[e-canon]}}{}^{*} X'$ and $X_2 \xrightarrow{\text{[e-canon]}}{}^{*} X'$.* □

Lemma 4.3 is proven in appendix A.2.

Lemma 4.4 (Confluence of [e-canon])**.** *If $X \xrightarrow{\text{[e-canon]}}{}^{*} X_1$ and $X \xrightarrow{\text{[e-canon]}}{}^{*} X_2$ then there exists X' such that $X_1 \xrightarrow{\text{[e-canon]}}{}^{*} X'$ and $X_2 \xrightarrow{\text{[e-canon]}}{}^{*} X'$.* □

Proof. The result follows by Newman's Lemma from theorem 4.2 and lemma 4.3. □

Lemma 4.5 (Canonical entities are [e-canon]-normal forms)**.** *$\neg(\bar{X} \xrightarrow{\text{[e-canon]}} X')$ for any X'.* □

Proof. By induction on $\bar{X}$. □

Lemma 4.6 (Progress of [e-canon]-reduction)**.**
For all X, either X is canonical or there exists some X' such that $X \xrightarrow{\text{[e-canon]}} X'$. □

Proof. By induction on X. □

Lemma 4.7. *The canonical entities* $\overline{\mathsf{Entity}}$ *are exactly the* [e-canon]*-normal forms.* □

Proof. Follows directly from lemmas 4.5 and 4.6. This is also Lemma C_R2 in the Coq proofs. □

Theorem 4.8. $\twoheadleftarrow^{[\text{e-canon}]}\!\!\twoheadrightarrow = \equiv$. □

Theorem 4.8 is proven in appendix A.3 and is also theorem R_equiv_R3 in the Coq proofs.

Note that the Coq proof follows a slightly different strategy which does not depend on the way the proof of lemma A.10 uses confluence. This explains why it was possible to omit proving confluence in Coq. We omitted proving confluence formally because there was no reason to doubt this portion of the proofs and it would have been too much work (and proving anything formally in Coq is very expensive in terms of human time) to find or develop a formulation of Newman's Lemma that could be combined with our results to obtain confluence. (We found a previously developed version of Newman's Lemma, but at the wrong type.)

5 Sorted Expansion Algebra

In any actual use of the expansion algebra, one is not interested in using all the syntactically possible terms in the set Entity. To allow restricting attention to some subset of Entity, this section develops *sorted expansion algebra* which parameterizes the expansion algebra with sorts and rules for determining the sorts of entities.

Let Sort be the set of *sorts* ranged over by Sort. The set Sort must contain at least the constant Ex, the sort of *expansion terms*. Additional parameters are a function v-sort that maps Variable to Sort and a function c-sort such that for any constructor c and sorts $s_1, \ldots, s_n$ (for any n), it holds that $\mathsf{c\text{-}sort}(c, s_1, \ldots, s_n) \in \mathsf{Sort}$. Furthermore, if $\mathsf{c\text{-}sort}(c, s_1, \ldots, s_n) = \mathsf{Ex}$, then $s_1 = \cdots = s_n = \mathsf{Ex}$. Given v-sort and c-sort, the entity sorting function $\triangleright$ (used in infix notation) is the smallest function satisfying the following statements:

$$\frac{}{v \triangleright \mathsf{v\text{-}sort}(v)} \qquad \frac{}{\boxdot \triangleright \mathsf{Ex}} \qquad \frac{X_1 \triangleright \mathsf{Ex} \wedge X_2 \triangleright s}{X_1 \circ X_2 \triangleright s}$$

$$\frac{X_1 \triangleright \mathsf{v\text{-}sort}(v) \wedge X_2 \triangleright \mathsf{Ex}}{v := X_1, X_2 \triangleright \mathsf{Ex}} \qquad \frac{X_1 \triangleright s_1 \wedge \cdots \wedge X_n \triangleright s_n}{c(X_1, \ldots, X_n) \triangleright \mathsf{c\text{-}sort}(c, s_1, \ldots, s_n)}$$

With the constraints imposed on the parameters Sort, v-sort, and c-sort, note that axiom E8 of figure 1 is the only axiom that can require relating entities of distinct sorts; this can only happen in the case of a nullary use of a constructor which can have sort Ex. Because our intended application to System E needs a concept of the ω type which will have sort Ty and a corresponding concept of the ω expansion which will have sort Ex, this is a problem. For example, using the constructors and sorting rules for System E defined in section 6, it would hold using (the old not-yet-fixed version of) axiom E8 that $\omega^{\mathsf{Ex}} \circ (T_1 \to T_2) \triangleright \mathsf{Ty}$, that $\omega^{\mathsf{Ex}} \circ (T_1 \to T_2) \equiv \omega^{\mathsf{Ex}}$, and that $\omega^{\mathsf{Ex}} \triangleright \mathsf{Ex}$. Because we want expansion to preserve sorts in general, we need to change axiom E8 and the corresponding rewriting rule appropriately.

To make sorted expansion sort-preserving, we take as a further parameter a function resort that maps some members of Constructor × Sort to Constructor such that if $\mathsf{c\text{-}sort}(c) = \mathsf{Ex}$, then for each s there exists some c' such that $\mathsf{resort}(c, s) = c'$ and $\mathsf{c\text{-}sort}(c') = s$. (Note this can only apply for a nullary usage of c.) We restrict the old axiom E8 to require $n \geq 1$ and handle the nullary case with axiom E8′ as follows:

$$\text{(E8}') \quad c() \circ X \equiv c'() \quad \text{if } X \triangleright s \text{ and } \mathsf{resort}(c, s) = c'$$

We correspondingly restrict rule $\xrightarrow{\text{e-canon}}_{(8)}$ to the cases where $n \geq 1$ and augment it by a rule $\xrightarrow{\text{e-canon}}_{(8')}$ with a similar change. Only the constructor in the right-hand side of axiom E8′ differs from axiom E8, and it is easy to check that these changes do not invalidate any of the properties proved in the previous section. With these changes, the following lemma holds:

Lemma 5.1 (Expansion preserves sorts). *If $X_1 \triangleright s$ and $X_1 \equiv X_2$, then $X_2 \triangleright s$.* □

6 Application to Intersection Types: System E

System E is a type system for the pure λ-calculus featuring intersection types and E-variables. In contrast with the earlier definition [2], we redefine System E here with its key entities as instances of sorted expansion algebra. We present a simplified and stripped-down version of System E that has no subtyping, is fully linear (omitting support for non-linear types via the ! type constructor and subtyping), and omits other secondary features (e.g., the explicit substitution operator that aids the subject reduction proof, the presence of accumulated subtyping constraints in typing judgements used to aid reasoning about type inference, etc.). Properties of System E other than how expansion works are discussed and proved elsewhere.

6.1 Instantiating Sorted Expansion Algebra

This section introduces the syntactic entities of System E as an instance of sorted expansion algebra (defined in section section 3 and refined in section 5), from which System E inherits some of its essential properties.

Recall that expansion algebra is parameterized by (1) a set Variable and (2) a set Constructor, and that sorted expansion algebra further requires (3) a set Sort, (4) a function v-sort assigning sorts to variables, (5) a function c-sort assigning sorts to constructed terms, and (6) a function resort supporting sort-preservation for constructors with nullary uses at sort Ex.

