Manipulating Trees with Hidden Labels

Luca Cardelli[a,1], Philippa Gardner[b,2] and Giorgio Ghelli[c,3]

[a] *Microsoft Research, Cambridge, UK*
[b] *Department of Computing, Imperial College London, UK*
[c] *Dipartimento di Informatica, Università di Pisa, Italy*

Abstract

We define an operational semantics and a type system for manipulating semistructured data that contains hidden information. The data model is simple labeled trees with a hiding operator. Data manipulation is based on pattern matching, with types that track the use of hidden labels.

Keywords: Operational Semantics, Type Systems, Semistructured Data, Information Hiding.

1 Introduction

1.1 Languages for Semistructured Data

Semistructured data [1], such as XML, is inspiring a new generation of programming and query languages based on more flexible type systems [29, 5, 6, 17]. Traditional type systems are grounded on mathematical constructions such as cartesian products, disjoint unions, function spaces, and recursive types. The type systems for semistructured data, in contrast, resemble grammars or logics, with associative products, untagged unions and Kleene star operators. The theory of formal languages, for strings and trees for example, provides a wealth of ready results, but it does not account in particular for functions. Some integration of the two approaches to type systems is necessary.

Towards this integration, Pierce has introduced a typed pattern-matching language XDuce for analysing trees [29], where the types are regular expression types which extend the DTDs associated with XML. Castagna extended this work in his language CDuce incorporating first-order logical analysis in the types [5]. Calcagno, Cardelli and Gordon go one step further by using Ambient Logic formulae as the types [12]. All these languages are designed to manipulate

1. Email: luca@microsoft.com
2. Email: pg@doc.ic.ac.uk
3. Email: ghelli@di.unipi.it

simple tree data. In this paper, we introduce a substantial typed pattern-matching language for manipulating more complex trees, building on the work in [12] by incorporating the notion of *private* location.

We believe that the distinction between public and private locations is a natural feature of semistructured data. A public location can be accessed by anyone: for example, a URI is a public location. By contrast, a private location has restricted access: for example, the XML identifiers in a XML document are typically regarded as private to that document. In this paper, we study a more compositional tree model, consisting of both public and private names: for example, if we regard XML as a data store that can be manipulated rather than as a static document, then this distinction becomes important; another example is a distributed bibliography with public locations for accessing published work and private locations for hiding work in progress. We model public and private locations using names and name hiding, applying known concepts arising from the π-calculus [30] to this non-standard setting of semistructured data. Our approach treats the identity of a private name as unimportant, provided the distinctions between it and other (public or private) names are preserved.

Our typed pattern-matching language pulls together techniques for dealing with hidden names from many sources. Pitts argues that the transposition [33], or swapping, of names is more fundamental than substitution for manipulating free and bound names, and with Gabbay uses transpositions to develop the typed functional language FreshML [24]. Transpositions play a significant role in our language. Gabbay and Pitts also invented a freshness quantifier for reasoning about bound names [25,33]. Caires and Cardelli [10,11] use this freshness quantifier to develop a logic for reasoning about the pi-calculus, extending the ideas of Ambient Logic and previous work of Dam [21] to account for name hiding. We use this freshness quantifier in a substantial way, in particular applying key results of [10,11] in our treatment of transpositions. We also use a notion of dependent types for names, which we believe was first shown to be tractable in [28].

In the remainder of the introduction, we give a detailed description of the key features of our pattern-matching language. This paper is an extension of our conference paper [15], incorporating a considerable amount of additional detail.

1.2 Data Model and Pattern Matching Language

We use the simplest tree model possible for studying public and private locations. Our trees are simple rooted trees, with the locations given by edge labels (names) which may be hidden. Locations are not unique, since this would require factoring in partial composition of trees which is a minor, but irritating, addition to the theory. Our ideas will apply to locations in many other data models, such as graph nodes, addresses in heaps, and identifiers in structures combining trees and graphical links such as our trees-with-pointers model [13,16] and XML (XML identifiers and idrefs). In fact, we believe the manipulation of

hidden resources in all these data structures is fundamental.

The data model we investigate here has the following constructors. Essentially, we extend a simple tree model (such as XML) in a general and orthogonal way with a hiding operator.

0	the tree consisting of a single root node;
$n[P]$	a tree with a single edge from the root, labeled n, leading to P;
$P \mid Q$	the root-merge of two trees (commutative and associative);
$(\nu n)P$	a tree P where the label n is hidden/private/restricted. As in π-calculus, we call *restriction* the act of hiding a name.

Trees are inspected by pattern matching. For example, program (1) below inspects a tree t having shape $n[P] \mid Q$, for some P,Q, and produces $P \mid m[Q]$. Here n,m are constant (public) labels, x,y are pattern variables, and $\mathbf{T}$ is both the pattern that matches any tree and the type of all trees. It is easy to imagine that, when parameterized in t, this program should have the type indicated.

match t ***as*** $(n[x{:}\mathbf{T}] \mid y{:}\mathbf{T})$ ***then*** $(x \mid m[y])$ (1)
transforms a tree $t = n[P] \mid Q$ into $P \mid m[Q]$
expected typing: $(n[\mathbf{T}] \mid \mathbf{T}) \to (\mathbf{T} \mid m[\mathbf{T}])$

Using the same pattern match as in (1), let us now remove the public label n and insert a private one, p, that is created and bound to the program variable z at "run-time":

match t ***as*** $(n[x{:}\mathbf{T}] \mid y{:}\mathbf{T})$ ***then*** $(\nu z)\ (x \mid z[y])$ (2)
transforms $t = n[P] \mid Q$ into $(\nu p)\ (P \mid p[Q])$ for a fresh label p
expected typing: $(n[\mathbf{T}] \mid \mathbf{T}) \to (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

In our type system, the *hidden name quantifier* H is the type construct corresponding to the data construct ν [10]. More precisely, $\mathsf{H}z.\mathcal{A}$ means that there is a hidden label p denoted by the variable z, such that the data is described by $\mathcal{A}\{z{\leftarrow}p\}$. (Scope extrusion [30] makes the relationship non trivial, see Sections 2 and 4.) Because of the $\mathsf{H}z.\mathcal{A}$ construct, types contain name variables; that is, types are dependent on names.

The first two examples pattern match on the public name n. Suppose instead that we want to find and manipulate private names. The following example is similar to (2), except that now a private label p from the data is matched and bound to the variable z.

match t ***as*** $((\nu z)\ (z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))$ ***then*** $x \mid z[y]$ (3)
transforms $t = (\nu p)(p[P] \mid Q)$ into $(\nu p)(P \mid p[Q])$
expected typing: $(\mathsf{H}z.\ (z[\mathbf{T}] \mid \mathbf{T})) \to (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

The restriction (νp) in the result is not apparent in the program: it is implicitly applied by a match that opens a restriction, so that the restricted name does not escape.

As the fourth and remaining case, we convert a private name in the data into a public one. The only change from (3) is a public name m instead of z in the result:

match t **as** $((\nu z)\ z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))$ **then** $x \mid b[y]$ (4)
transforms $t = (\nu n)\ (n[P] \mid Q)$ into $(\nu n)(P \mid b[Q])$
expected typing: $(\mathsf{H}z.\ z[\mathbf{T}] \mid \mathbf{T}) \to (\mathsf{H}z.\ \mathbf{T} \mid b[\mathbf{T}])$

This program transforms a tree of the shape $(\nu n)\ (n[P] \mid Q)$ into $P \mid b[Q]$; provided, though, that P,Q do not contain n. In general we cannot eliminate the (νn) restriction from the result, so the result will have the form $(\nu n)(P \mid b[Q])$ even though z appears nowhere after **then** in the program text. Again, such restriction is automatically reapplied as a result of opening it via pattern matching. The resulting type should now be:

$$(\mathsf{H}z.\ z[\mathbf{T}] \mid \mathbf{T}) \to (\mathsf{H}z.\ \mathbf{T} \mid b[\mathbf{T}])$$

Where $\mathsf{H}z.\ \mathbf{T} \mid b[\mathbf{T}]$ is not the same as $\mathbf{T} \mid b[\mathbf{T}]$ even though z appears nowhere in the rest of the type, because the name denoted by z may appear in the data denoted by the two $\mathbf{T}$.

As an example of an incorrectly typed program consider the following attempt to assign a simpler type to the result of example (4), via a typed ***let*** biding:

$$\textbf{\textit{let}}\ w : (\mathbf{T} \mid m[\mathbf{T}]) = \textbf{\textit{match}}\ t\ \textbf{\textit{as}}\ ((\nu z)\ (z[x{:}\mathbf{T}] \mid y{:}\mathbf{T}))\ \textbf{\textit{then}}\ x \mid m[y]$$

Here we would have to check that $\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}])$ is compatible with $(\mathbf{T} \mid m[\mathbf{T}])$. This would work if we could first show that $\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}])$ is a subtype of $(\mathsf{H}z.\ \mathbf{T}) \mid (\mathsf{H}z.\ m[\mathbf{T}])$, and then simplify. But such a subtyping does not hold since, e.g., $(\nu p)(p[0] \mid m[p[0]])$ matches the former type but not the latter, because the restriction (νp) cannot be distributed.

1.3 Transpositions

So far, we have illustrated the manipulation of individual private or public names by pattern matching and data constructors. However, we may want to replace throughout a whole data structure a public name with another, or a public one with a private one, or vice versa. We could do this by recursive analysis, but it would be very difficult to reflect what has happened in the type structure, likely resulting in programs of type $\mathbf{T}{\to}\mathbf{T}$. So, we introduce a *transposition* facility as a primitive operation, and as a corresponding type operator. In the simplest case, if we want to transpose (exchange) a name n with a name m in a data structure t we write $t(n{\leftrightarrow}m)$. If t has type $\mathscr{A}$, then $t(n{\leftrightarrow}m)$ has type $\mathscr{A}(n{\leftrightarrow}m)$. We define rules to manipulate type level transpositions; for example we derive that, as types, $n[\mathbf{0}](n{\leftrightarrow}m) = m[\mathbf{0}]$.

Transposition types are interesting when exchanging public and private labels. Consider the following program and its initial syntax-driven type:

$$\lambda x{:}n[\mathbf{T}].\ (\nu z)\ x(m{\leftrightarrow}z) \quad : \quad n[\mathbf{T}]{\to}\mathsf{H}z.n[\mathbf{T}](m{\leftrightarrow}z) \quad (=\ n[\mathbf{T}]{\to}n[\mathbf{T}]\ ) \qquad (5)$$

This program takes data of the form $n[P]$, creates a fresh label p denoted by z, and swaps the public m with the fresh p in $n[P]$, to yield $(\nu p)n[P](m{\leftrightarrow}p)$, where the fresh p has been hidden in the result. Since n is a constant different from m, and p is fresh, the result is in fact $(\nu p)n[P(m{\leftrightarrow}p)]$. The result type can be simi-

larly simplified to $\mathsf{H}z.n[\mathbf{T}(m{\leftrightarrow}z)]$. Now, swapping two names in the set of all trees, $\mathbf{T}$, produces again the set of all trees. Therefore, the result type can be further simplified to $\mathsf{H}z.n[\mathbf{T}]$. We then have that $\mathsf{H}z.n[\mathbf{T}] = n[\mathbf{T}]$, since a restriction can be pushed through a public label, where it is absorbed by $\mathbf{T}$. Therefore, the type of our program is $n[\mathbf{T}]{\to}n[\mathbf{T}]$.

Since we already need to handle name-dependent types, we can introduce, without much additional complexity, a dependent function type $\Pi w.\ \mathcal{A}$. This is the type of functions $\lambda w{:}\mathbf{N}.t$ that take a name m (of type $\mathbf{N}$) as input, and return a result of type $\mathcal{A}\{w{\leftarrow}m\}$. We can then write a more parametric version of example (5), where the constant n is replaced by a name variable w which is a parameter:

$$\lambda w{:}\mathbf{N}.\ \lambda x{:}w[\mathbf{T}].\ (\nu z)\ x(m{\leftrightarrow}z) \quad :\quad \Pi w.\ (w[\mathbf{T}] \to \mathsf{H}z.\ w[\mathbf{T}](m{\leftrightarrow}z)) \qquad (6)$$

Now, the type $\mathsf{H}z.\ w[\mathbf{T}](m{\leftrightarrow}z)$ simplifies to $\mathsf{H}z.\ w(m{\leftrightarrow}z)[\mathbf{T}]$, but no further, since m can in fact be given for w, in which case it would be transposed to the private z.

Transpositions are emerging as a unifying and simplifying principle in the formal manipulation of binding operators [33]. If some type-level manipulation of names is of use, then transpositions seem a good starting point.

1.4 General Structure

The type system of our calculus has, at the basis, tree types. Function types are built on top of the tree types in standard higher-order style. This separation between trees and functions greatly simplifies our technical development. The tree types are unusual: they are the formulas of a spatial logic. Therefore, we can write types such as these that combine logical, structural and functional operators:

$\mathbf{T} \to \neg\mathbf{0}$	for any input produce a non-zero output
$((\mathcal{A} \wedge \neg\mathbf{0}) \mid n[\mathcal{B}]) \to (n[\mathcal{A}] \mid \mathcal{B})$	a structural transformation on trees

A subtyping relation is defined between types. On tree types, subtyping is defined as a sound approximation to logical implication; that is, $\mathcal{A} <: \mathcal{B}$ implies $\mathcal{A} \Rightarrow \mathcal{B}$. For example, we have the basic subtyping $\mathcal{A} <: \mathbf{T}$ for all $\mathcal{A}$. Subtyping is then extended to function types by the usual contravariant rule. This means that a logical implication check needs to be used during static typechecking, whenever a subtyping check is required.

Tree data is manipulated via pattern matching constructs that perform "run-time type checks". Since tree types are formulas, we have the full power of the logic to express the pattern matching conditions. Those run-time type checks are executed as run-time satisfaction checks ($\models$). For example, one of our matching construct is a test to see whether the value denoted by expression t has type $\mathcal{A}$:

$$t?(x{:}\mathcal{A}).u,v$$

This construct first computes the tree P denoted by the expression t, and then performs a test $P \models \mathcal{A}$. If the test is successful, it binds P to x and executes u; oth-

erwise it binds P to x and executes v. The variable x can be used both inside u and v, but in u it has type $\mathcal{A}$, while in v it has type $\neg\mathcal{A}$.

For example, the following program inspects an arbitrary tree (i.e., anything of type **T**). If the tree is 0 it returns the tree $a[0]$, otherwise it returns the input tree. Hence the result is never 0, and the result type can be set to $\neg\mathbf{0}$.

$$\lambda x{:}\mathbf{T}.\ x?(y{:}\mathbf{0}).\ a[0],\ y \quad : \quad \mathbf{T} \to \neg\mathbf{0}$$

To summarize, our formulas are used as a very expressive type system for tree data, within a typed λ-calculus. A satisfaction algorithm is used to analyze data at run time, and an entailment algorithm is needed during static typechecking. In absence of polymorphism, type are ground, which facilitates matters. With name-dependent types (necessary to handle a restriction construct), types can contain variables of sort Name, but this can be handled without much difficulty. At run-time, all values and types are ground. As usual, the type system checks whether an *open* term has a type: it can do so without additional difficulties, even though the basic satisfaction test we have is for *closed* terms (i.e., values).

1.5 Related and Future Work

It should be clear from Section 1.2 that sophisticated type-level manipulations are required for our data model, involving transposition types (which seem to be unique to our work), hiding quantifiers, and dependent types. Furthermore, we work in the context of a data model and type system that is "non-structural", both in the sense of supporting grammar-like types (with $\wedge\ \vee\ \neg$) and in the sense of supporting π-calculus-style extruding scopes. In both these aspects we differ from FreshML [34], although we base much of our development on the same foundations [33]. Our technique of automatically rebinding restrictions sidesteps some complex issues in the FreshML type system, and yet seems to be practical for many examples. FreshML uses "apartness types" $\mathcal{A}\#w$, which can be used to say that a function takes a name denoted by w and a piece of data $\mathcal{A}$ that does not contain that name. We can express that idiom differently as $\Pi w.\ (\mathcal{A} \wedge \neg\copyright w) \to \mathcal{B}$, where $\copyright w$ [10] means "contains free the name denoted by w".

Our calculus is based on a pattern matching construct that performs runtime type tests; in this respect, it is similar to the XML manipulation languages XDuce [29] and CDuce [5]. However, those languages do not deal with hidden names, whose study is our main goal. XDuce types are based on tree grammars: they are more restrictive than ours but are based on well-known algorithms. CDuce types are in some aspects richer than ours: they mix the logical and functional levels that we keep separate; such mixing would not easily extend to our $\mathsf{H}x.\mathcal{A}$ types. Other differences stem from the data model (our $P \mid Q$ is commutative), and from auxiliary programming constructs.

The database community has defined many languages to query semistructured data [1,3,6,8,17,19,22,23], but they do not deal with hidden names. The theme of hidden identifiers (OIDs) has been central in the field of object-oriented

database languages [2, 4]. However, the debate there was between languages where OIDs are hidden to the user, and lower-level languages where OIDs are fully visible. The second approach is more expressive but has the severe problem that OIDs lose their meaning once they are exported outside their natural scope. We are not aware of any proposal with operators to define a scope for, reveal, and rehide private identifiers, as we do in our calculus.

In TQL [17], the semistructured query language closest to this work, a programmer writes a logical formula, and the system chooses a way to retrieve all pieces of data that satisfy that formula. In our calculus, such formulas are our tree types, but the programmer has to write the recursion patterns that collect the result (as in Section 8). The TQL approach is best suited to collecting the subtrees that satisfy a condition, but the approach we explore here is much more expressive; for example, we can apply transformations at an arbitrary depth, which is not possible in TQL. Other query-oriented languages, such as XQuery [6], also support structural recursion.

As a major area of future work, our subtyping relation is not prescribed in detail here (apart for the non-trivial subtypings coming from transposition equivalence). Our type system is parameterized by an unspecified set of ValidEntailments, which are simply assumed to be sound for typing purposes. The study of related subtyping relations (a.k.a. valid logical implications in spatial logics [11]) is in rapid development. The work in [12] provides a complete subtyping algorithm for ground types (i.e. not including $\mathsf{H}z.\mathscr{A}$), and other algorithms are being developed that include Kleene star [20]. Such theories and algorithms could be taken as the core of our ValidEntailments. But adding quantifiers is likely to lead to either undecidability or incompleteness. In the middle ground, there is a collection of sound and practical inclusion rules [10,11] that can be usefully added to the ground subtyping relation (e.g., $\mathsf{H}z.n[\mathscr{A}] <: n[\mathsf{H}z.\mathscr{A}]$ for example (5)). By parameterizing over the ValidEntailments, we show that these issues are orthogonal to the handling of transpositions and hiding.

2 Tree Values

Our programs manipulate *values*; either *name values* (from a countable set of names Λ), *tree values*, or *function values* (i.e., closures). In this section, we describe tree values and name transpositions.

Definition 2-1 (Tree Values)

Λ	Names: a countable set of names $n, m, p, \ldots$
$P,Q,R ::=$	Tree values
0	void
$P \mid Q$	composition
$n[P]$	location
$(\nu n)P$	restriction

Definition 2-2 (Name Occurrences and Free Names)

All names: $na(P)$	Free names: $fn(P)$
$na(0) \triangleq \{\}$	$fn(0) \triangleq \{\}$
$na(P \mid Q) \triangleq na(P) \cup na(Q)$	$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$
$na(n[P]) \triangleq \{n\} \cup na(P)$	$fn(n[P]) \triangleq \{n\} \cup fn(P)$
$na((\nu n)P) \triangleq \{n\} \cup na(P)$	$fn((\nu n)P) \triangleq fn(P) - \{n\}$

We define an *actual transposition* operation on tree values, $P\bullet(m\leftrightarrow m')$, that blindly swaps free and bound names m,m' within P. (By an *actual* transposition we mean an operation on terms, while a *formal* transposition is a syntactic construct within terms.) The interaction of transpositions with binders such as $(\nu n)P$ supports a general formal treatment of bound names [33].

Definition 2-3 (Actual Transposition of Names and Tree Values)

$n\bullet(n\leftrightarrow m) = m$	$0\bullet(m\leftrightarrow m') = 0$
$n\bullet(m\leftrightarrow n) = m$	$(P \mid Q)\bullet(m\leftrightarrow m') = P\bullet(m\leftrightarrow m') \mid Q\bullet(m\leftrightarrow m')$
$n\bullet(m\leftrightarrow m') = n$ if $n\neq m$ and $n\neq m'$	$n[P]\bullet(m\leftrightarrow m') = n\bullet(m\leftrightarrow m')[P\bullet(m\leftrightarrow m')]$
	$((\nu n)P)\bullet(m\leftrightarrow m') = (\nu n\bullet(m\leftrightarrow m'))P\bullet(m\leftrightarrow m')$

Lemma 2-4 (Distribution of Transpositions)

(1) $p\bullet(n\leftrightarrow n')\bullet(m\leftrightarrow m') = p\bullet(m\leftrightarrow m')\bullet(n\bullet(m\leftrightarrow m')\leftrightarrow n'\bullet(m\leftrightarrow m'))$
$p\bullet(n\leftrightarrow n')\bullet(m\leftrightarrow m') = p\bullet(m\bullet(n\leftrightarrow n')\leftrightarrow m'\bullet(n\leftrightarrow n'))\bullet(n\leftrightarrow n')$

(2) $P\bullet(n\leftrightarrow n')\bullet(m\leftrightarrow m') = P\bullet(m\leftrightarrow m')\bullet(n\bullet(m\leftrightarrow m')\leftrightarrow n'\bullet(m\leftrightarrow m'))$
$P\bullet(n\leftrightarrow n')\bullet(m\leftrightarrow m') = P\bullet(m\bullet(n\leftrightarrow n')\leftrightarrow m'\bullet(n\leftrightarrow n'))\bullet(n\leftrightarrow n')$

Transpositions are used in the definition of α-congruence and capture-avoiding substitution. Structural congruence is analogous to the standard definition for π-calculus [30]; the "scope extrusion" rule for ν over -|- is written in an equivalent equational style. Over the tree values, we define a structural congruence relation ≡ that factors out the equivalence laws for | and 0, and the scoping laws for restriction.

Definition 2-5 (Structural Congruence on Tree Values)

α-congruence, $\equiv_\alpha$, is the least equivalence relation on tree values such that:
$P \equiv_\alpha P' \wedge Q \equiv_\alpha Q' \Rightarrow (P \mid Q) \equiv_\alpha (P' \mid Q')$
$P \equiv_\alpha P' \Rightarrow n[P] \equiv_\alpha n[P']$
$P \equiv_\alpha P' \Rightarrow ((\nu n)P) \equiv_\alpha ((\nu n)P')$
$(\nu n)P \equiv_\alpha (\nu m)(P\bullet(n\leftrightarrow m))$ where $m\notin na(P)$

N.B.: This notion of α-congruence can be shown equivalent to the standard one. Structural congruence, ≡, is the least equivalence relation on tree values such that:

$P \equiv P' \wedge Q \equiv Q' \Rightarrow (P \mid Q) \equiv (P' \mid Q')$
$P \equiv P' \Rightarrow n[P] \equiv n[P']$
$P \equiv P' \Rightarrow ((\nu n)P) \equiv ((\nu n)P')$

$P \equiv_\alpha Q \Rightarrow P \equiv Q$	$(\nu n)0 \equiv 0$
$P \mid Q \equiv Q \mid P$	$(\nu n)m[P] \equiv m[(\nu n)P]$ if $n \neq m$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$(\nu n)(P \mid (\nu n)Q) \equiv ((\nu n)P) \mid ((\nu n)Q)$
$P \mid 0 \equiv P$	$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$

Lemma 2-6 (Structural Congruence Properties)

If $P \equiv Q$ then $fn(P)=fn(Q)$
If $P \equiv Q$ then $P\bullet\{m\leftrightarrow m'\} \equiv Q\bullet\{m\leftrightarrow m'\}$.

Definition 2-7 (Free Name Substitution on Tree Values)

$0\{n\leftarrow m\} = 0$
$(P \mid Q)\{n\leftarrow m\} = P\{n\leftarrow m\} \mid Q\{n\leftarrow m\}$
$p[P]\{n\leftarrow m\} = p\{n\leftarrow m\}[P\{n\leftarrow m\}]$
$((\nu p)P)\{n\leftarrow m\} = (\nu q)((P\bullet\{p\leftrightarrow q\})\{n\leftarrow m\})$ for $q \notin n\alpha((\nu p)P)\cup\{n,m\}$

N.B.: different choices of q in the last clause, lead to α-congruent results.