Define parameters (1), (2), and (3) as follows:

$$\begin{aligned} e \in \mathsf{Ex\text{-}Variable} &::= \mathsf{e}_i \\ a, b, c \in \mathsf{Ty\text{-}Variable} &::= \mathsf{a}_i \\ v \in \mathsf{Variable} &::= e \mid a \\ s \in \mathsf{Sort} &::= \mathsf{Ex} \mid \mathsf{Ty} \mid \mathsf{Sk}(M) \\ c \in \mathsf{Constructor} &::= \diamond \cap \diamond \mid \omega^s \mid \diamond \to \diamond \mid x^{:\diamond} \mid \lambda x.\, \diamond \mid \diamond \mathbin{@} \diamond \end{aligned}$$

Note that there is a sort $\mathsf{Sk}(M)$ for each pure λ-term M and that there is a constructor ω^s for each sort s. (The λ-term M in $\mathsf{Sk}(M)$ allows the skeleton ω^s to fully determine the typing judgement in which it can appear; see below.) Define parameter (4) so that $\mathsf{v\text{-}sort}(e) = \mathsf{Ex}$ and $\mathsf{v\text{-}sort}(a) = \mathsf{Ty}$. Define parameter (5) as follows:

$$\begin{aligned} \mathsf{c\text{-}sort}(\diamond \cap \diamond, s, s) &= s & \mathsf{c\text{-}sort}(x^{:\diamond}, \mathsf{Ty}) &= \mathsf{Sk}(x) \\ \mathsf{c\text{-}sort}(\omega^s) &= s & \mathsf{c\text{-}sort}(\lambda x.\, \diamond, \mathsf{Sk}(M)) &= \mathsf{Sk}(\lambda x.M) \\ \mathsf{c\text{-}sort}(\diamond \to \diamond, \mathsf{Ty}, \mathsf{Ty}) &= \mathsf{Ty} & \mathsf{c\text{-}sort}(\diamond \mathbin{@} \diamond, \mathsf{Sk}(M), \mathsf{Sk}(N)) &= \mathsf{Sk}(M \mathbin{@} N) \end{aligned}$$

Note that the constructor $\diamond \cap \diamond$ is "sort-polymorphic" and can be used inside entities of multiple sorts. (In the full System E, the constructor ! is also sort-polymorphic.) Finally, define parameter (6) as $\mathsf{resort}(\omega^{\mathsf{Ex}}, s) = \omega^s$.

Given parameters (1) to (6), recall that sorted expansion algebra provides (i) a set Entity of entities, (ii) an equivalence relation ≡ on entities, (iii) a set $\overline{\mathsf{Entity}}$ of canonical entities, (iv) a confluent, terminating, sort-preserving rewriting relation $\xrightarrow{\text{[e-canon]}}$ that can rewrite any entity into an equivalent (w.r.t. ≡) canonical entity, and (v) an entity sorting function ▷.

6.2 Conventions and Quotienting

This section introduces convenient names for various subconcepts of what has been defined above. This section also declares conventions for how to read expressions, for which entities to allow, and for which entities to treat as equal.

Let E range over the set Expansion $\subset$ Entity of entities that are of sort Ex; we name these entities *expansion terms* or *expansions*. Let T range over the subset Type $\subset$ Entity of entities that are of sort Ty; we name these entities *types*. Let Q range over the subset Skeleton $\subset$ Entity of entities that are of sort Sk(M) for some M; we name these entities *skeletons*. A skeleton of sort Sk(M) is shorthand notation for a *typing derivation* (a tree of typing judgements) proving that M can be assigned some typing $\langle A \vdash T \rangle$. (The typing rules given below are carefully designed so that Q uniquely determines M, A, and T, and indeed the complete contents of every typing judgement in the derivation.)

Define that an entity X is *well sorted* iff there exists some sort s such that $X \triangleright s$. Recall that an entity has at most one sort. Recall that canonical entities are those with the smallest number of uses of $\boxdot$ and where every use of $\circ$ is of the form $v \circ X$ for some v and X.

Convention 6.1. Henceforth, only well sorted entities are considered, and the expression $X_1 \circ X_2$ stands for the unique (canonical) $\overline{X}$ (possibly $X_1 \circ X_2$ itself) such that $X_1 \circ X_2 \xrightarrow{\text{[e-canon]}}\!\!\!\!\rightarrow \overline{X}$. □

Note that convention 6.1 allows us to safely work on entities modulo the equivalence relation $\equiv$.

To disambiguate when not enough parentheses are supplied, we define precedence for operators (including ordinary function application ($f(a)$) and modification ($f[a \mapsto b]$)) so that precedence from highest to lowest is in this order: $f(a)$, $f[a \mapsto b]$, $E \circ X$, $X_1 \cap X_2$, $T_1 \to T_2$, $X_1 \mathbin{@} X_2$, $(v \coloneqq X, E)$, $\lambda x.X$. For example, $e \circ a_1 \cap a_2 \to a_3 = ((e \circ a_1)) \cap a_2) \to a_3$, and $\lambda x.\, x^{:a_1} \mathbin{@} y^{:a_2} = \lambda x.\, (x^{:a_1} \mathbin{@} y^{:a_2})$. 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 the function type constructor is right-associative so that $T_1 \to T_2 \to T_3 = T_1 \to (T_2 \to T_3)$. Terms and skeletons are quotiented by α-conversion as usual, where $\lambda x.M$ binds x in M (and similarly for skeletons).

The set of System E types is modified by imposing equalities for the $\cap$ and ω^{Ty} constructors, and to preserve consistency also for expansion application. These equalities apply only to types, and not to entities of other sorts. We take $\cap$ on types to be (1) associative, (2) commutative, and (3) have ω^{Ty} as its unit. (The constant ω^{Ty} can be viewed as a nullary version of $\cap$.) We also take E-variable application to (4) distribute over $\cap$ and (5) be absorbed by ω^{Ty} (this simply implements the nullary version of distribution). Formally, these equalities hold:

$$\begin{array}{lll} {\scriptstyle (1)}\ T_1 \cap (T_2 \cap T_3) = (T_1 \cap T_2) \cap T_3 & {\scriptstyle (2)}\ T_1 \cap T_2 = T_2 \cap T_1 & {\scriptstyle (3)}\ \omega^{\mathsf{Ty}} \cap T = T \\ {\scriptstyle (4)}\ \ e \circ (T_1 \cap T_2) = e \circ T_1 \cap e \circ T_2 & {\scriptstyle (5)}\ \ e \circ \omega^{\mathsf{Ty}} = \omega^{\mathsf{Ty}} & \end{array}$$

Given equalities (4) and (5), the following further equalities are a consequence:

$$E \circ (T_1 \cap T_2) = E \circ T_1 \cap E \circ T_2 \qquad E \circ \omega^{\mathsf{Ty}} = \omega^{\mathsf{Ty}}$$

We have made heavy use of these equalities throughout our earlier publications on System E [2, 3]. Without these equalities, we would have had to define a lot of auxiliary machinery to replace them, and the papers would have been much longer and more of a burden for the reader.

6.3 The Type System

This section gives the typing rules of System E. Like for most type systems for calculi with free variables such as the λ-calculus, typing judgements depend on a notion of type environments, which we also define here.

Type environments, ranged over by A, are functions that map all members of Term-Variable into Type, and which furthermore map only a finite number of term variables to types other than ω^{Ty}. Let ω^{Env} denote the type environment mapping every term variable to ω^{Ty}, i.e., $\omega^{\mathsf{Env}}(x) = \omega^{\mathsf{Ty}}$ for all x. Let $(x_1 : T_1, \ldots, x_n : T_n)$ abbreviate $\omega^{\mathsf{Env}}[x_1 \mapsto T_1] \cdots [x_n \mapsto T_n]$. We define the following additional operations on type environments:

$$E \circ A = \{\, (x, E \circ A(x)) \mid x \in \text{Term-Variable} \,\}$$
$$A_1 \cap A_2 = \{\, (x, A_1(x) \cap A_2(x)) \mid x \in \text{Term-Variable} \,\}$$

$$\frac{Q \blacktriangleright M : \langle A \vdash T \rangle}{e \circ Q \blacktriangleright M : \langle e \circ A \vdash e \circ T \rangle}\ \text{e-app} \qquad \text{(E-variable application)}$$

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

$$\frac{}{\omega^{\mathsf{Sk}(M)} \blacktriangleright M : \langle \omega^{\mathsf{Env}} \vdash \omega^{\mathsf{Ty}} \rangle}\ \omega \qquad (\omega^{\mathsf{Ty}} \text{ introduction})$$

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

$$\frac{Q \blacktriangleright M : \langle A \vdash T \rangle}{\lambda x.\, Q \blacktriangleright \lambda x.M : \langle A[x \mapsto \omega^{\mathsf{Ty}}] \vdash A(x) \to T \rangle}\ \text{abs} \qquad \text{(abstraction)}$$

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

Figure 3: Typing rules of System E.