3 Terms and High Values

3.1 Syntax

Our λ-calculus is stratified in terms of *low types* and *high types*. The low types are the tree types and the type of names, **N**. (Basic data types such as integers could be added to low types.) The novel aspects of the type structure are the richness of the tree types, which come from the formulas of spatial logics [18, 10], and the presence of transposition types. We then have higher types over the low types: function types and name-dependent types. The precise meaning of types is given in Section 4.

The same stratification holds on terms, which can be of low or high type, as is more apparent in the operational semantics of Section 5 and in the type rules of Section 7.

Definition 3-1 (Syntax)

Var	Variables: a countable set of variables x, y, z, ...
$\mathcal{N},\mathcal{M}$ ::=	Name Expressions
x	name variable
n	name constant
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	name transposition
$\mathcal{A},\mathcal{B}$ ::=	Tree Types
0	void
$\mathcal{N}[\mathcal{A}]$	location
$\mathcal{A} \mid \mathcal{B}$	composition
$\mathsf{H}x.\mathcal{A}$	hiding quantifier (x:**N**)
$©\mathcal{N}$	occurence

$\mathbf{F}$	false
$\mathcal{A} \wedge \mathcal{B}$	conjunction
$\mathcal{A} \Rightarrow \mathcal{B}$	implication
$\mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	transposition
$\mathcal{F},\mathcal{G},\mathcal{H}$ ::=	High Types
$\mathcal{A}$	tree types
$\mathbf{N}$	name type
$\mathcal{F}\rightarrow\mathcal{G}$	function types ($\mathcal{F}\neq\mathbf{N}$)
$\Pi x.\mathcal{G}$	name dependent function types (x:$\mathbf{N}$)
t,u,v ::=	Terms
0	void
$\mathcal{N}[u]$	location
$t \mid u$	composition
$(\nu x)t$	restriction (binding x)
$t\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$	term transposition
$t\div(\mathcal{N}[y{:}\mathcal{A}]).u$	location match (binding y only)
$t\div(x{:}\mathcal{A} \mid y{:}\mathcal{B}).u$	composition match (binding x,y)
$t\div((\nu x)y{:}\mathcal{A}).u$	restriction match (binding x,y) (auto-rebind u)
$t?(x{:}\mathcal{A}).u,v$	tree type test (binding x)
x	high variable
$\mathcal{N}$	name expression (incl. name variables)
$\lambda x{:}\mathcal{F}.t$	function (binding x)
$t(u)$	application

Underlined variables indicate binding occurrences. The scoping rules should be clear: in location match y scopes u; in composition match x and y scope u; in restriction match x scopes $\mathcal{A}$ and u, and y scopes u; in tree type test x scopes u and v.

The various matching constructs can be combined, for example, to obtain a more convenient case statement:

case t ***of***	analyze the value R of t:
$0.\ u_1,$	if $R \equiv 0$, run u_1, else
$n[x{:}\mathcal{A}].\ u_2,$	if $R \equiv n[P]$ and $P \vDash \mathcal{A}$, bind P to x and run u_2, else
$(x{:}\mathcal{A} \mid y{:}\mathcal{B}).\ u_3,$	if $R \equiv P \mid Q$ and $P \vDash \mathcal{A}$, $Q \vDash \mathcal{B}$, bind P to x, Q to y and run u_3,
$(\nu x)y{:}\mathcal{A}.\ u_4,$	else if $R \equiv (\nu n)P$ and $P \vDash \mathcal{A}\{x\leftarrow n\}$, bind n to x and P to y then run u_4 to obtain a result Q and return $(\nu n)Q$ (here n is chosen fresh)
else u_5	else run u_5
$\triangleq$	can be translated as:

$t?(z_1{:}\ \mathbf{0}).\ u_1,$
$t?(z_2{:}\ n[\mathcal{A}]).\ z_2\div(n[x{:}\mathcal{A}]).u_2,$
$t?(z_3{:}\ \mathcal{A} \mid \mathcal{B}).\ z_3\div(x{:}\mathcal{A} \mid y{:}\mathcal{B}).u_3,$
$t?(z_4{:}\ \mathsf{H}x.\mathcal{A}).\ z_4\div((\nu x)y{:}\mathcal{A}).u_4{:}\mathcal{B},$
u_5

We define name sets, such as $na(\mathcal{A})$, and actual transpositions on all syntax, such as $t\bullet(n\leftrightarrow m)$, in the obvious way (there are no name binders in the syntax). We also define free-variable sets $fv(\text{-})$ on all syntax (based on the mentioned

binding occurrences), and capture-avoiding substitutions of name expressions for variables.

Definition 3-2 (Actual Transposition on All Syntax)

$\mathcal{N}\bullet(n\leftrightarrow m)$, $\mathcal{A}\bullet(n\leftrightarrow m)$, $\mathcal{T}\bullet(n\leftrightarrow m)$, and $t\bullet(n\leftrightarrow m)$ transpose the names n and m in $\mathcal{N}$, $\mathcal{A}$, $\mathcal{T}$, and t.

Definition 3-3 (Variables Substitution on Names and Types)

$\mathcal{N}\{x\leftarrow\mathcal{M}\}$, $\mathcal{A}\{x\leftarrow\mathcal{M}\}$, and $\mathcal{T}\{x\leftarrow\mathcal{M}\}$, are the capture-avoiding substitutions of $\mathcal{M}$ for x in $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{T}$.

Name expressions, tree types, and terms all include *(formal) transposition* operations that are part of the syntax; they represent (actual) transpositions on data, indicated by the $\bullet$ symbol.

The tree types are formulas in a spatial logic, so we can derive the standard types (formulas) for negation $\neg\mathcal{A} \triangleq \mathcal{A}\Rightarrow\mathbf{F}$ and disjunction $\mathcal{A}\vee\mathcal{B} \triangleq \neg(\neg\mathcal{A}\wedge\neg\mathcal{B})$.

The terms include a standard λ-calculus fragment, the basic tree constructors, and some matching operators for analyzing tree data. The *tree type test* construct (distinguished by the character '?') performs a run-time check to see whether a tree has a given type: if tree t satisfies type $\mathcal{A}$ then u is run with x of type $\mathcal{A}$ bound to t; otherwise v is run with x of type $\neg\mathcal{A}$ bound to t. In addition, one needs matching constructs (distinguished by the character '÷') to decompose the tree: *composition match* splits a tree in two components, *location match* strips an edge from a tree, and *restriction match* inspects a hidden label in a tree. A *zero match* is redundant because of the tree type test construct. These multiple matching constructs are designed to simplify the operational semantics and the type rules. In practice, one would use a single case statement with patterns over the structure of trees, but this can be encoded.

In the quantifier $\mathsf{H}x.\mathcal{A}$ and in the restriction match construct, the type $\mathcal{A}$ is dependent on variable x (denoting a hidden name). This induces the need for handling dependent types, and motivates the $\Pi x.\mathcal{G}$ dependent function type constructor. The type dependencies, however, are restricted to name variables, which may be replaced only by name expressions (that is, not by general computations on names). Because of this, these dependent types are relatively easy to handle.

3.2 High Values

High values are the results of evaluating terms; they can be either names, trees, or closures. Name transpositions are defined on all values. Closures are triples of a term t (Section 3.1) with respect to an input variable x (essentially, $\lambda x.t$) and a *stack* for free variables. A stack ρ is a list of bindings of variables to values.

Definition 3-4 (High Values and Stacks)

(1) A stack is a finite map ρ from variables in *Var* to high values *F*. We write $\emptyset$ for the empty map, and $\rho[x \leftarrow F]$ for the map that is ρ except for mapping *x* to *F*.
(2) High values are defined as follows:

F,G,H ::=	High Values
n	name values
P	tree values
$\langle \rho, x, t \rangle$	function values

(3) Transpositions of function values and stacks are defined as follows:

$\langle \rho, x, t \rangle \bullet (n \leftrightarrow n') \triangleq \langle \rho \bullet (n \leftrightarrow n'), x, t \bullet (n \leftrightarrow n') \rangle$
$\emptyset \bullet (n \leftrightarrow n') \triangleq \emptyset$
$\rho[x \leftarrow F] \bullet (n \leftrightarrow n') \triangleq \rho \bullet (n \leftrightarrow n')[x \leftarrow F \bullet (n \leftrightarrow n')]$ $(x \notin dom(\rho))$

Definition 3-5 (Name Occurrences and Free Names)

All names: $na(F)$	Free names: $fn(F)$
$na(n) \triangleq \{n\}$	$fn(n) \triangleq \{n\}$
$na(P)$: see tree values	$fn(P)$: see tree values
$na(\langle \rho, x, t \rangle) \triangleq na(t) \cup na(\rho)$	$fn(\langle \rho, x, t \rangle) \triangleq fn(t) \cup fn(\rho)$
$na(\rho) \triangleq \bigcup_{x \in dom(\rho)} na(\rho(x))$	$fn(\rho) \triangleq \bigcup_{x \in dom(\rho)} fn(\rho(x))$

4 Satisfaction

The satisfaction relation, written $\models$, relates values to types, and thus provides the semantic meaning of typing that is enforced by the type system of Section 7. For type constructs such as $\wedge$ and $\Rightarrow$, this is related to the standard notion of satisfaction from logic. Over tree types we have essentially the satisfaction relation studied in [18, 10], extended to hiding and transpositions. Satisfaction is then generalized to high types, where it is defined in terms of the operational semantics $\Downarrow_\rho$ of Section 5, which is defined in terms of the satisfaction relation over tree types only.

Definition 4-1 (Satisfaction)

On Name Expressions: $n \models_N \mathcal{N}$, for $\mathcal{N}$ closed (no free variables), is defined by:

$n \models_N m$	iff $m = n$
$n \models_N \mathcal{N}(\mathcal{M} \leftrightarrow \mathcal{M}')$	iff $\exists m,m'.\ m \models_N \mathcal{M}$ and $m' \models_N \mathcal{M}'$ and $n \bullet (m \leftrightarrow m') \models_N \mathcal{N}$

On Tree Types: $P \models_T \mathcal{A}$, for $\mathcal{A}$ closed, is defined by:

$P \models_T \mathbf{0}$	iff $P \equiv 0$
$P \models_T \mathcal{N}[\mathcal{A}]$	iff $\exists n,P'.\ n \models_N \mathcal{N}$ and $P \equiv n[P']$ and $P' \models_T \mathcal{A}$
$P \models_T \mathcal{A} \mid \mathcal{B}$	iff $\exists P',P''.\ P \equiv P' \mid P''$ and $P' \models_T \mathcal{A}$ and $P'' \models_T \mathcal{B}$
$P \models_T \mathrm{H}x.\mathcal{A}$	iff $\exists n,P'.\ P \equiv (\nu n)P'$ and $n \notin na(\mathcal{A})$ and $P' \models_T \mathcal{A}\{x \leftarrow n\}$
$P \models_T \copyright\mathcal{N}$	iff $\exists n.\ n \models_N \mathcal{N}$ and $n \in fn(P)$
$P \models_T \mathbf{F}$	never
$P \models_T \mathcal{A} \wedge \mathcal{B}$	iff $P \models_T \mathcal{A}$ and $P \models_T \mathcal{B}$
$P \models_T \mathcal{A} \Rightarrow \mathcal{B}$	iff $P \models_T \mathcal{A}$ implies $P \models_T \mathcal{B}$
$P \models_T \mathcal{A}(\mathcal{M} \leftrightarrow \mathcal{M}')$	iff $\exists m,m'.\ m \models_N \mathcal{M}$ and $m' \models_N \mathcal{M}'$ and $P \bullet (m \leftrightarrow m') \models_T \mathcal{A}$

On High Types: $F \vDash_H \mathcal{T}$, for $\mathcal{T}$ closed, is defined by:

$F \vDash_H \mathbf{N}$	iff $F \vDash_N \mathcal{N}$ for some $\mathcal{N}$
$F \vDash_H \mathcal{A}$	iff $F \vDash_T \mathcal{A}$
$H \vDash_H \mathcal{T}\to\mathcal{G}$ $(\mathcal{T}\neq\mathbf{N})$	iff $H = \langle \rho, z, t\rangle$ and $\forall F,G.\ (F \vDash_H \mathcal{T} \wedge\ t \Downarrow_{\rho[z\leftarrow F]} G) \Rightarrow G \vDash_H \mathcal{G}$
$H \vDash_H \Pi x.\ \mathcal{G}$	iff $H = \langle \rho, z, t\rangle$ and $\forall n,G.\ t \Downarrow_{\rho[z\leftarrow n]} G\ \Rightarrow\ G \vDash_H \mathcal{G}\{x\leftarrow n\}$

(We will omit the subscripts on $\vDash$.) The constructs $\mathsf{H}x.\mathcal{A}$ and $\copyright\mathcal{N}$ are derived operators in [10], and are taken here as primitive, in the original spirit of [9]. In the definition of $\mathsf{H}x.\mathcal{A}$, the clause $P \equiv (\nu n)P'$ pulls a restriction (even a dummy one) from elsewhere in the data, via scope extrusion (Definition 2-5). The type $\mathsf{H}w.\copyright w$ is the type of non-redundant restrictions, with the quantifier $\mathsf{H}w$ revealing a restricted name n, and $\copyright w$ declaring that this n is used in the data. The meaning of formal transpositions relies on actual transpositions. At high types, a closure $\langle \rho, z, t\rangle$ satisfies a function type $\mathcal{T}\to\mathcal{G}$ if, on any input satisfying $\mathcal{T}$, every output satisfies $\mathcal{G}$; similarly for $\Pi x.\ \mathcal{G}$.

Proposition 4-2 (Tree Satisfaction Under Structural Congruence)

If $P \vDash \mathcal{A}$ and $P \equiv Q$ then $Q \vDash \mathcal{A}$.

Lemma 4-3 (Tree Satisfaction Under Actual Transposition)

If $n \vDash \mathcal{N}$ then $n\bullet(m\leftrightarrow m') \vDash \mathcal{N}\bullet(m\leftrightarrow m')$.
If $P \vDash \mathcal{A}$ then $P\bullet(m\leftrightarrow m') \vDash \mathcal{A}\bullet(m\leftrightarrow m')$.

This Lemma is not extended to high types because it would depend on the operational semantics of Section 5. Moreover, this extension is not needed, because transposition tends to be useful when reasoning about restriction, which is not defined on high types.

5 Operational Semantics

We give a *big step* operational semantics that is later used for a subject reduction result (Theorem 7-4). This style of semantics, namely a relation between a program and all its potential final results, is sufficient to clarify the intended behavior of our operations. It could be extended with error handling. Alternatively, a *small step* semantics could be given. In either case, one could go further and establish a type soundness theorem stating that well-typed programs (preserve types and) do not get stuck. All this is relatively routine, and we opt to give only the essential semantics.

The operational semantics is given by a relation $t \Downarrow_\rho F$ between terms t, stacks ρ, and values F, meaning that t can evaluate to F on stack ρ. An auxiliary relation, $\mathcal{N} \downarrow_\rho n$, deals with evaluation of name expressions. The semantics of run-time tests makes use of the satisfaction relation from Section 4. We use, $t \Downarrow_\rho P$ to indicate that t evaluates to a tree value. We use $t \Downarrow_\rho \equiv P$ as an abbreviation for $t \Downarrow_\rho Q$ and $Q \equiv P$, for some Q.

Definition 5-1 (Operational Semantics)

$$\frac{x \in dom(\rho) \quad \rho(x) \in \Lambda}{x \downarrow_\rho \rho(x)}\ (\text{NRed } x) \qquad \frac{}{n \downarrow_\rho n}\ (\text{NRed } n) \qquad \frac{\mathcal{N} \downarrow_\rho n \quad \mathcal{M} \downarrow_\rho m \quad \mathcal{M}' \downarrow_\rho m'}{\mathcal{N}\{\mathcal{M} \leftrightarrow \mathcal{M}'\} \downarrow_\rho n \bullet \{m \leftrightarrow m'\}}\ (\text{NRed} \leftrightarrow)$$

$$\frac{}{0 \Downarrow_\rho 0}\ (\text{Red } 0) \qquad \frac{\mathcal{N} \downarrow_\rho n \quad t \Downarrow_\rho P}{\mathcal{N}[t] \Downarrow_\rho n[P]}\ (\text{Red } \mathcal{N}[]) \qquad \frac{t \Downarrow_\rho P \quad u \Downarrow_\rho Q}{t \mid u \Downarrow_\rho P \mid Q}\ (\text{Red } \mid) \qquad \frac{n \notin na(t, \rho) \quad t \Downarrow_{\rho[x \leftarrow n]} P}{(\nu x)t \Downarrow_\rho (\nu n)P}\ (\text{Red } \nu)$$

$$\frac{t \Downarrow_\rho P \quad \mathcal{M} \downarrow_\rho m \quad \mathcal{M}' \downarrow_\rho m'}{t\{\mathcal{M} \leftrightarrow \mathcal{M}'\} \Downarrow_\rho P \bullet \{m \leftrightarrow m'\}}\ (\text{Red} \leftrightarrow) \qquad \frac{\mathcal{N} \downarrow_\rho n \quad t \Downarrow_\rho \equiv n[P] \quad P \vDash \rho(\mathcal{A}) \quad u \Downarrow_{\rho[y \leftarrow P]} F}{t \div (\mathcal{N}[y{:}\mathcal{A}]).u \Downarrow_\rho F}\ (\text{Red} \div \mathcal{N}[])$$

$$\frac{t \Downarrow_\rho \equiv P' \mid P'' \quad P' \vDash \rho(\mathcal{A}) \quad P'' \vDash \rho(\mathcal{B}) \quad x \neq y \quad u \Downarrow_{\rho[x \leftarrow P'][y \leftarrow P'']} F}{t \div (x{:}\mathcal{A} \mid y{:}\mathcal{B}).u \Downarrow_\rho F}\ (\text{Red} \div \mid) \qquad \frac{n \notin na(t, \mathcal{A}, u, \rho) \quad t \Downarrow_\rho \equiv (\nu n)P \quad P \vDash \rho[x \leftarrow n](\mathcal{A}) \quad x \neq y \quad u \Downarrow_{\rho[x \leftarrow n][y \leftarrow P]} Q}{t \div ((\nu x)y{:}\mathcal{A}).u \Downarrow_\rho (\nu n)Q}\ (\text{Red} \div \nu)$$

$$\frac{t \Downarrow_\rho P \quad P \vDash \rho(\mathcal{A}) \quad u \Downarrow_{\rho[x \leftarrow P]} F}{t?(x{:}\mathcal{A}).u,v \Downarrow_\rho F}\ (\text{Red } ?\vDash) \qquad \frac{t \Downarrow_\rho P \quad P \vDash \neg\rho(\mathcal{A}) \quad v \Downarrow_{\rho[x \leftarrow P]} F}{t?(x{:}\mathcal{A}).u,v \Downarrow_\rho F}\ (\text{Red } ?\nvDash)$$

$$\frac{x \in dom(\rho)}{x \Downarrow_\rho \rho(x)}\ (\text{Red } x) \qquad \frac{\mathcal{N} \downarrow_\rho n}{\mathcal{N} \Downarrow_\rho n}\ (\text{Red } \mathcal{N}) \qquad \frac{}{\lambda x{:}\mathcal{F}.t \Downarrow_\rho \langle \rho, x, t \rangle}\ (\text{Red } \lambda) \qquad \frac{t \Downarrow_\rho \langle \rho', x, t' \rangle \quad u \Downarrow_\rho G \quad t' \Downarrow_{\rho'[x \leftarrow G]} H}{t(u) \Downarrow_\rho H}\ (\text{Red App})$$

We now comment on some properties of the operational semantics.

The operations (Red ÷ -) and (Red ? -) may, at run time, execute satisfaction tests on dependent types that have been instantiated during execution. E.g., note the role of x in $\lambda x{:}\mathbf{N}.\ t?(y{:}x[\mathbf{0}]).u,v$, which, by (Red ?⊨), requires a satisfaction test of the form $P \vDash \neg\rho(x[\mathbf{0}])$, where $t \Downarrow_\rho P$ and where the dependent type variable x must be bound in ρ.

In these rules, $\rho(\mathcal{A})$ replaces every free variable $x \in dom(\rho)$ in $\mathcal{A}$ with $\rho(x)$. The rules are applicable only if $\rho(\mathcal{A})$ is a well-formed type (otherwise, the rules are *stuck*): the type rules of Section 7 guarantee this well-formedness condition.

The matching reductions are nondeterministic. Hence we have a nondeterministic big step semantics, which gives no information on the existence of divergent reduction paths.

Reduction is not closed up to ≡ (0 does not reduce to 0 | 0), nor up to $\equiv_\alpha$ (see (Red ν) and (Red ÷ν), which exclude some of the bound names that can be returned). But this is a matter of choice that has no effect on our results.

The following lemma is crucial in the subject reduction cases for (Red ν) (Theorem 7-4). Only this transposition lemma is needed there, not a harder substitution lemma.

Lemma 5-2 (Reduction Under Transposition)

If $\mathcal{M} \downarrow_\rho m$ then $\mathcal{M}\bullet(n\leftrightarrow n') \downarrow_{\rho\bullet(n\leftrightarrow n')} m\bullet(n\leftrightarrow n')$.
If $t \Downarrow_\rho F$ then $t\bullet(n\leftrightarrow n') \Downarrow_{\rho\bullet(n\leftrightarrow n')} F\bullet(n\leftrightarrow n')$.

6 Transposition Equivalence and Apartness

We define a type equivalence relation on name expressions and tree types, which in particular allows any type transposition to be eliminated or pushed down to the name expressions that appear in the type. The main aim of this section is to establish the soundness of such an equivalence relation, which is inspired by [25,11]. A crucial equivalence rule, (EqN ↔ Apart) (Definition 6-2), requires the notion of *apartness* of name expressions, meaning that the names that those expressions denote are distinct. (*C.f.* examples (5) and (6) in Introduction.) Apartness of name expressions depends on apartness of variables and names; we keep track of such relationships via a *freshness signature*.

Definition 6-1 (Freshness Signature)

A *freshness signature* ϕ is an ordered list of names followed by an ordered list of distinct variables annotated with a quantifier Q that is either $\forall$ or H. (For example: $n,m,m,p,\forall x,\mathsf{H}y,\mathsf{H}z,\forall w$.)

Notation: $dom(\phi)$ is the set of variables in ϕ; $na(\phi)$ is the set of names in ϕ; $\phi(x)$ is the symbol $\forall$ or H associated to x in ϕ. We write $x <_\phi y$ if $x \neq y$ and x precedes y in ϕ. We write $\phi \supseteq \mathcal{N}$ (ϕ covers $\mathcal{N}$) when $fv(\mathcal{N}) \subseteq dom(\phi)$ and $fn(\mathcal{N}) \subseteq na(\phi)$; similarly for $\phi \supseteq \mathcal{A}$ and $\phi \supseteq \mathcal{T}$. We write $S \# S'$ for disjointness of sets of names and variables.

Next we define three equivalence relations between name expressions, $\sim_N$, tree types, $\sim_T$, and high types, $\sim_H$ (often omitting the subscripts), and an apartness relation on name expressions, #. These relations are all indexed by a freshness signature that is understood to cover the free variables and names occurring in the expressions involved. Whether an equivalence or apartness holds, depends crucially on such freshness signature; consider the following examples:

$n \#_{n,m} m$	by (Apart Names)
$n \#_{n,\mathsf{H}x} x$	by (Apart Name Var)
$x \#_{\forall x,\mathsf{H}y} y$	by (Apart Vars)
$y \#_{n,\forall x,\mathsf{H}y} x,\ y \#_{n,\forall x,\mathsf{H}y} n \ \Rightarrow\ y(x\leftrightarrow n) \sim_{n,\forall x,\mathsf{H}y} y$	by (EqN ↔ Apart)
$y \#_{n,\forall x,\mathsf{H}y} x \ \Rightarrow\ y(y\leftrightarrow n) \#_{n,\forall x,\mathsf{H}y} x(y\leftrightarrow n)$	by (Apart Congr)
$n \sim_{n,\forall x,\mathsf{H}y} y(y\leftrightarrow n),\ y(y\leftrightarrow n) \#_{n,\forall x,\mathsf{H}y} x(y\leftrightarrow n)$ $\Rightarrow\ n \#_{n,\forall x,\mathsf{H}y} x(y\leftrightarrow n)$	by (Apart Equiv)

A notion of apartness of names from types is not necessary, since transpositions on types can be distributed down to transpositions on name expressions.