Note that the way $\circ$ is extended to handle type environments means that axioms E1, E3, and E8 of expansion algebra given in figure 1 are also satisfied whenever the rightmost X in the axiom is replaced by an A. In effect, for the purposes of expansion, a type environment $(x_1 : T_1, \ldots, x_n : T_n)$ acts roughly like a constructed term $c(T_1, \ldots, T_n)$, and if viewed this way one sees that axiom E7 also holds.

Note also that all of the following equalities hold:

$$A_1 \cap (A_2 \cap A_3) = (A_1 \cap A_2) \cap A_3 \qquad A_1 \cap A_2 = A_2 \cap A_1 \qquad \omega^{\mathsf{Env}} \cap A = A$$
$$e \circ (A_1 \cap A_2) = (e \circ A_1) \cap (e \circ A_2) \qquad e \circ \omega^{\mathsf{Env}} = \omega^{\mathsf{Env}}$$

The typing rules of System E are given in Figure 3. These rules derive *typing judgements* of the form $Q \blacktriangleright M : \langle A \vdash T \rangle$, which can be read as stating that "Q denotes a proof that M can be assigned the typing $\langle A \vdash T \rangle$". Although the sort of Q uniquely determines M, we still include M in typing judgements because it makes the typing rules easier to read, and because M is the real "subject" of the typing judgement, unlike Q which is just a piece of syntax denoting a typing derivation.

In System E, every operation $E \circ Q$ applying an expansion E to a skeleton Q corresponds to *splicing in* a number of uses of typing rules as determined by E. We can now finally present the key property that the expansion machinery was designed to achieve, namely that an expansion E can be applied to the typing derived by the skeleton Q to obtain the typing that would be derived by $E \circ Q$, without needing to inspect Q:

Lemma 6.2 (Expansion preserves derivability of typings)**.**
If $Q \blacktriangleright M : \langle A \vdash T \rangle$*, then* $E \circ Q \blacktriangleright M : \langle E \circ A \vdash E \circ T \rangle$. □

Proof. By induction on E and Q. □

We now give in figure 4 an extended example illustrating how expansion works on skeletons and types inside skeletons.

References

[1] F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. One citation in section A.2.

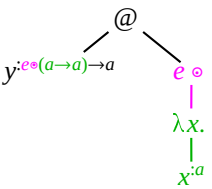
[2] S. Carlier, J. Polakow, J. B. Wells, A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004. 4 citations in section(s) 1, 3, 6, and 6.2.

[3] S. Carlier, J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. In *Proc. 6th Int'l Conf. Principles & Practice Declarative Programming*, 2004. Completely supersedes [4]. 2 citations in section(s) 6.2 and 6.3.

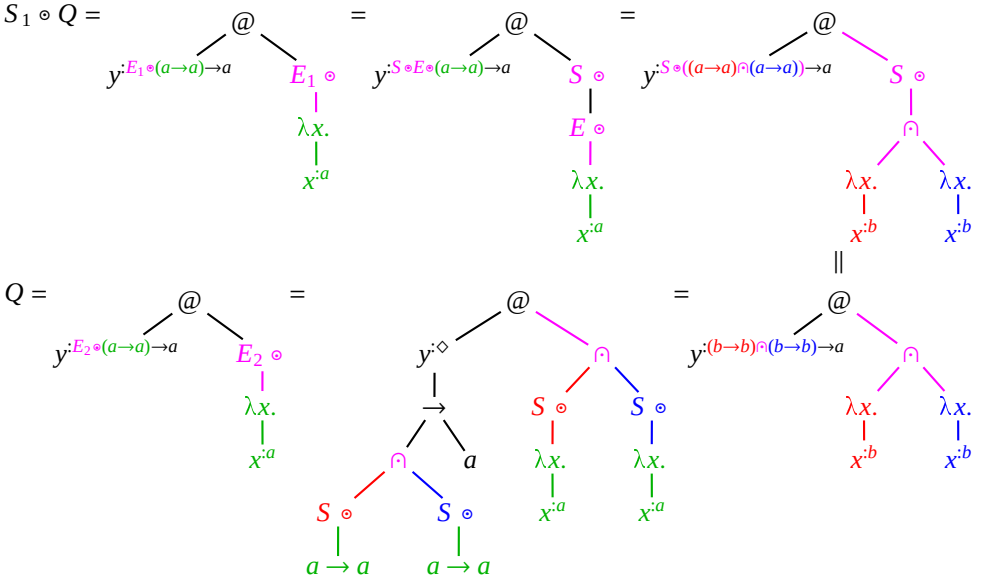
[4] S. Carlier, J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. Technical Report HW-MACS-TR-0012, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004. Completely superseded by [3]. One citation in section 6.3.

Assume the following definitions:

$$E = \boxdot \cap \boxdot \qquad S = (a := b, \boxdot) \qquad Q =$$
$$E_1 = S \circledcirc E \qquad S_1 = (e := E_1, \boxdot)$$
$$E_2 = S \cap S \qquad S_2 = (e := E_2, \boxdot)$$



These equalities hold:



This examples composes in E_1 the substitution S with the expansion E. The expansion E_2 is the result of this composition, and both E_1 and E_2 have the same effect when applied to Q. In fact, it is easy to check that $E_1 = E_2$.

Figure 4: Example of expansion composition.

[5] S. Carlier, J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. In *Proc. 3rd Int'l Workshop Intersection Types & Related Systems (ITRS 2004)*, 2005. The ITRS '04 proceedings appears as vol. 136 (2005-07-19) of *Elec. Notes in Theoret. Comp. Sci.* One citation in section 1.

[6] 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. 3 citations in section(s) 1, 1, and 3.

[7] 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 [9]. 4 citations in section(s) 1, 3, 6.3, and 6.3.

[8] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [7], 2003. One citation in section 6.3.

[9] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3), 2004. Supersedes [7]. For omitted proofs, see the longer report [8]. 3 citations in section(s) 1, 3, and 6.3.

[10] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2), 1988. 2 citations in section(s) 1 and 3.

[11] S. Ronchi Della Rocca, B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2), 1984. 2 citations in section(s) 1 and 3.

[12] S. van Bakel, F. Barbanera, M. Fernández. Polymorphic intersection type assignment for rewrite systems with abstractions and *eta*-rule. In *TYPES*, 1999. 2 citations in section(s) 1 and 3.

[13] S. J. van Bakel. Principal type schemes for the strict type assignment system. *J. Logic Comput.*, 3(6), 1993. 2 citations in section(s) 1 and 3.

[14] S. J. van Bakel. Intersection type assignment systems. *Theoret. Comput. Sci.*, 151(2), 1995. 2 citations in section(s) 1 and 3.

A Human-Checked Proofs

This appendix presents additional proof details that are not appropriate for the main body of the paper because they are either too technical, or they are not important enough and also use too much space.

A.1 Termination of [e-canon]

Lemma A.1. *For any X, X', and S, we have $||X|| \geq 1$, $||X \circ X'|| \geq 2$, and $||S|| \geq 3$.* □

Lemma A.2. *Let $a, b, n \in \mathbb{N}$ such that $a, b \geq 1$ and $n \geq 2$. Then $(a+b)^n > a^n + b^n + 1$.* □

Proof. By induction on n. □

Lemma A.3. *Let $m, n, a_1, \ldots, a_m \in \mathbb{N}$ such that $m, n \geq 2$ and $a_1, \ldots, a_m \geq 1$. We have:*

$$1 + (\sum_{i=1}^{m} a_i)^n > m + \sum_{i=1}^{m} a_i^n = \sum_{i=1}^{m} (a_i^n + 1)$$ □

Proof. By induction on m, making use of lemma A.2. □

Lemma A.4. *If $X_1 \sqsubset X_2$ then $||X_1|| < ||X_2||$.* □

Proof. By induction on X_2. □

Lemma A.5. *If $X_1 \sqsubset X_2$ and $X_3 \sqsubset X_4$ then $||X_1 \circ X_3|| < ||X_2 \circ X_4||$.* □

Lemma A.6. *If $||X_1|| < ||X_2||$ then $||X_1 \circ X|| < ||X_2 \circ X||$.* □

Proof of Lemma 4.1. By induction on the derivation of $X \xrightarrow{\text{[e-canon]}} X'$.

- Rules 1, 2, and 5 are by lemma A.4.
- Rule 4a is by lemma A.5.
- Rules 4b and 6 are by lemmas A.6 and A.4.
- Rule 3.

$$\begin{aligned}
&||(v \circ X_1) \circ X_2|| \\
&= ((||X_1|| + 1)||X_2|| + 1)^{||X_1||+1} \\
&= (||X_1||||X_2|| + ||X_2|| + 1)^{||X_1||+1} \\
&> (||X_1||||X_2|| + 1 + 1)^{||X_1||} \\
&\geq (||X_1||||X_2|| + 1)^{||X_1||} + 1 \\
&= ||v \circ (X_1 \circ X_2)||
\end{aligned}$$

- Rule 7.

$$\begin{aligned}
&||S \circ (v := X_1, X_2)|| \\
&= (||S||(||X_1|| + ||X_2|| + 1) + 1)^{||S||} \\
&= (||S||||X_1|| + ||S||||X_2|| + ||S|| + 1)^{||S||} \\
&> ((||S||||X_1|| + 1) + (||S||||X_2|| + 1))^{||S||} \\
\text{(lemma A.2)} \quad &> (||S||||X_1|| + 1)^{||S||} + (||S||||X_2|| + 1)^{||S||} + 1 \\
&= ||v := S \circ X_1, S \circ X_2||
\end{aligned}$$

- Rule 8.
  - Case $n = 0$ is by lemma A.4.
  - Case $n = 1$ is similar to the case for rule 7. This case needs to be considered separately because the next case requires $n \geq 2$ when using lemma A.3.

– If $n \geq 2$ then

$$\begin{aligned}
& \|S \circ c(X_1, \ldots, X_n)\| \\
= & (\|S\|\,(1 + n + \textstyle\sum_{i=1}^{n} \|X_i\|) + 1)^{\|S\|} \\
= & (\|S\|\,(1 + \textstyle\sum_{i=1}^{n} (\|X_i\| + 1)) + 1)^{\|S\|} \\
= & (1 + \|S\|\,(1 + \textstyle\sum_{i=1}^{n} (\|X_i\| + 1)))^{\|S\|} \\
= & (1 + \|S\| + \textstyle\sum_{i=1}^{n} (\|S\|\,\|X_i\| + \|S\|))^{\|S\|} \\
> & (1 + \textstyle\sum_{i=1}^{n} (\|S\|\,\|X_i\| + 1))^{\|S\|} \\
\text{(lemma A.2)} \quad > & 1^{\|S\|} + (\textstyle\sum_{i=1}^{n} (\|S\|\,\|X_i\| + 1))^{\|S\|} + 1 \\
= & 1 + (\textstyle\sum_{i=1}^{n} (\|S\|\,\|X_i\| + 1))^{\|S\|} + 1 \\
\text{(lemma A.3)} \quad > & 1 + \textstyle\sum_{i=1}^{n} ((\|S\|\,\|X_i\| + 1)^{\|S\|} + 1) \\
= & 1 + n + \textstyle\sum_{i=1}^{n} (\|S\|\,\|X_i\| + 1)^{\|S\|} \\
= & \|c(S \circ X_1, \ldots, S \circ X_n)\|
\end{aligned}$$

- Rule 9.

Let $p = \|c(X_1, \ldots, X_n)\| = 1 + n + \sum_{i=1}^{n} \|X_i\|$. Then $\|c(X_1, \ldots, X_n) \circ X\| = (p\|X\| + 1)^p$.

– Case $n = 0$:

$$\|c() \circ X\| = \|X\| + 1 > 1 = \|c()\|$$

– Case $n = 1$:

$$\begin{aligned}
& \|c(X_1) \circ X_2\| \\
= & (\|X_2\| + (\|X_2\| + \|X_1\|\,\|X_2\|) + 1)^{2+\|X_1\|} \\
> & 2 + (\|X_1\|\,\|X_2\| + 1)^{\|X_1\|} \\
= & \|c(X_1 \circ X_2)\|
\end{aligned}$$

– If $n \geq 2$, then $p \geq 2$ and:

$$\begin{aligned}
& \|c(X_1, \ldots, X_n) \circ X\| \\
= & (p\|X\| + 1)^p \\
= & ((1 + n + \textstyle\sum_{i=1}^{n} \|X_i\|)\|X\| + 1)^p \\
\geq & (1 + \textstyle\sum_{i=1}^{n} (\|X_i\|\|X\| + 1))^p \\
\text{(lemma A.2)} \quad > & 1^p + (\textstyle\sum_{i=1}^{n} (\|X_i\|\,\|X\| + 1))^p + 1 \\
= & 1 + (\textstyle\sum_{i=1}^{n} (\|X_i\|\,\|X\| + 1))^p + 1 \\
\geq & 1 + (\textstyle\sum_{i=1}^{n} (\|X_i\|\,\|X\|))^{\|X_i\|} + 1 \\
\text{(lemma A.3)} \quad > & 1 + \textstyle\sum_{i=1}^{n} \left((\|X_i\|\|X\| + 1)^{\|X_i\|}\right) \\
= & 1 + n + \textstyle\sum_{i=1}^{n} \left((\|X_i\|\|X\| + 1)^{\|X_i\|}\right) \\
= & \|c(X_1 \circ X, \ldots, X_n \circ X)\|
\end{aligned}$$

- Assume the property holds for $X_1 \xrightarrow{\text{[e-canon]}} X_2$ (i.e., $\|X_1\| > \|X_2\|$).

All cases of compatibility, listed below, simply use lemma A.4 and IH.

– $X_1 \circ X \xrightarrow{\text{[e-canon]}} X_2 \circ X$

– $X \circ X_1 \xrightarrow{\text{[e-canon]}} X \circ X_2$

– $v := X_1, X \xrightarrow{\text{[e-canon]}} v := X_2, X$

– $v := X, X_1 \xrightarrow{\text{[e-canon]}} v := X, X_1$

– $c(X'_1, \ldots, X'_n, X_1, X'_{n+1}, \ldots, X'_{n+m}) \xrightarrow{\text{[e-canon]}} c(X'_1, \ldots, X'_n, X_2, X'_{n+1}, \ldots, X'_{n+m})$ □

A.2 Local Confluence of [e-canon]

Proof of Lemma 4.3. Joinability of all critical pairs implies local confluence [1, 6.2.4]. There are 6 critical pairs, all of which are joinable using $\xrightarrow{\text{[e-canon]}}{}\!\!\!\!\!\rightarrow$:

(1)
$$\begin{aligned} \boxdot \circ \boxdot &\xrightarrow{\text{e-canon}}_1 \boxdot \\ \boxdot \circ \boxdot &\xrightarrow{\text{e-canon}}_2 \boxdot \end{aligned}$$