Definition 6-2 (Equivalence and Apartness)

Name expression equivalence, $\mathcal{N} \sim_{N\phi} \mathcal{M}$, (abbrev. $\mathcal{N} \sim_\phi \mathcal{M}$) and apartness, $\mathcal{N} \#_\phi \mathcal{M}$, are the least relations on name expressions such that $\phi \supseteq \mathcal{N},\mathcal{M}$, and:

$\mathcal{N}\#_\phi\mathcal{M} \;\Rightarrow\; \mathcal{M}\#_\phi\mathcal{N}$ (Apart Symm)
$n \neq m \;\Rightarrow\; n \#_\phi m$ (Apart Names)
$\phi(x)=\mathsf{H} \;\Rightarrow\; n \#_\phi x$ (Apart Name Var)
$x <_\phi y$ and $\phi(y)=\mathsf{H} \;\Rightarrow\; x \#_\phi y$ (Apart Vars)
$\mathcal{N}\#_\phi\mathcal{M} \wedge \mathcal{P} \sim_\phi \mathcal{P}' \wedge \mathcal{Q} \sim_\phi \mathcal{Q}' \;\Rightarrow\; \mathcal{N}\{\mathcal{P}\leftrightarrow\mathcal{Q}\} \#_\phi \mathcal{M}\{\mathcal{P}'\leftrightarrow\mathcal{Q}'\}$ (Apart Congr)
$\mathcal{N} \sim_\phi \mathcal{N}'$ and $\mathcal{N}' \#_\phi \mathcal{M}'$ and $\mathcal{M}' \sim_\phi \mathcal{M} \;\Rightarrow\; \mathcal{N}\#_\phi\mathcal{M}$ (Apart Equiv)

$\mathcal{N} \sim_\phi \mathcal{N}$ (EqN Refl)
$\mathcal{N} \sim_\phi \mathcal{M} \;\Rightarrow\; \mathcal{M} \sim_\phi \mathcal{N}$ (EqN Symm)
$\mathcal{N} \sim_\phi \mathcal{M}$ and $\mathcal{M} \sim_\phi \mathcal{P} \;\Rightarrow\; \mathcal{N} \sim_\phi \mathcal{P}$ (EqN Trans)
$\mathcal{N} \sim_\phi \mathcal{N}'$, $\mathcal{M} \sim_\phi \mathcal{M}'$, $\mathcal{P} \sim_\phi \mathcal{P}' \;\Rightarrow\; \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{P}\} \sim_\phi \mathcal{N}'\{\mathcal{M}'\leftrightarrow\mathcal{P}'\}$ (EqN $\leftrightarrow$ Congr)

$\mathcal{N}\{\mathcal{N}\leftrightarrow\mathcal{M}\} \sim_\phi \mathcal{M}$ (EqN $\leftrightarrow$ App)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}\} \sim_\phi \mathcal{N}$ (EqN $\leftrightarrow$ Id)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}\{\mathcal{M}'\leftrightarrow\mathcal{M}\}$ (EqN $\leftrightarrow$ Symm)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}$ (EqN $\leftrightarrow$ Inv)
$\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\{\mathcal{P}\leftrightarrow\mathcal{P}'\} \sim_\phi \mathcal{N}\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\{\mathcal{M}\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\leftrightarrow\mathcal{M}'\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\}$ (EqN $\leftrightarrow\leftrightarrow$)
$\mathcal{N}\#_\phi\mathcal{M}$ and $\mathcal{N}\#_\phi\mathcal{M}' \;\Rightarrow\; \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}$ (EqN $\leftrightarrow$ Apart)

Tree type equivalence, $\mathcal{A} \sim_{T\phi} \mathcal{B}$, (abbrev. $\mathcal{A} \sim_\phi \mathcal{B}$),
are the least relations on tree types such that $\phi \supseteq \mathcal{A},\mathcal{B}$, and:

$\mathcal{A} \sim_\phi \mathcal{A}$ (EqT Refl)
$\mathcal{A} \sim_\phi \mathcal{B} \;\Rightarrow\; \mathcal{B} \sim_\phi \mathcal{A}$ (EqT Symm)
$\mathcal{A} \sim_\phi \mathcal{B}$ and $\mathcal{B} \sim_\phi \mathcal{C} \;\Rightarrow\; \mathcal{A} \sim_\phi \mathcal{C}$ (EqT Trans)
$\mathcal{N} \sim_{N\phi} \mathcal{N}'$ and $\mathcal{A} \sim_\phi \mathcal{A}' \;\Rightarrow\; \mathcal{N}'[\mathcal{A}] \sim_\phi \mathcal{N}'[\mathcal{A}']$ (EqT $\mathcal{N}[]$ Congr)
$\mathcal{A} \sim_\phi \mathcal{A}'$ and $\mathcal{B} \sim_\phi \mathcal{B}' \;\Rightarrow\; \mathcal{A} \mid \mathcal{B} \sim_\phi \mathcal{A}' \mid \mathcal{B}'$ (EqT $\mid$ Congr)
$\mathcal{A} \sim_\phi \mathcal{A}'$ and $\mathcal{B} \sim_\phi \mathcal{B}' \;\Rightarrow\; \mathcal{A} \wedge \mathcal{B} \sim_\phi \mathcal{A}' \wedge \mathcal{B}'$ (EqT $\wedge$ Congr)
$\mathcal{A} \sim_\phi \mathcal{A}'$ and $\mathcal{B} \sim_\phi \mathcal{B}' \;\Rightarrow\; \mathcal{A} \Rightarrow \mathcal{B} \sim_\phi \mathcal{A}' \Rightarrow \mathcal{B}'$ (EqT $\Rightarrow$ Congr)
$\mathcal{A} \sim_{(\phi,\mathsf{H}x)} \mathcal{A}' \;\Rightarrow\; \mathsf{H}x.\,\mathcal{A} \sim_\phi \mathsf{H}x.\,\mathcal{A}'$ (EqT H Congr)
$\mathcal{N} \sim_{N\phi} \mathcal{N}' \;\Rightarrow\; ©\mathcal{N} \sim_\phi ©\mathcal{N}'$ (EqT © Congr)

$\mathbf{0}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathbf{0}$ (EqT $\mathbf{0}\leftrightarrow$)
$(\mathcal{N}[\mathcal{A}])\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}[\mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}]$ (EqT $\mathcal{N}[]\leftrightarrow$)
$(\mathcal{A} \mid \mathcal{B})\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \mid \mathcal{B}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$ (EqT $\mid\leftrightarrow$)
$\mathbf{F}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathbf{F}$ (EqT $\mathbf{F}\leftrightarrow$)
$(\mathcal{A} \wedge \mathcal{B})\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \wedge \mathcal{B}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$ (EqT $\wedge\leftrightarrow$)
$(\mathcal{A} \Rightarrow \mathcal{B})\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \Rightarrow \mathcal{B}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}$ (EqT $\Rightarrow\leftrightarrow$)
$(\mathsf{H}x.\mathcal{A})\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi$ (EqT H $\leftrightarrow$)
$\quad \mathsf{H}x.(\mathcal{A}\{x\leftarrow x\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\}\{\mathcal{M}\leftrightarrow\mathcal{M}'\})$ with $x \notin fv(\mathcal{M},\mathcal{M}')$
$(©\mathcal{N})\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi ©(\mathcal{N}\{\mathcal{M}\leftrightarrow\mathcal{M}'\})$ (EqT ©$\leftrightarrow$)
$\mathcal{A}\{\mathcal{P}\leftrightarrow\mathcal{P}'\}\{\mathcal{M}\leftrightarrow\mathcal{M}'\} \sim_\phi \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\{\mathcal{P}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\leftrightarrow\mathcal{P}'\{\mathcal{M}\leftrightarrow\mathcal{M}'\}\}$ (EqT $\leftrightarrow\leftrightarrow$)

$\mathsf{H}x.\,\mathcal{A} \sim_\phi \mathsf{H}y.\,\mathcal{A}\{x\leftarrow y\}$ with $y \notin fv(\mathcal{A})$ (EqT H-α)

High type equivalence, $\mathcal{F} \sim_{H\phi} \mathcal{G}$, (abbrev. $\mathcal{F} \sim_\phi \mathcal{G}$),
are the least relations on high types such that $\phi \supseteq \mathcal{F},\mathcal{G}$, and:

$\mathcal{F} \sim_\phi \mathcal{F}$ (EqH Refl)
$\mathcal{F} \sim_\phi \mathcal{G} \;\Rightarrow\; \mathcal{G} \sim_\phi \mathcal{F}$ (EqH Symm)
$\mathcal{F} \sim_\phi \mathcal{G}$ and $\mathcal{G} \sim_\phi \mathcal{H} \;\Rightarrow\; \mathcal{F} \sim_\phi \mathcal{H}$ (EqH Trans)

$\mathcal{A} \sim_{\mathrm{T}\phi} \mathcal{B} \;\Rightarrow\; \mathcal{A} \sim_{\mathrm{H}\phi} \mathcal{B}$ (EqH $\mathcal{A}$ Congr)
$\mathcal{F} \sim_\phi \mathcal{F}' \wedge \mathcal{G} \sim_\phi \mathcal{G}' \;\Rightarrow\; \mathcal{F}\to\mathcal{G} \sim_\phi \mathcal{F}'\to\mathcal{G}'$ (EqH $\to$ Congr)
$\mathcal{F} \sim_{(\phi,\forall x)} \mathcal{G} \;\Rightarrow\; \Pi x.\, \mathcal{F} \sim_\phi \Pi x.\, \mathcal{G}$ (EqH Π Congr)
$\Pi x.\, \mathcal{G} \sim_\phi \Pi y.\, \mathcal{G}\{x\leftarrow y\}$ with $y \notin fv(\mathcal{G})$ (EqH Π-α)

For (EqT H $\leftrightarrow$) see [11], where it is shown that a similar property holds for quantifiers.

There are interesting type transpositions that do not simplify away, such as this one: $\Pi x.\; x(n\leftrightarrow m)[\mathbf{T}]$, which is a latent substitution. However, for $\mathsf{H}x.\; x(n\leftrightarrow m)[\mathbf{T}]$, which is the same as $\mathsf{H}x.\; (x[\mathbf{T}](n\leftrightarrow m))$, we have an equivalence with $(\mathsf{H}x.\; x[\mathbf{T}])(n\leftrightarrow m)$. This is really saying that swapping an H-bound name via two concrete names cound not have any effect, because the bound name must be fresh with respect to the concrete names. This swapping of H and $\leftrightarrow$ is the most interesting rule about type transpositions, and only works if the transposition does not mention the bound variable. (In $\mathsf{H}y.\mathbf{T}(n\leftrightarrow y)$ we have such an entaglement with bound variables, but that type can be simplified through another path: $\mathbf{T}(n\leftrightarrow y) \sim \mathbf{T}$ and hence $\mathsf{H}y.\mathbf{T}(n\leftrightarrow y) \sim \mathsf{H}y.\mathbf{T}$.)

The full story about H and $\leftrightarrow$ is that $(\mathsf{H}x.\; \mathcal{A})(\mathcal{M}\leftrightarrow\mathcal{M}') \sim \mathsf{H}x.\; \mathcal{A}(\mathcal{M}\leftrightarrow\mathcal{M}')$ for x not in $\mathcal{M}\leftrightarrow\mathcal{M}'$, "except that the transpostion is never applied to the occurrences of x". This is expressed by two rules, which combined give rule ((EqT H $\leftrightarrow$)); one pushing the transposition inside the quantifier:

$(\mathsf{H}x.\; \mathcal{A})(\mathcal{M}\leftrightarrow\mathcal{M}') \sim \mathsf{H}x.\; \mathcal{A}(\mathcal{M}\leftrightarrow\mathcal{M}')$ where $x \notin fv(\mathcal{M}\leftrightarrow\mathcal{M}')$

and one saying that a transposition has no effect on bound variables:

$\mathsf{H}x.\; \mathcal{A} \sim \mathsf{H}x.\; \mathcal{A}\{x\leftarrow x(\mathcal{M}\leftrightarrow\mathcal{M}')\}$ where $x \notin fv(\mathcal{M}\leftrightarrow\mathcal{M}')$

Such a rule is sound for all quantifiers. If $\exists x.\; \mathcal{A}\{x\}$, then also $\exists x.\; \mathcal{A}\{x(m\leftrightarrow n)\}$: if in the first case $x{=}n$ or $x{=}m$, then in the second case take $x{=}m$ or $x{=}n$. Similarly, if $\forall x.\; \mathcal{A}\{x\}$, then also $\forall x.\; \mathcal{A}\{x(m\leftrightarrow n)\}$: if something holds for any x, then it also holds for any $x(m\leftrightarrow n)$, which is the same set. Finally, if $\mathsf{H}x.\; \mathcal{A}$, then also $\mathsf{H}x.\; \mathcal{A}\{x(m\leftrightarrow n)\}$: if in the first case x denotes a fresh name, even if we happen to pick n or m for it, then in the second case we can pick a different fresh name, and $x(m\leftrightarrow n)$ has no effect on it.