(2)
$$\begin{aligned} (v \circ \boxdot) \circ X &\xrightarrow{\text{e-canon}}_3 v \circ (\boxdot \circ X) \\ &\xrightarrow{\text{[e-canon]}}_1 v \circ X \\ (v \circ \boxdot) \circ X &\xrightarrow{\text{[e-canon]}}_2 v \circ X \end{aligned}$$

(3)
$$\begin{aligned} (v \circ X) \circ \boxdot &\xrightarrow{\text{e-canon}}_2 v \circ X \\ (v \circ X) \circ \boxdot &\xrightarrow{\text{e-canon}}_3 v \circ (X \circ \boxdot) \\ &\xrightarrow{\text{[e-canon]}}_2 v \circ X \end{aligned}$$

(4)
$$\begin{aligned} (v \coloneqq X_1, X_2) \circ (v \circ \boxdot) &\xrightarrow{\text{e-canon}}_{4a} X_1 \circ \boxdot \\ &\xrightarrow{\text{e-canon}}_2 X_1 \\ (v \coloneqq X_1, X_2) \circ (v \circ \boxdot) &\xrightarrow{\text{[e-canon]}}_2 (v \coloneqq X_1, X_2) \circ v \\ &\xrightarrow{\text{e-canon}}_5 X_1 \end{aligned}$$

(5)
$$\begin{aligned} (v \coloneqq X_1, X_2) \circ (v' \circ \boxdot) &\xrightarrow{\text{e-canon}}_{4b} X_2 \circ (v' \circ \boxdot) \\ &\xrightarrow{\text{[e-canon]}}_2 X_2 \circ v' \\ (v \coloneqq X_1, X_2) \circ (v' \circ \boxdot) &\xrightarrow{\text{[e-canon]}}_2 (v \coloneqq X_1, X_2) \circ v' \\ &\xrightarrow{\text{e-canon}}_6 X_2 \circ v' \end{aligned}$$

(6)
$$\begin{aligned} c(X_1, \ldots, X_n) \circ \boxdot &\xrightarrow{\text{e-canon}}_2 c(X_1, \ldots, X_n) \\ c(X_1, \ldots, X_n) \circ \boxdot &\xrightarrow{\text{[e-canon]}}_9 c(X_1 \circ \boxdot, \ldots, X_n \circ \boxdot) \\ &\xrightarrow{\text{[e-canon]}}_2 c(X_1, \ldots, X_n) \end{aligned}$$

□

A.3 Correspondence of ECanon with Expansion Algebra

Lemma A.7. *$X_1 \xrightarrow{\text{e-canon}} X_2$ implies $X_1 \equiv X_2$.* □

Proof. By case analysis of $X_1 \xrightarrow{\text{e-canon}} X_2$; most cases are instances of axioms of $\equiv$, while a few cases require using multiple axioms of $\equiv$ in succession. □

Lemma A.8. *$X_1 \xrightarrow{\text{[e-canon]}} X_2$ implies $X_1 \equiv X_2$.* □

Proof. By induction on $X_1 \xrightarrow{\text{[e-canon]}} X_2$, using lemma A.7 for the base case. □

Lemma A.9. *$X_1 \xrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X_2$ implies $X_1 \equiv X_2$.* □

Proof. By induction on the length of $X_1 \xrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X_2$, using lemma A.8 when length is 1. □

Lemma A.10. *$X_1 \xleftrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X_2$ implies $X_1 \equiv X_2$.* □

Proof. If $X_1 \xleftrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X_2$, then by confluence there exists X s.t. $X_1 \xrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X$ and $X_2 \xrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X$. Using lemma A.9 we have $X_1 \equiv X$ and $X_2 \equiv X$, therefore $X_1 \equiv X_2$. □

Lemma A.11.

$$\overline{X_1} \circ (\overline{X_2} \circ \overline{X_3}) \xleftrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow (\overline{X_1} \circ \overline{X_2}) \circ \overline{X_3}$$

□

Proof. By induction on $\overline{X_1}$, with a nested induction in the case where $\overline{X_1} = S_1$, with a further nested induction on $\overline{X_3}$ in the subcase where $\overline{X_2} = S_2$. □

Lemma A.12. *$X_1 \equiv X_2$ implies $X_1 \xleftrightarrow{\text{[e-canon]}}\!\!\!\!\!\rightarrow X_2$.* □

Proof. By induction on the derivation of $X_1 \equiv X_2$. In case (4), using lemma 4.7 and then lemma A.11. All other cases (including the other ground rules, compatibility, reflexivity, symmetry, and transitivity) are trivial. □

Proof of Theorem 4.8. By lemma A.10 and lemma A.12. □

B Proofs Formalized in Coq

This appendix presents proof details that have been mechanically and formally checked with the Coq proof assistant. Human-checked versions of all these proofs are also present; the Coq proofs add a gigantic increase in the level of confidence.

Only the definitions and proven statements are included. The proofs themselves are essentially unreadable without using an interactive development environment because they are written as applications of tactics that look like "`left; subst; reflexivity.`" and "`pose eq_nat_dec. decide equality.`". The proofs can be supplied if specifically requested.

As is usual (and virtually unavoidable) when formalizing proofs in any proof assistant, details of definitions and proofs needed to change in order to fit the rigid type system imposed by Coq; fortunately (and somewhat unusually) in the particular case of these proofs the overall proof structure was able to remain fairly similar. Also, the foundation of mathematics used in these proofs is of course the calculus of constructions used by Coq, while in the human-checked proofs we use set theory; as is usual in this circumstance the reader will need to use their judgement to understand how the proofs are proving essentially the same facts, despite the conflict in foundations.

B.1 Module MyArith

Lemma plus_n_Sm : ∀ a b, a + S b = S (a + b).

Lemma plus_Sn_m : ∀ a b, S a + b = S (a + b).

Lemma S_plus_le_lt_compat : ∀ n m p, 1 ≤ m → n < p → S n < m + p.

Lemma nz : ∀ m, 0 < m → { n : nat | m = S n }.

Lemma ge_1_is_S : ∀ m, 1 ≤ m → { n : nat | m = S n }.

Lemma le_minus_O : ∀ n m, n ≤ m → m - n = 0 → n = m.

Lemma plus_ge1_ge1_not_le1 : ∀ n1 n2, 1 ≤ n1 → 1 ≤ n2 → ¬ n1 + n2 ≤ 1.

Lemma plus_permute_3_in_6 : ∀ a1 b1 c1 a2 b2 c2,
a1 + b1 + c1 + (a2 + b2 + c2) = (a1 + a2) + (b1 + b2) + (c1 + c2).

Lemma plus_permute_lr_r : ∀ n m p, (m + n) + p = (m + p) + n.

Lemma le_plus_trans_l : ∀ m n p, m ≤ n → m ≤ p + n.

Lemma not_lt_m_plus_n : ∀ m n, ¬ m + n < m.

Lemma plus_lt_le_0_compat : ∀ m n p, m ≤ n → 0 < p → m < n + p.

Lemma mult_m_Sn : ∀ m n, m × S n = m + m × n.

Lemma pull_mult_through_plus_le : ∀ n n' m m' P Q R, 0 < n → 0 < m → n ≤ n' → m ≤ m' → P + Q ≤ R → n × P + m × Q ≤ n' × m' × R.

Lemma le_mult_trans : ∀ m n p, 1 ≤ p → m ≤ n → m ≤ n × p.

Lemma lt_mult_trans : ∀ m n p, 1 ≤ p → m < n → m < n × p.

Lemma mult_lt_compat_l : ∀ n m p : nat, n < m → 0 < p → p × n < p × m.

Lemma mult_lt_compat : ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 < m2 → ∀ m3 m4, 1 ≤ m3 → 1 ≤ m4 → m3 < m4 → m1 × m3 < m2 × m4.