Remark: Transpositions seem to be a problem for dependent types:

$(\Pi x.\; x(m\leftrightarrow n)[0]) = (\Pi x.\; x[0]\{x\leftarrow x(m\leftrightarrow n)\})$ is not the same as $(\Pi x.\; x[0])$,

so the cancellation rule does not apply. We do not allow transpositions on high types.

Definition 6-3 (Valuation)

(1) If $\varepsilon{:}\mathit{Var}\to\Lambda$ is a finite map, then we say that ε is a *valuation*.
(2) We indicate by $\varepsilon(\mathcal{N})$, $\varepsilon(\mathcal{A})$ the homomorphic extensions of ε to name expressions and tree types, with the understanding that in such extension $\varepsilon(x) = x$ for $x \notin dom(\varepsilon)$.
(3) If $fv(\mathcal{N}) \subseteq dom(\varepsilon)$ then we say that ε is a *ground valuation* for $\mathcal{N}$, and we write ε *grounds* $\mathcal{N}$; similarly for $\mathcal{A}$ and $\mathcal{F}$.

(4) We indicate by $\varepsilon(\phi)$ the freshness signature obtained by:
$\varepsilon(\emptyset) = \emptyset$
$\varepsilon(n,\phi) = n,\varepsilon(\phi)$
$\varepsilon(\phi,\mathsf{Q}x) = \varepsilon(x),\varepsilon(\phi)$ if $x \in dom(\varepsilon)$, and $\varepsilon(\phi),\mathsf{Q}x$ otherwise
N.B.: if $\phi \supseteq \mathcal{N}$ then $\varepsilon(\phi) \supseteq \varepsilon(\mathcal{N})$.

We say that a valuation ε satisfies a freshness signature ϕ if it respects the freshness constraints of ϕ, in the following sense:

Definition 6-4 (Freshness Signature Satisfaction)

$\varepsilon \vDash \phi$ iff $dom(\varepsilon) \subseteq dom(\phi)$
and $\forall x \in dom(\varepsilon).\ \phi(x)=\mathsf{H} \Rightarrow \varepsilon(x) \notin na(\phi)$
and $\forall y \in dom(\varepsilon).\ \forall x \in dom(\phi).\ (x <_\phi y \wedge \phi(y)=\mathsf{H}) \Rightarrow (x \in dom(\varepsilon) \wedge \varepsilon(x) \neq \varepsilon(y))$

In $\varepsilon \vDash \phi$ we do not require $dom(\phi) \subseteq dom(\varepsilon)$, to allow for partial valuations. But we require any partial valuation that instantiates an H variable to instantiate all the variables to the left of it (with distinct names).

Lemma 6-5 (Composition of Signature Satisfaction)

$\varepsilon_1 \vDash \phi \wedge \varepsilon_2 \vDash \varepsilon_1(\phi)$ implies $dom(\varepsilon_1) \# dom(\varepsilon_2) \wedge \varepsilon_2 \circ \varepsilon_1 \vDash \phi$

Note that if the third condition of Definition 6-4 is changed to the apparently more natural:
$\forall x,y \in dom(\varepsilon).\ (x <_\phi y \wedge \phi(y)=\mathsf{H}) \Rightarrow \varepsilon(x) \neq \varepsilon(y)$
(allowing uninstantiated variables to the left of H variables) then Lemma 6-5 fails. Consider $\phi = \forall x,\mathsf{H}y$, $\varepsilon_1 = y \mapsto n$, $\varepsilon_2 = x \mapsto n$; then $\varepsilon_1 \vDash \phi$, $\varepsilon_2 \vDash \varepsilon_1(\phi) = n,\forall x$, but $\varepsilon_2 \circ \varepsilon_1 = y \mapsto n,\ x \mapsto n \nvDash \phi$.

Equivalence and apartness are preserved by (partial) valuation:

Proposition 6-6 (Instances of Equivalance and Apartness)

(1) If $\mathcal{N} \#_\phi \mathcal{M}$ then $\forall \varepsilon \vDash \phi.\ \varepsilon(\mathcal{N}) \#_{\varepsilon(\phi)} \varepsilon(\mathcal{M})$.
(2) If $\mathcal{N} \sim_\phi \mathcal{M}$ then $\forall \varepsilon \vDash \phi.\ \varepsilon(\mathcal{N}) \sim_{\varepsilon(\phi)} \varepsilon(\mathcal{M})$.
(3) If $\mathcal{A} \sim_\phi \mathcal{B}$ then $\forall \varepsilon \vDash \phi.\ \varepsilon(\mathcal{A}) \sim_{\varepsilon(\phi)} \varepsilon(\mathcal{B})$.
(4) If $\mathcal{T} \sim_\phi \mathcal{G}$ then $\forall \varepsilon \vDash \phi.\ \varepsilon(\mathcal{T}) \sim_{\varepsilon(\phi)} \varepsilon(\mathcal{G})$.

We now study the relationship between satisfaction and transposition equivalence: for example, situations like $P \vDash \mathcal{A}$ and $\mathcal{A} \sim_\phi \mathcal{B}$. Fortunately, here $\mathcal{A}$ must be closed, and we require $\mathcal{B}$ to be closed too, so ϕ does not play a significant role.

Proposition 6-7 (Soundness of Closed Type Equivalence)

(1) If $P \vDash \mathcal{A}$ and $\mathcal{A} \sim_\phi \mathcal{B}$ ($\mathcal{B}$ closed) then $P \vDash \mathcal{B}$.
(2) If $F \vDash \mathcal{T}$ and $\mathcal{T} \sim_\phi \mathcal{G}$ ($\mathcal{G}$ closed) then $F \vDash \mathcal{G}$.

Proposition 6-8 (Soundness of Equivalence and Apartness)

If $\mathcal{N} \#_\phi \mathcal{M}$ then $\forall \varepsilon \vDash \phi.\ (\varepsilon \textit{ grounds } \mathcal{N},\mathcal{M}) \Rightarrow \varepsilon(\mathcal{N}) \neq \varepsilon(\mathcal{M})$.

If $\mathcal{N} \sim_\phi \mathcal{M}$ then $\forall\varepsilon{\vDash}\phi.\ (\varepsilon\ \textit{grounds}\ \mathcal{N},\mathcal{M}) \Rightarrow \varepsilon(\mathcal{N}) = \varepsilon(\mathcal{M})$.
If $\mathcal{A} \sim_\phi \mathcal{B}$ then $\forall\varepsilon{\vDash}\phi.\ (\varepsilon\ \textit{grounds}\ \mathcal{A},\mathcal{B}) \Rightarrow \forall P.\ P \vDash \varepsilon(\mathcal{A}) \Rightarrow P \vDash \varepsilon(\mathcal{B})$.
If $\mathcal{F} \sim_\phi \mathcal{G}$ then $\forall\varepsilon{\vDash}\phi.\ (\varepsilon\ \textit{grounds}\ \mathcal{F},\mathcal{G}) \Rightarrow \forall F.\ F \vDash \varepsilon(\mathcal{F}) \Rightarrow F \vDash \varepsilon(\mathcal{G})$.

7 Type System

We now present a type system that is sound for the operational semantics of Section 5. Subtyping includes the transposition equivalence of Section 6 (see rule (Sub Equiv)), and an unspecified collection of *ValidEntailments* that may capture aspects of logical implication. Apart from the flexibility given by subtyping through rule (Subsumption), the type rules for terms are remarkably straightforward and syntax-driven.

The type system uses environments E that have a slightly unusual structure. They are ordered lists of either names (covering all the names occurring in expressions; see rule (NExpr n)), or variables (covering all the free variables of expressions; see rule (Term x)). Variables have associated type and freshness information of the form $x{:}\mathcal{T}$ if $\mathcal{T}{\neq}\mathbf{N}$, and $\mathsf{Q}x{:}\mathcal{T}$ (either $\forall x{:}\mathcal{T}$ or $\mathsf{H}x{:}\mathcal{T}$) if $\mathcal{T}{=}\mathbf{N}$. We write $dom(E)$ and $na(E)$ for the set of variables and the set of names defined by E. We group names to the left and variables to the right; hence we write n,E for the extension of E with a name n, and we write $E,x{:}\mathcal{T}$ and $E,\mathsf{Q}x{:}\mathbf{N}$ for the extension of E with a new variable association (provided that $x{\notin}dom(E)$), where $\mathcal{T}$ may depend on $dom(E)$. We write $E(x)$ for the (open) type associated to $x{\in}dom(E)$ in E. A freshness signature (Definition 6-1) can be extracted as follows:

Definition 7-1 (Freshness Signature of an Environment)

$$\begin{array}{llll} fs(\emptyset) \triangleq \emptyset & & fs(E, \mathsf{Q}x{:}\mathbf{N}) \triangleq fs(E), \mathsf{Q}x \\ fs(n, E) \triangleq n, fs(E) & & fs(E, x{:}\mathcal{T}) \triangleq fs(E) \end{array}$$

Through $fs(E)$, in typing rule (Sub Equiv), and through rule (Env n), typing environments are connected to the freshness signatures used in transposition equivalence.

Definition 7-2 (Type Rules)

Environments. Rules for $E \vdash \diamond$ (that is, E is well-formed).

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{E \vdash \diamond}{n, E \vdash \diamond}\ (\text{Env } n) \qquad \frac{E \vdash \diamond \quad \mathsf{Q}{\in}\{\forall,\mathsf{H}\} \quad x{\notin}dom(E)}{E, \mathsf{Q}x{:}\mathbf{N} \vdash \diamond}\ (\text{Env } x{\in}\mathbf{N}) \qquad \frac{E \vdash \mathcal{T} \quad \mathcal{T}{\neq}\mathbf{N} \quad x{\notin}dom(E)}{E, x{:}\mathcal{T} \vdash \diamond}\ (\text{Env } x{\notin}\mathbf{N})$$

Names. Rules for $E \vdash_{\mathrm{N}} \mathcal{N}$ (that is, $\mathcal{N}$ is a name expression in E).

$$\frac{E \vdash \diamond \quad n{\in}na(E)}{E \vdash_{\mathrm{N}} n}\ (\text{NExpr } n) \qquad \frac{E \vdash \diamond \quad E(x){=}\mathbf{N}}{E \vdash_{\mathrm{N}} x}\ (\text{NExpr } x) \qquad \frac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash_{\mathrm{N}} \mathcal{N}(\mathcal{M}{\leftrightarrow}\mathcal{M}')}\ (\text{NExpr } {\leftrightarrow})$$

Tree Types. Rules for $E \vdash_{\mathrm{T}} \mathcal{A}$ (that is, $\mathcal{A}$ is a tree type in E).

(Type **0**) $\dfrac{E \vdash \diamond}{E \vdash_{\mathrm{T}} \mathbf{0}}$ (Type $\mathcal{N}[]$) $\dfrac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{T}} \mathcal{A}}{E \vdash_{\mathrm{T}} \mathcal{N}[\mathcal{A}]}$ (Type $\mid$) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{T}} \mathcal{B}}{E \vdash_{\mathrm{T}} \mathcal{A} \mid \mathcal{B}}$

(Type **F**) $\dfrac{E \vdash \diamond}{E \vdash_{\mathrm{T}} \mathbf{F}}$ (Type $\wedge$) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{T}} \mathcal{B}}{E \vdash_{\mathrm{T}} \mathcal{A} \wedge \mathcal{B}}$ (Type $\Rightarrow$) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{T}} \mathcal{B}}{E \vdash_{\mathrm{T}} \mathcal{A} \Rightarrow \mathcal{B}}$

(Type H) $\dfrac{E, \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} \mathcal{A}}{E \vdash_{\mathrm{T}} \mathsf{H}x.\mathcal{A}}$ (Type ©) $\dfrac{E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash_{\mathrm{T}} ©\mathcal{N}}$ (Type $\leftrightarrow$) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash_{\mathrm{T}} \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}}$

Types. Rules for $E \vdash \mathcal{T}$ (that is, $\mathcal{T}$ is a type in E).

(Type Tree) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A}}{E \vdash \mathcal{A}}$ (Type **N**) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{N}}$ (Type $\rightarrow$) $\dfrac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T} \neq \mathbf{N}}{E \vdash \mathcal{T} \rightarrow \mathcal{G}}$ (Type Π) $\dfrac{E, \forall x{:}\mathbf{N} \vdash \mathcal{G}}{E \vdash \Pi x.\, \mathcal{G}}$

Subtyping. Rules for $E \vdash \mathcal{T} <: \mathcal{G}$ (that is, $\mathcal{T}$ is a subtype of $\mathcal{G}$ in E).

(Sub Tree) $\dfrac{E \vdash_{\mathrm{T}} \mathcal{A} \quad E \vdash_{\mathrm{T}} \mathcal{B} \quad \langle \mathcal{A}, fs(E), \mathcal{B} \rangle \in \mathit{ValidEntailments}}{E \vdash \mathcal{A} <: \mathcal{B}}$ (Sub Equiv) $\dfrac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T} \sim_{fs(E)} \mathcal{G}}{E \vdash \mathcal{T} <: \mathcal{G}}$

(Sub **N**) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{N} <: \mathbf{N}}$ (Sub $\rightarrow$) $\dfrac{E \vdash \mathcal{T}' <: \mathcal{T} \quad E \vdash \mathcal{G} <: \mathcal{G}' \quad \mathcal{T}, \mathcal{T}' \neq \mathbf{N}}{E \vdash \mathcal{T} \rightarrow \mathcal{G} <: \mathcal{T}' \rightarrow \mathcal{G}'}$ (Sub Π) $\dfrac{E, \forall x{:}\mathbf{N} \vdash \mathcal{G} <: \mathcal{G}'}{E \vdash \Pi x.\, \mathcal{G} <: \Pi x.\, \mathcal{G}'}$

Terms. Rules for $E \vdash t : \mathcal{T}$ (t has type $\mathcal{T}$ in E, with $E \vdash_{\mathrm{T}} t : \mathcal{A} \triangleq E \vdash_{\mathrm{T}} \mathcal{A} \wedge E \vdash t : \mathcal{A}$).

(Term 0) $\dfrac{E \vdash \diamond}{E \vdash 0 : \mathbf{0}}$ (Term $\mathcal{N}[]$) $\dfrac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash \mathcal{N}[t] : \mathcal{N}[\mathcal{A}]}$ (Term $\mid$) $\dfrac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{T}} u : \mathcal{B}}{E \vdash t \mid u : \mathcal{A} \mid \mathcal{B}}$

(Term ν) $\dfrac{E, \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash (\nu x)t : \mathsf{H}x.\mathcal{A}}$ (Term $\leftrightarrow$) $\dfrac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash t\{\mathcal{M}\leftrightarrow\mathcal{M}'\} : \mathcal{A}\{\mathcal{M}\leftrightarrow\mathcal{M}'\}}$

(Term $\div\mathcal{N}[]$) $\dfrac{E \vdash_{\mathrm{T}} t : \mathcal{N}[\mathcal{A}] \quad E, y{:}\mathcal{A} \vdash u : \mathcal{T}}{E \vdash t \div (\mathcal{N}[y{:}\mathcal{A}]).u : \mathcal{T}}$ (Term $\div\mid$) $\dfrac{E \vdash_{\mathrm{T}} t : \mathcal{A} \mid \mathcal{B} \quad E, x{:}\mathcal{A}, y{:}\mathcal{B} \vdash u : \mathcal{T}}{E \vdash t \div (x{:}\mathcal{A} \mid y{:}\mathcal{B}).u : \mathcal{T}}$

(Term $\div\nu$) $\dfrac{E \vdash_{\mathrm{T}} t : \mathsf{H}x.\mathcal{A} \quad E, \mathsf{H}x{:}\mathbf{N}, y{:}\mathcal{A} \vdash_{\mathrm{T}} u : \mathcal{B}}{E \vdash t \div ((\nu x)y{:}\mathcal{A}).u : \mathsf{H}x.\mathcal{B}}$ (Term ?) $\dfrac{E \vdash_{\mathrm{T}} t : \mathcal{B} \quad E, x{:}\mathcal{A} \vdash u : \mathcal{T} \quad E, x{:}\neg\mathcal{A} \vdash v : \mathcal{T}}{E \vdash t?(x{:}\mathcal{A}).u, v : \mathcal{T}}$

(Term x) $\dfrac{E \vdash \diamond \quad x \in dom(E)}{E \vdash x : E(x)}$ (Term $\mathcal{N}$) $\dfrac{E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash \mathcal{N} : \mathbf{N}}$ (Term λ) $\dfrac{E, x{:}\mathcal{T} \vdash t : \mathcal{G} \quad \mathcal{T} \neq \mathbf{N}}{E \vdash \lambda x{:}\mathcal{T}.t : \mathcal{T} \rightarrow \mathcal{G}}$ (Term App) $\dfrac{E \vdash t : \mathcal{T} \rightarrow \mathcal{G} \quad E \vdash u : \mathcal{T}}{E \vdash t(u) : \mathcal{G}}$

$$\frac{E,\ \forall x{:}\mathbf{N} \vdash t : \mathcal{G}}{E \vdash \lambda x{:}\mathbf{N}.t : \Pi x.\ \mathcal{G}}\ (\text{Term Dep}\lambda) \qquad \frac{E \vdash t : \Pi x.\ \mathcal{G} \quad E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash t(\mathcal{N}) : \mathcal{G}\{x \leftarrow \mathcal{N}\}}\ (\text{Term DepApp}) \qquad \frac{E \vdash t : \mathcal{F} \quad E \vdash \mathcal{F} <: \mathcal{G}}{E \vdash t : \mathcal{G}}\ (\text{Subsumption})$$

Here are some comments on these type rules:

As we already mentioned, the type system includes dependent types, with binding operators $\mathsf{H}x.\mathcal{A}$ (the type of hiding in trees) and $\Pi x.\mathcal{F}$ (the type of those functions $\lambda x{:}\mathbf{N}.t$ such that the type $\mathcal{F}$ of t may depend on the input variable x).

The subtyping relation is parameterized by a set *ValidEntailments*, assumed to consist of triples $\langle \mathcal{A}, \phi, \mathcal{B} \rangle$ that are sound ($\forall \varepsilon \vDash \phi.\ (\varepsilon$ *grounds* $\mathcal{A},\mathcal{B}) \Rightarrow \forall P.\ P \vDash \varepsilon(\mathcal{A}) \Rightarrow P \vDash \varepsilon(\mathcal{B})$).

(Sub Tree) is the base case for tree subtyping, using *ValidEntailments*. (Sub Equiv) is the base case for subtyping of high types, using transposition equivalence from Section 6.

In (Term ?), we inspect the run time type of t. Hence t can have any static type $\mathcal{B}$, but we require it to be well-typed, hence the assumption $E \vdash_{\mathrm{T}} t : \mathcal{B}$.

The use of $x{:}\neg\mathcal{A}$ in (Term ?) means $x{:}\mathcal{A} \Rightarrow \mathbf{F}$, but this assumption is not very useful without a rich theory of subtyping: see discussion in Section 1.5. On the other hand, there are no significant problems in executing run-time type tests such as $P \vDash \neg\mathcal{A}$ (see Definition 4-1), e.g., resulting from $t?(x{:}\neg\mathcal{A}).u,v$. A more informative typing of x for the third assumption of this rule is $x{:}\mathcal{B} \wedge \neg\mathcal{A}$, but we lack a compelling use for it.

In (Term $\div\mathcal{N}$[]) (and (Term $\div|$), (Term ?)) we do not need extra assumptions $E \vdash \mathcal{F}$ to avoid the escape of y (and x,y, and x) into $\mathcal{F}$, because these are not variables of type **N**, and $\mathcal{F}$ cannot depend on them. We do not need the extra assumption in (Term $\div\nu$) for x because there we rebind the result type.

In (TermDepApp) we require the argument $\mathcal{N}$ to be a name expression, not an expression of type **N**, so we can do a substitution $\mathcal{G}\{x \leftarrow \mathcal{N}\}$ into the type. Note that $E \vdash t : \mathbf{N}$ means that t can be any computation of type **N**, unlike $E \vdash_{\mathrm{N}} \mathcal{N}$.

A stack satisfies an environment, $\rho \vDash E$, if $\rho(x) \vDash \rho(E(x))$ for all x's in $dom(E)$; note the extra $\rho(\text{-})$ used to bind the dependent variables in $E(x)$. Here $\rho(\mathcal{F})$ or $\rho(\mathcal{A})$ means that ρ is used as a valuation (Definition 6-3). In addition, though, we require ρ to satisfy the freshness requirements of the H-quantified variables in E.

Definition 7-3 (Environment Satisfaction)

$\rho \vDash E$, for a stack ρ and environment E, is defined as follows:

$\rho \vDash n_1, \ldots, n_k$ always
$\rho \vDash E', x{:}\mathcal{A}$ iff $\exists \rho', x, F.\ \rho = \rho'[x \leftarrow F] \wedge x \notin dom(\rho') \wedge \rho' \vDash E' \wedge F \vDash \rho'(\mathcal{A})$
$\rho \vDash E', \forall x{:}\mathbf{N}$ iff $\exists \rho', x, n.\ \rho = \rho'[x \leftarrow n] \wedge x \notin dom(\rho') \wedge \rho' \vDash E'$
$\rho \vDash E', \mathsf{H}x{:}\mathbf{N}$ iff $\exists \rho', x, n.\ \rho = \rho'[x \leftarrow n] \wedge x \notin dom(\rho') \wedge \rho' \vDash E'$
$\quad \wedge\ n \notin na(E') \wedge n \notin na(\rho')$

Finally, we obtain:

Theorem 7-4 (Subject Reduction)
(1) If $E \vdash \mathcal{T} <: \mathcal{G}$ and $\rho \vDash E$ and $F \vDash_H \rho(\mathcal{T})$ then $F \vDash_H \rho(\mathcal{G})$.
(2) If $E \vdash_N \mathcal{N}$ and $\rho \vDash E$ and $\mathcal{N} \downarrow_\rho n$ then $n \vDash_N \rho(\mathcal{N})$.
(3) If $E \vdash t : \mathcal{T}$ and $\rho \vDash E$ and $t \Downarrow_\rho F$, then $F \vDash_H \rho(\mathcal{T})$.

Proof

We show the (Term ν) case of (3), which is by induction on the derivation of $E \vdash t : \mathcal{T}$.

We have $E \vdash (\nu x)t : \mathsf{H}x.\mathcal{A}$ and $\rho \vDash E$ and $(\nu x)t \Downarrow_\rho F$. We have from (Term ν) E, $\mathsf{H}x{:}\mathbf{N} \vdash_T t : \mathcal{A}$, and from (Red ν) $F = (\nu n)P$ and $t \Downarrow_{\rho[x \leftarrow n]} P$ for $n \notin na(t,\rho)$. Since n could appear in $\mathcal{A}$, blocking the last step of this proof, take $n' \notin na(t,\mathcal{A},\rho,P)$, so that $F \equiv_\alpha (\nu n')P\bullet\{n \leftrightarrow n'\}$. By Lemma 5-2 $t\bullet\{n \leftrightarrow n'\} \Downarrow_{\rho[x \leftarrow n]\bullet\{n \leftrightarrow n'\}} P\bullet\{n \leftrightarrow n'\}$, that is $t \Downarrow_{\rho[x \leftarrow n']} P\bullet\{n \leftrightarrow n'\}$. We have $\rho[x \leftarrow n'] \vDash E, \mathsf{H}x{:}\mathbf{N}$. By Ind Hyp, $P\bullet\{n \leftrightarrow n'\} \vDash \rho[x \leftarrow n'](\mathcal{A})$, that is, $P\bullet\{n \leftrightarrow n'\} \vDash \rho \setminus x(\mathcal{A})\{x \leftarrow n'\}$. Since $F \equiv (\nu n')P\bullet\{n \leftrightarrow n'\}$ and $n' \notin na(\rho \setminus x(\mathcal{A}))$, by Definition 4-1, $F \vDash \mathsf{H}x.\rho \setminus x(\mathcal{A})$. That is, $F \vDash \rho(\mathsf{H}x.\mathcal{A})$. □

8 Examples

We discuss some programming examples, using plausible (but not formally checked) extensions of the formal development of the previous sections. In particular, we use recursive types, ***rec*** $X.\ \mathcal{A}$, existential types $\exists x.\mathcal{A}$ where x ranges over names (these are simpler to handle than $\mathsf{H}x.\mathcal{A}$), and a variant of location matching, $t \div (x[y{:}\mathcal{A}]).u$, that binds labels x from the data in addition to contents y (its typing requires existential types). Examples of transposition types have been discussed in the Introduction; here we concentrate on pattern matching, using some abbreviations:

test *t* ***as*** *w*:$\mathcal{A}$ ***then*** *u* ***else*** *v*for $t?(w{:}\mathcal{A}).\ u, v$
match *t* ***as*** (*pattern*) ***then*** *u* ***else*** *v*for$t?(w{:}\mathcal{B}).\ (w \div (pattern).u), v$
where $\mathcal{B}$ is the type naturally extracted from *pattern*.