Lemma mult_le_lt_compat : ∀ m3 m4, 1 ≤ m3 → 1 ≤ m4 → m3 < m4 → ∀ m1 m2, 1 ≤ m2 → m1 ≤ m2 → m1 × m3 < m2 × m4.

Lemma mult_lt_le_compat : ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 < m2 → ∀ m3 m4, 1 ≤ m4 → m3 ≤ m4 → m1 × m3 < m2 × m4.

Lemma mult_reg_l : ∀ m n k, m + m × k = n + n × k → m = n.

Lemma mult_le_reg_l : ∀ k, 1 ≤ k → ∀ a b, k × a ≤ k × b → a ≤ b.

Lemma m_plus_n_mult_m : ∀ m n, m + n × m = m × (n + 1).

Fixpoint exp (m n:nat) {struct n}: nat :=
match n with
| O ⇒ 1
| S n ⇒ m × exp m n
end.

Notation "x ^ y" := (exp x y).

Lemma exp_0 : ∀ n, 1 ≤ n → 0 ^ n = 0.

Lemma exp_1 : ∀ n, 1 ^ n = 1.

Lemma m_mult_m_exp_n : ∀ m n, m × m ^ n = m ^ (n + 1).

Lemma exp_mult_distr : ∀ n p q, (p × q) ^ n = p ^ n × q ^ n.

Lemma exp_plus_distr : ∀ p n q, n ^ (p + q) = n ^ p × n ^ q.

Lemma exp_ge_1 : ∀ n m, 1 ≤ m → 1 ≤ m ^ n.

Lemma exp_plus : ∀ n, 2 ≤ n → ∀ a b, 1 ≤ a → 1 ≤ b → a ^ n + b ^ n + 1 < (a + b) ^ n.

Lemma mult_ge_1 : ∀ n1 n2, 1 ≤ n1 → 1 ≤ n2 → 1 ≤ n1 × n2.

Lemma exp_le_plus : ∀ n, 1 ≤ n → ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 ^ n < (m2 + m1) ^ n.

Lemma le_exp_trans : ∀ n m1 m2, 1 ≤ n → 1 ≤ m2 → m1 ≤ m2 → m1 ≤ m2 ^ n.

Lemma exp_le_compat_r : ∀ n, 1 ≤ n → ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 ≤ m2 → m1 ^ n ≤ m2 ^ n.

Lemma exp_le_compat_l : ∀ n1 n2, n1 ≤ n2 → ∀ m, 1 ≤ m → m ^ n1 ≤ m ^ n2.

Lemma exp_le_compat : ∀ n3 n4, 1 ≤ n3 → n3 ≤ n4 → ∀ n1 n2, 1 ≤ n1 → 1 ≤ n2 → n1 ≤ n2 → n1 ^ n3 ≤ n2 ^ n4.

Lemma lt_exp_trans : ∀ n m1 m2, 1 ≤ n → 1 ≤ m2 → m1 < m2 → m1 < m2 ^ n.

Lemma exp_lt_compat : ∀ n, 1 ≤ n → ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 < m2 → m1 ^ n < m2 ^ n.

Lemma exp_lt_lt_compat : ∀ n1 n2, 1 ≤ n1 → 1 ≤ n2 → n1 < n2 → ∀ m1 m2, 1 ≤ m1 → 1 ≤ m2 → m1 < m2 → m1 ^ n1 < m2 ^ n2.

Lemma one_le_exp : ∀ n k, 1 ≤ k → 1 ≤ k ^ n.

Lemma mult_exp_le_lt_compat : ∀ k p1 p2 q1 q2, 1 ≤ k → p1 ≤ p2 → q1 < q2 → exp k p1 × q1 < exp k p2 × q2.

Lemma max_0_r : ∀ n, max n 0 = n.

Lemma max_assoc : ∀ n1 n2 n3, max (max n1 n2) n3 = max n1 (max n2 n3).

Lemma max_permute : ∀ n1 n2 n3, max n1 (max n2 n3) = max n2 (max n1 n3).

B.2 Module MyList

Lemma length_map : ∀ (A B:Set) (f:A→B) (xs:list A), length (map f xs) = length xs.

Lemma length_app : ∀ (A:Set) (xs ys:list A), length (xs ++ ys) = length xs + length ys.

Lemma map_id : ∀ (X:Set) (xs:list X), map (fun x ⇒ x) xs = xs.

Fixpoint sum (ns : list nat) {struct ns} : nat :=
match ns with
| nil ⇒ O
| cons n ns' ⇒ n + sum ns'
end.

Lemma le_sum_trans : ∀ (A:Set) (f:A→nat) (Xs:list A) (X1 X2:A), In X2 Xs → f X1 ≤ f X2 → f X1 ≤ sum (map f Xs).

Lemma length_plus_sum : ∀ (A:Set) (f:A→nat) (Xs:list A), length Xs + sum (map f Xs) = sum (map (fun X ⇒ f X + 1) Xs).

Lemma mult_sum_distr_l : ∀ (A:Set) k (f:A→nat) Xs, k × sum (map f Xs) = sum (map (fun X ⇒ k × f X) Xs).

Lemma mult_sum_distr_r : ∀ (A:Set) k (f:A→nat) Xs, sum (map f Xs) × k = sum (map (fun X ⇒ f X × k) Xs).

Lemma sum_map_mult_k_f : ∀ (A:Set) (f:A→nat) (xs:list A) (k:nat), sum (map (fun x ⇒ k × f x) xs) = k × sum (map f xs).

Fixpoint all (A:Set) (P:A→Prop) (xs:list A) {struct xs} : Prop :=
match xs with
| nil ⇒ True
| x :: xs ⇒ P x ∧ all _ P xs
end.

Lemma all_proj : ∀ (A:Set) (P:A→Prop) (X:A) (Xs:list A), In X Xs → all A P Xs → P X.

Lemma all_ext : ∀ (A:Set) (P:A→Prop) (Xs:list A) (H:∀ X, In X Xs → P X), all A P Xs.

Lemma all_map : ∀ (A B:Set) (P:B→Prop) (f:A→B) Xs, (∀ X, P (f X)) → all B P (map f Xs).

Lemma exp_sum_0 :
∀ ps,
all _ (fun m ⇒ 1 ≤ m) ps →
∀ n, 2 ≤ n →
∀ a1 a2, 1 ≤ a1 → 1 ≤ a2 →
sum (map (fun m ⇒ m ^ n + 1) (a1 :: a2 :: ps)) <
1 + sum (a1 :: a2 :: ps) ^ n.

Lemma exp_sum :
∀ ps,

2 ≤ length ps →
all _ (fun m ⇒ 1 ≤ m) ps →
∀ n, 2 ≤ n →
sum (map (fun m ⇒ m ^ n + 1) ps) < 1 + (sum ps) ^ n.

Inductive all2 (A B:Set) (R:A→B→Prop) : list A → list B → Prop :=
| all2_nil : all2 A B R nil nil
| all2_cons : ∀ X1 X2 Xs1 Xs2, R X1 X2 → all2 A B R Xs1 Xs2 → all2 A B R (X1 :: Xs1) (X2 :: Xs2).

Lemma sum_map_ext : ∀ (A:Set) (f1 f2:A→nat) Xs, (∀ X, In X Xs → f1 X = f2 X) → sum (map f1 Xs) = sum (map f2 Xs).

Lemma sum_map_le_compat : ∀ (A:Set) (f1 f2:A→nat) Xs, (∀ X, In X Xs → f1 X ≤ f2 X) → sum (map f1 Xs) ≤ sum (map f2 Xs).

Lemma sum_map_lt_compat : ∀ (A:Set) (f1 f2:A→nat) Xs, 1 ≤ length Xs → (∀ X, In X Xs → f1 X < f2 X) → sum (map f1 Xs) < sum (map f2 Xs).