We also use nested patterns, in the examples, which can be defined in a similar way. We use standard notations for recursive function definitions. We sometimes underline binding occurrences of variables, for clarity. We explicitly list the subtypings, if any, that must be included in *ValidEntailments* for these examples to typecheck (none are needed for the examples in Section 1.2).

Basic. Duplicating a given label, and duplicating a hidden label:

$\lambda x{:}\mathbf{N}.\ \lambda y{:}x[\mathbf{T}].\ x[y] \quad : \Pi x.\ (x[\mathbf{T}] \to x[x[\mathbf{T}]])$
$\lambda z{:}(\mathsf{H}x.x[\mathbf{T}]).\ z \div ((\nu x)y{:}x[\mathbf{T}]).\ x[y] : (\mathsf{H}x.x[\mathbf{T}]) \to (\mathsf{H}x.x[x[\mathbf{T}]])$

Collect. Collect all the subtrees that satisfy $\mathcal{A}$, even under restrictions:

let type *Result* = ***rec*** $X.\ \mathbf{0} \vee \mathcal{A} \vee (X \mid X) \vee \mathsf{H}x.X$

let rec *collect*($\underline{x}$: **T**): *Result* =
(***test*** *x* ***as*** $\underline{w}{:}\mathcal{A}$ ***then*** *w* ***else*** 0) |

(***test*** *x* ***as*** $\underline{w}$:**0** ***then*** 0 ***else***
match *x* ***as*** ($\underline{y}$:¬**0** | $\underline{w}$:¬**0**) ***then*** *collect*(*y*) | *collect*(*w*) ***else***
match *x* ***as*** ($\underline{y}$[$\underline{w}$:**T**]) ***then*** *collect*($\underline{w}$) ***else***
match *x* ***as*** ((ν$\underline{y}$)$\underline{w}$:©*y*) ***then*** *collect*(*w*) ***else*** 0)

Recall that, in the last match, a (ν*y*) is automatically wrapped around the result; hence the H*x*.*X* in the definition of *Result*. The typing $\underline{w}$:©*y* (instead of $\underline{w}$:**T**) is used to reduce nondeterminism by forcing the analysis of non-redundant restrictions. Similarly, the pattern ¬**0** | ¬**0** is used to avoid vacuous splits where one component is 0. In general, the splitting of composition is nondeterministic; in this case the result may or may not be uniquely determined depending on the shape of $\mathcal{A}$. The subtypings needed here are $\mathcal{A}$ <: **T**, $\mathcal{A}$ <: $\mathcal{A}\vee\mathcal{B}$ and ***rec*** fold/unfold.

Removing Dangling Pointers. We can encode addresses and pointers in the same style as in XML. An address definition is encoded as *addr*[*n*[0]], where *addr* is a conventional name, and *n* is the name of a particular address. A pointer to an address is encoded as *ptr*[*n*[0]], where *ptr* is another conventional name. Addresses may be global, like URLs, or local, like XML's IDs; local addresses are represented by restriction: (ν*n*) ... *addr*[*n*[0]] ... *ptr*[*n*[0]] ... . A tree should not contain two address definitions for the same name, but this assumption is not important in our example.

We write a function that copies a tree, including both public and private addresses, but deletes all the pointers that do not have a corresponding address in the tree. Every time a *ptr*[*n*[0]] is found, we need to see if there is an *addr*[*n*[0]] somewhere in the tree. But we cannot search for *addr*[*n*[0]] in the original tree, because *n* may be a restricted address we have come across. So, we first open all the (non-trivial) restrictions, and then we proceed as above, passing the root of the restriction-free tree as an additional parameter. The search for *addr*[*n*[0]] can be done by a single type test for *Somewhere*(*addr*[*n*[**0**]]), where *Somewhere*($\mathcal{A}$) ≜ ***rec*** *X*. ($\mathcal{A}$ | **T**) ∨ ∃*y*.(*y*[*X*] | **T**).

let rec *deDangle*($\underline{x}$: **T**): **T** =
  match *x* ***as*** ((ν$\underline{y}$)$\underline{w}$:©*y*) ***then*** *deDangle*(*w*) ***else*** *f* (*x*, *x*)
and *f* ($\underline{x}$: **T**, $\underline{root}$: **T**): **T** =
  test *x* ***as*** $\underline{w}$:**0** ***then*** 0 ***else***
  match *x* ***as*** ($\underline{y}$:¬**0** | $\underline{w}$:¬**0**) ***then*** *f* (*y*, *root*) | *f* (*w*, *root*) ***else***
  match *x* ***as*** (*ptr*[$\underline{y}$[**0**]]) ***then***
    test *root* ***as*** $\underline{w}$:*Somewhere*(*addr*[*y*[**0**]]) ***then*** *ptr*[*y*[0]] ***else*** 0 ***else***
  match *x* ***as*** ($\underline{z}$[$\underline{w}$:**T**]) ***then*** *z*[*f* (*w*, *root*)] ***else*** 0

Note that *deDangle* automatically recloses, in the result, all the restrictions that it opens. The subtypings needed here are just $\mathcal{A}$ <: **T**.

9 Conclusions

We have introduced a language and a rich type system for manipulating trees with public and private locations. We stripped our tree model to the absolute ba-

sics, since the theory was complicated enough for this simple case. We believe that our distinction between public and private locations is an important feature of web data, and that our use of names and name hiding to model locations is fundamental. We believe the techniques and results described here will easy transfer other, more complex models of semistructured data: for example, our trees-with-pointers model which includes unique locations and trees with graphical links [13,16].

Our key contribution is a subject reduction theorem, capturing the interplay between the type system, the operational semantics and the satisfaction relation in this complex setting involving name binding transpositions and hiding quantification. We are not aware of previous uses of transpositions in structural operational semantics although it does fall within the general framework of [33]. We believe ours is the first language with explicit transpositions in the syntax of terms and types. Our formal development could have been carried out within a metatheory with transpositions, as advocated in [26,33]. In this case, Lemmas 4-3 and 5-2 would just follow from the metatheory, and one would be less exposed to mistakes associated with α-conversion. We have not gone that far, but should seriously consider this option in the future.

Acknowledgments. Thanks to Murdoch J. Gabbay for illuminating discussions on transpositions, and to Luís Caires who indirectly influenced this paper through earlier work with the first author. Moreover, Gabbay and Caires helped simplify the technical presentation.

References

[1] S. Abiteboul, P. Buneman, D. Suciu: Data on the Web. Morgan Kaufmann Publishers, 2000.

[2] S. Abiteboul, P. Kanellakis: Object identity as a query language primitive. Journal of the ACM, 45(5):798-842, 1998. A first version appeared in SIGMOD'89.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener: The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.

[4] M.P. Atkinson, F. Bancilhon, et al.: The Object-Oriented Database System Manifesto. Building an Object-Oriented Database System, The Story of O2, 1992, pp. 3-20.

[5] V. Benzaken, G. Castagna, A. Frisch: CDuce: a white paper. PLAN-X: Programming Language Technologies for XML, Pittsburgh PA, Oct. 2002. http://www.cduce.org.

[6] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, J. Siméon: XQuery 1.0: An XML Query Language, W3C Working Draft, 2002, http://www.w3.org/TR/xquery.

[7] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler: Extensible Markup Language (XML) 1.0 (Second Edition), W3C document, http://www.w3.org/TR/REC-xml.

[8] P. Buneman, S.B. Davidson, G.G. Hillebrand, D. Suciu: A Query Language and Optimization Techniques for Unstructured Data. SIGMOD Conference 1996, pp. 505-516.

[9] L. Caires: A Model for Declarative Programming and Specification with Concurrency and Mobility. Ph.D. Thesis, Dept. de Informática, FTC, Universidade Nove de Lisboa, 1999.

[10] L. Caires, L Cardelli: A Spatial Logic for Concurrency: Part I. Information and Computation, Vol 186/2 November 2003. pp 194-235.

[11] L. Caires, L. Cardelli: A Spatial Logic for Concurrency: Part II. Theoretical Computer Science, 322(3), September 2004. pp. 517-565

[12] C. Calcagno, L. Cardelli, A.D. Gordon: Deciding Validity in a Spatial Logic for Trees. Journal of Func-

tional Programming, Vol 15, Cambridge University Press, 2005. pp 543-572.

[13] C. Calcagno, P. Gardner and U. Zarfaty: Context Logic & Tree Update, Proc. POPL 2005.

[14] C. Calcagno, H. Yang, P.W. O'Hearn: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. Proc. FSTTCS 2001, pp. 108-119.

[15] L. Cardelli, P. Gardner, G. Ghelli: A Spatial Logic for Querying Graphs. Proc. ICALP'02, Peter Widmayer et al. (Eds.). LNCS 2380, Springer, 2002. pp 597-610.

[16] L. Cardelli, P. Gardner and G. Ghelli: Querying Trees with Pointers. Unpublished notes, 2003; talk at APPSEM 2001.

[17] L. Cardelli, G. Ghelli: A Query Language Based on the Ambient Logic. Proc. ESOP'01, David Sands (Ed.). LNCS 2028, Springer, 2001, pp. 1-22.

[18] L. Cardelli, A.D. Gordon: Anytime, Anywhere. Modal Logics for Mobile Ambients. Proc. of the 27th ACM Symposium on Principles of Programming Languages, 2000, pp. 365-377.

[19] S. Cluet, S. Jacqmin, and J. Simeon: The New YATL: Design and Specifications. INRIA, 1999.

[20] E. Cohen: Validity and Model Checking for Logics of Finite Multisets. Draft.

[21] M. Dam: Proof Systems for Pi-Calculus Logics. In R. de Queiroz (ed.), Logic for Concurrency and Synchronisation, Trends in Logic, Logica Library, Kluwer, 2003, pp. 145-212

[22] D. Florescu, A. Deutsch, A. Levy, D. Suciu, M. Fernandez: A Query Language for XML. In Proc. of Eighth International World Wide Web Conference, 1999.

[23] D. Florescu, A. Levy, M. Fernandez, D. Suciu, A Query Language for a Web-Site Management System. SIGMOD Record , vol. 26 , no. 3 September, 1997. pp. 4-11.

[24] M.J. Gabbay: A Theory of Inductive Definitions with a-Equivalence: Semantics, Implementation, Programming Language. Ph.D. Thesis, University of Cambridge, 2000.

[25] M.J. Gabbay, A.M. Pitts, A New Approach to Abstract Syntax Involving Binders. Proc. LICS1999. IEEE Computer Society Press, 1999. pp 214-224.

[26] M.J. Gabbay: FM-HOL, A Higher-Order Theory of Names. In Thirty Five years of Automath, Heriot-Watt University, Edinburgh, April 2002. Inforal Proc., 2002.

[27] A.D. Gordon: Notes on Nominal Calculi for Security and Mobility. R.Focardi, R.Gorrieri (Eds.): Foundations of Security Analysis and Design. LNCS 2171. Springer, 1998.

[28] A.D. Gordon, A. Jeffrey: Typing Correspondence Assertions for Communication Protocols. MFPS 17, Elsevier Electronic Notes in Theoretical Computer Science, Vol 45, 2001.

[29] H. Hosoya, B. C. Pierce: XDuce: A Typed XML Processing Language (Preliminary Report). WebDB (Selected Papers) 2000, pp: 226-244

[30] R. Milner: Communicating and Mobile Systems: the π-Calculus. Cambridge U. Press, 1999.

[31] P.W. O'Hearn, D. Pym: Logic of Bunched Implication. Bulletin of Symbolic Logic 5(2), 1999. pp 215-244.

[32] P.W. O'Hearn, J.C. Reynolds, H. Yang: Local Reasoning about Programs that Alter Data Structures. Proc. CSL 2001, pp. 1-19.

[33] A.M. Pitts: Nominal Logic, A First Order Theory of Names and Binding. Proc. TACS 2001, Naoki Kobayashi and Benjamin C. Pierce (Eds.). LNCS 2215. Springer, 2001. pp 219-242.

[34] A.M. Pitts, M.J. Gabbay: A Metalanguage for Programming with Bound Names Modulo Renaming. R. Backhouse and J.N. Oliveira (Eds.): MPC 2000, LNCS 1837, Springer, pp. 230-255.