B.3 Module Utils

Lemma eq_l : ∀ (A:Set) (m n:A) (B : Set) (p:B) (R : A → B → Prop), n = m → R m p → R n p.

Lemma eq_r : ∀ (B:Set) (m p:B) (A : Set) (n:A) (R : A → B → Prop), m = p → R n m → R n p.

B.4 Module Expansion

Inductive Entity : Set :=
| App : Entity → Entity → Entity
| Id : Entity
| Var : nat → Entity
| Sub : nat → Entity → Entity → Entity
| Con : nat → list Entity → Entity.

Inductive subterm : Entity → Entity → Prop :=
| subterm_refl : ∀ X, subterm X X
| subterm_App_1 : ∀ X X1 X2, subterm X X1 → subterm X (App X1 X2)
| subterm_App_2 : ∀ X X1 X2, subterm X X2 → subterm X (App X1 X2)
| subterm_Sub_1 : ∀ X v X1 X2, subterm X X1 → subterm X (Sub v X1 X2)
| subterm_Sub_2 : ∀ X v X1 X2, subterm X X2 → subterm X (Sub v X1 X2)
| subterm_Con : ∀ c X1 X2 Xs, subterm X1 X2 → In X2 Xs → subterm X1 (Con c Xs).

Inductive proper_subterm : Entity → Entity → Prop :=
| proper_subterm_App_1 : ∀ X X1 X2, subterm X X1 → proper_subterm X (App X1 X2)
| proper_subterm_App_2 : ∀ X X1 X2, subterm X X2 → proper_subterm X (App X1 X2)
| proper_subterm_Sub_1 : ∀ X v X1 X2, subterm X X1 → proper_subterm X (Sub v X1 X2)
| proper_subterm_Sub_2 : ∀ X v X1 X2, subterm X X2 → proper_subterm X (Sub v X1 X2)
| proper_subterm_Con : ∀ c X1 X2 Xs, subterm X1 X2 → In X2 Xs → proper_subterm X1 (Con c Xs).

Definition appL (X1:Entity) (Xs:list Entity) : list Entity := map (fun X2 ⇒ App X1 X2) Xs.

Definition appR (Xs:list Entity) (X2:Entity) : list Entity := map (fun X1 ⇒ App X1 X2) Xs.

Inductive R0 : Entity → Entity → Prop :=

| R0_1 : ∀ X, R0 (App Id X) X
| R0_2 : ∀ X, R0 (App X Id) X
| R0_3v : ∀ v X1 X2, R0 (App (App (Var v) X1) X2) (App (Var v) (App X1 X2))
| R0_34 : ∀ v X1 X2 X, R0 (App (Sub v X1 X2) (App (Var v) X)) (App X1 X)
| R0_35 : ∀ v1 X1 X2 v2 X, v1 ≠ v2 → R0 (App (Sub v1 X1 X2) (App (Var v2) X)) (App X2 (App (Var v2) X))
| R0_4 : ∀ v X1 X2, R0 (App (Sub v X1 X2) (Var v)) X1
| R0_5 : ∀ v1 X1 X2 v2, v1 ≠ v2 → R0 (App (Sub v1 X1 X2) (Var v2)) (App X2 (Var v2))
| R0_6 : ∀ v1 X1 X2 v2 X3 X4, R0 (App (Sub v1 X1 X2) (Sub v2 X3 X4)) (Sub v2 (App (Sub v1 X1 X2) X3) (App (Sub v1 X1 X2) X4))
| R0_7 : ∀ v X1 X2 c Xs, R0 (App (Sub v X1 X2) (Con c Xs)) (Con c (appL (Sub v X1 X2) Xs))
| R0_8 : ∀ c Xs X, R0 (App (Con c Xs) X) (Con c (appR Xs X)).

Inductive R1 : Entity → Entity → Prop :=
| R1_App_1 : ∀ X1 X2 X, R1 X1 X2 → R1 (App X1 X) (App X2 X)
| R1_App_2 : ∀ X1 X2 X, R1 X1 X2 → R1 (App X X1) (App X X2)
| R1_Sub_1 : ∀ v X1 X2 X, R1 X1 X2 → R1 (Sub v X1 X) (Sub v X2 X)
| R1_Sub_2 : ∀ v X1 X2 X, R1 X1 X2 → R1 (Sub v X X1) (Sub v X X2)
| R1_Con : ∀ c Xs1 Xs2, R1s Xs1 Xs2 → R1 (Con c Xs1) (Con c Xs2)
| R1_R0 : ∀ X1 X2, R0 X1 X2 → R1 X1 X2
with R1s : list Entity → list Entity → Prop :=
| R1s_1 : ∀ X Xs1 Xs2, R1s Xs1 Xs2 → R1s (X :: Xs1) (X :: Xs2)
| R1s_2 : ∀ Xs X1 X2, R1 X1 X2 → R1s (X1 :: Xs) (X2 :: Xs).

Scheme R1_R1s := Induction for R1 Sort Prop
with R1s_R1 := Induction for R1s Sort Prop.

Inductive R2 : Entity → Entity → Prop :=
| R2_R1 : ∀ X1 X2, R1 X1 X2 → R2 X1 X2
| R2_trans : ∀ X1 X2 X3, R1 X1 X2 → R2 X2 X3 → R2 X1 X3.

Inductive R2s : list Entity → list Entity → Prop :=
| R2s_R1s : ∀ Xs1 Xs2, R1s Xs1 Xs2 → R2s Xs1 Xs2
| R2s_trans : ∀ Xs1 Xs2 Xs3, R1s Xs1 Xs2 → R2s Xs2 Xs3 → R2s Xs1 Xs3.

Inductive R3 : Entity → Entity → Prop :=
| R3_R1 : ∀ X1 X2, R1 X1 X2 → R3 X1 X2
| R3_refl : ∀ X, R3 X X
| R3_symm : ∀ X1 X2, R3 X1 X2 → R3 X2 X1
| R3_trans : ∀ X1 X2 X3, R3 X1 X2 → R3 X2 X3 → R3 X1 X3.

Inductive R : Entity → Entity → Prop :=
| R_1 : ∀ X, R (App Id X) X
| R_2 : ∀ X, R (App X Id) X
| R_3 : ∀ X1 X2 X3, R (App (App X1 X2) X3) (App X1 (App X2 X3))
| R_4 : ∀ v X1 X2, R (App (Sub v X1 X2) (Var v)) X1
| R_5 : ∀ v1 X1 X2 v2, v1 ≠ v2 → R (App (Sub v1 X1 X2) (Var v2)) (App X2 (Var v2))
| R_6 : ∀ v1 X1 X2 v2 X3 X4, R (App (Sub v1 X1 X2) (Sub v2 X3 X4)) (Sub v2 (App (Sub v1 X1 X2) X3) (App (Sub v1 X1 X2) X4))
| R_7 : ∀ v X1 X2 c Xs, R (App (Sub v X1 X2) (Con c Xs)) (Con c (appL (Sub v X1 X2) Xs))
| R_8 : ∀ c Xs X, R (App (Con c Xs) X) (Con c (appR Xs X))
| R_App_1 : ∀ X1 X2 X, R X1 X2 → R (App X1 X) (App X2 X)

| R_App_2 : ∀ X1 X2 X, R X1 X2 → R (App X X1) (App X X2)
| R_Sub_1 : ∀ v X1 X2 X, R X1 X2 → R (Sub v X1 X) (Sub v X2 X)
| R_Sub_2 : ∀ v X1 X2 X, R X1 X2 → R (Sub v X X1) (Sub v X X2)
| R_Con : ∀ c Xs1 Xs2, all2 _ _ R Xs1 Xs2 → R (Con c Xs1) (Con c Xs2)
| R_refl : ∀ X, R X X
| R_symm : ∀ X1 X2, R X1 X2 → R X2 X1
| R_trans : ∀ X1 X2 X3, R X1 X2 → R X2 X3 → R X1 X3.

Lemma Entity_eq_dec : ∀ (X1 X2 : Entity), { X1 = X2}+{ X1 ≠ X2}.

Fixpoint size (X:Entity) : nat :=
match X with
| App X1 X2 ⇒ (size X1 × size X2 + 1) ^ (size X1)
| Id ⇒ 1
| Var _ ⇒ 1
| Sub v X1 X2 ⇒ 1 + size X1 + size X2
| Con _ Xs ⇒ 1 + length Xs + sum (map size Xs)
end.

Lemma size_ge_1 : ∀ X, 1 ≤ size X.

Lemma size_App_ge_2 : ∀ X1 X2, 2 ≤ size (App X1 X2).

Lemma size_Sub_ge_3 : ∀ v X1 X2, 3 ≤ size (Sub v X1 X2).

Lemma size_subterm : ∀ X1 X2, subterm X1 X2 → size X1 ≤ size X2.

Lemma size_proper_subterm : ∀ X1 X2, proper_subterm X1 X2 → size X1 < size X2.

Lemma size_App_1 : ∀ X X1 X2, size X1 < size X2 → size (App X1 X) < size (App X2 X).

Lemma size_App_2 : ∀ X X1 X2, size X1 < size X2 → size (App X X1) < size (App X X2).

Lemma size_proper_subterms_App : ∀ X1 X2 X3 X4,
  proper_subterm X1 X2 →
  proper_subterm X3 X4 →
  size (App X1 X3) < size (App X2 X4).

Lemma R0_decreases_size : ∀ X1 X2, R0 X1 X2 → size X2 < size X1.

Lemma R1s_length : ∀ Xs1 Xs2, R1s Xs1 Xs2 → length Xs1 = length Xs2.

Lemma R1_decreases_size : ∀ X1 X2, R1 X1 X2 → size X2 < size X1.

Lemma R2_decreases_size : ∀ X1 X2, R2 X1 X2 → size X2 < size X1.

Definition R2_ind' :
  ∀ P : Entity → Prop,
    (∀ X1, (∀ X2, R2 X1 X2 → P X2) → P X1) → ∀ X, P X.

Definition R2_rec :
  ∀ P : Entity → Set,
    (∀ X1, (∀ X2, R2 X1 X2 → P X2) → P X1) → ∀ X, P X.

Inductive C : Entity → Prop :=
| C_App : ∀ v X, X ≠ Id → C X → C (App (Var v) X)
| C_Id : C Id
| C_Var : ∀ v, C (Var v)
| C_Sub : ∀ v X1 X2, C X1 → C X2 → C (Sub v X1 X2)
| C_Con : ∀ c Xs, (∀ X, In X Xs → C X) → C (Con c Xs).

Lemma C_not_R0 : ∀ X1 X2, R0 X1 X2 → C X1 → False.

Lemma R1_Con_elem : ∀ c Xs2 Xs1, R1s Xs1 Xs2 → R1 (Con c Xs1) (Con c Xs2).

Lemma not_C_and_R1 : ∀ X1, C X1 → ∀ X2, R1 X1 X2 → False.

Lemma R1_or_C : ∀ X1, (∃ X2 : Entity, R1 X1 X2) ∨ C X1.

Lemma R2_Trans : ∀ X1 X2 X3, R2 X1 X2 → R2 X2 X3 → R2 X1 X3.

Lemma R2_App_1 : ∀ X1 X2 X, R2 X1 X2 → R2 (App X1 X) (App X2 X).

Lemma R2_App_2 : ∀ X1 X2 X, R2 X1 X2 → R2 (App X X1) (App X X2).

Lemma R2_Sub_1 : ∀ v X1 X2 X, R2 X1 X2 → R2 (Sub v X1 X) (Sub v X2 X).

Lemma R2_Sub_2 : ∀ v X1 X2 X, R2 X1 X2 → R2 (Sub v X X1) (Sub v X X2).

Lemma R2s_length : ∀ Xs1 Xs2, R2s Xs1 Xs2 → length Xs1 = length Xs2.

Lemma R2s_Trans : ∀ Xs1 Xs2 Xs3, R2s Xs1 Xs2 → R2s Xs2 Xs3 → R2s Xs1 Xs3.

Lemma R2_Con : ∀ c Xs1 Xs2, R2s Xs1 Xs2 → R2 (Con c Xs1) (Con c Xs2).

Lemma R2_or_C : ∀ X1, (∃ X2 : Entity, R2 X1 X2) ∨ C X1.

Lemma C_R2 : ∀ X, (∃ X' : Entity, C X' ∧ (R2 X X' ∨ X = X')).

Lemma R2_implies_R3 : ∀ X1 X2, R2 X1 X2 → R3 X1 X2.

Lemma R3_App_1 : ∀ X1 X2 X, R3 X1 X2 → R3 (App X1 X) (App X2 X).

Lemma R3_App_2 : ∀ X1 X2 X, R3 X1 X2 → R3 (App X X1) (App X X2).

Lemma R3_Sub_1 : ∀ v X1 X2 X, R3 X1 X2 → R3 (Sub v X1 X) (Sub v X2 X).

Lemma R3_Sub_2 : ∀ v X1 X2 X, R3 X1 X2 → R3 (Sub v X X1) (Sub v X X2).

Inductive R3s : list Entity → list Entity → Prop :=
| R3s_R1s : ∀ Xs1 Xs2, R1s Xs1 Xs2 → R3s Xs1 Xs2
| R3s_refl : ∀ Xs, R3s Xs Xs
| R3s_symm : ∀ Xs1 Xs2, R3s Xs1 Xs2 → R3s Xs2 Xs1
| R3s_trans : ∀ Xs1 Xs2 Xs3, R3s Xs1 Xs2 → R3s Xs2 Xs3 → R3s Xs1 Xs3.

Lemma R3s_1 : ∀ X Xs1 Xs2, R3s Xs1 Xs2 → R3s (X :: Xs1) (X :: Xs2).

Lemma R3s_2 : ∀ Xs X1 X2, R3 X1 X2 → R3s (X1 :: Xs) (X2 :: Xs).

Lemma R3s_ext : ∀ (Xs : list Entity) f1 f2, (∀ X, In X Xs → R3 (f1 X) (f2 X)) → R3s (map f1 Xs) (map f2 Xs).

Lemma R3_Con : ∀ c Xs1 Xs2, R3s Xs1 Xs2 → R3 (Con c Xs1) (Con c Xs2).

Lemma all2_R_refl : ∀ Xs, all2 _ _ R Xs Xs.

Lemma R0_implies_R : ∀ X1 X2, R0 X1 X2 → R X1 X2.

Lemma Entity_ind' : ∀ (X : Entity) (P : Entity → Prop)
(HApp : ∀ X1 X2, P X1 → P X2 → P (App X1 X2))
(HId : P Id)
(HVar : ∀ a, P (Var a))
(HSub : ∀ v X1 X2, P X1 → P X2 → P (Sub v X1 X2))
(HCon : ∀ c Xs, all _ P Xs → P (Con c Xs)), P X.

Lemma R1_implies_R : ∀ X1 X2, R1 X1 X2 → R X1 X2.

Lemma R2_implies_R : ∀ X1 X2, R2 X1 X2 → R X1 X2.

Lemma R3_implies_R : ∀ X1 X2, R3 X1 X2 → R X1 X2.

Lemma R3_App_assoc : ∀ X1, C X1 → ∀ X2, C X2 → ∀ X3, C X3 → R3 (App X1 (App X2 X3)) (App (App X1 X2) X3).

Lemma R_implies_R3 : ∀ X1 X2, R X1 X2 → R3 X1 X2.

Theorem R_equiv_R3 : ∀ X1 X2, R X1 X2 ↔ R3 X1 X2.