Manipulating Trees with Hidden Labels

Luca Cardelli - Microsoft Research
Philippa Gardner - Imperial College London
Giorgio Ghelli - Università di Pisa

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.

1 Introduction

1.1 Languages for Semistructured Data

XML and semistructured data [1] are inspiring a new generation of programming and query languages based on more flexible type systems [26, 5, 6, 15]. 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 untagged unions, associative products, and Kleene star operators. The theory of formal languages, for strings and trees, 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 [26, 5].

While investigating semistructured data models and associated languages, we became aware of the need for manipulating private data elements, such as XML identifiers, unique node identifiers in graph models [7], and even heap locations. Such private resources can be modeled using *names* and *name hiding* notions arising from the π-calculus [27]: during data manipulation, the identity of a private name is not important as long as the distinctions between it and other (public or private) names are preserved. Recent progress has been made in handling private resources in programming. FreshML [21] pioneers the *transposition* [30], or swapping, of names, within a type systems that prevents the disclosure of private names.

Other recent techniques can be useful for our purposes. The spatial logics of concurrency devised to cope with π-calculus restriction and scope extrusion [27], and the separation logics used to describe data structures [28,29], provide novel logical operators that can be used also in type systems. Moreover, the notion of dependent types, when the dependence is restricted to names, is tractable [25].

In this paper we bring together a few current threads of development: the effort to devise new languages, type systems, and logics for data structures, the logical operators that come from spatial and nominal logics for private resources, the techniques of transpositions, and the necessity to handle name-dependent types when manipulating private resources. We study these issues in the context of a simplified data model: simple labeled trees with hidden labels, and programs that manipulate such trees. The edges of such trees are labeled with *names*. Our basic techniques can be applied to related data models, such as graphs with hidden node and edge labels, which will be the subject of further work.

1.2 Data Model

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;
$n[P]$	the 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);
$(\nu n)P$ the 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: $(n[\mathbf{T}] \mid \mathbf{T}) \rightarrow (\mathbf{T} \mid m[\mathbf{T}])$

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: $(n[\mathbf{T}] \mid \mathbf{T}) \rightarrow (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

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 [27] 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: $(\mathsf{H}z.\ (z[\mathbf{T}] \mid \mathbf{T})) \rightarrow (\mathsf{H}z.\ (\mathbf{T} \mid z[\mathbf{T}]))$

Note that 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*** $x \mid m[y]$ (4)
transforms $t = (\nu p)\ (p[P] \mid Q)$ into $(\nu p)(P \mid m[Q])$
expected typing: $(\mathsf{H}z.\ (z[\mathbf{T}] \mid \mathbf{T})) \rightarrow (\mathsf{H}z.\ (\mathbf{T} \mid m[\mathbf{T}]))$

This program replaces only one occurrence of p: the residual restriction (νp) guarantees that any other occurrences inside P,Q remain bound. As a consequence, the binder $\mathsf{H}z$ has to remain in the result type. Note that, although we can replace a private name with a public one, we cannot “expose” a private name, because of the rebinding of the output.

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:

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

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 $\mathcal{A}$, then $t(n{\leftrightarrow}m)$ has type $\mathcal{A}(n{\leftrightarrow}m)$. 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:

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

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 similarly 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:

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

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 [30], which is a main goal of this paper. If some type-level manipulation of names is of use, then transpositions seem a good starting point.

1.3 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 [31], although we base much of our development on the same foundations [30]. 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 $\Pi w.\ (\mathcal{A} \wedge \neg©w) \to \mathcal{B}$, where the operator $©w$ [10] means "contains free the name denoted by w".

Our calculus is based on a pattern matching construct that performs run-time type tests; in this respect, it is similar to the XML manipulation languages XDuce [26] 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 [1,3,6,8,15,17,19,20], 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 [15], 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], support structural recursion as well, for expressiveness.

As a major area of needed 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 $\mathsf{H}z.\mathcal{A}$), and other algorithms are being developed that include Kleene star [18]. 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., $\mathsf{H}z.n[\mathcal{A}]$ <: $n[\mathsf{H}z.\mathcal{A}]$ for example (5)). By parameterizing over the ValidEntailments, we show that these issues are relatively orthogonal to the handling of transpositions and hiding.

A precursor of this work handles a simpler data model, with no hiding but with a similarly rich type system based on spatial logic [12]. However, even the richer data model considered in this paper is not all one could wish for. For example, hiding makes better sense for graph nodes [14], or for addresses in heaps [29], than for tree labels. More sophisticated data models include graphs, and combinations of trees and graphical links as in practical uses of XML (see example in Section 8). In any case, the manipulation of hidden resources in data structures is fundamental.

2 Values

Our programs manipulate *values*; either *name values* (from a countable set of names Λ), *tree values*, or *function values* (i.e., closures). Over the tree values, we define a structural congruence relation $\equiv$ that factors out the equivalence laws for $\mid$ and 0, and the scoping laws for restriction. Function values are triples of a term t (Section 3) with respect to an input variable x (essentially, $\lambda x.t$) and a stack for free variables ρ. A stack ρ is a list of bindings of variables to values. Name transpositions are defined on all values.

2-1 Definition: Tree Values

Λ	Names: a countable set of names $n, m, p, ...$		
P,Q,R ::=	Tree values	All names: $na(P)$	Free names: $fn(P)$
0	void	$na(0) \triangleq \{\}$	$fn(0) \triangleq \{\}$
$P \mid Q$	composition	$na(P \mid Q) \triangleq na(P) \cup na(Q)$	$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$
$n[P]$	location	$na(n[P]) \triangleq \{n\} \cup na(P)$	$fn(n[P]) \triangleq \{n\} \cup fn(P)$
$(\nu n)P$	restriction	$na((\nu n)P) \triangleq \{n\} \cup na(P)$	$fn((\nu n)P) \triangleq fn(P) - \{n\}$

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. The interaction of transpositions with binders such as $(\nu n)P$ supports a general formal treatment of bound names [30].

2-2 Definition: Actual Transposition of Names and Tree Values.

$n{\bullet}(n{\leftrightarrow}m) = m$ $\qquad$ $0{\bullet}(m{\leftrightarrow}m') = 0$

$n{\bullet}(m{\leftrightarrow}n) = m$ $\qquad$ $(P \mid Q){\bullet}(m{\leftrightarrow}m') = P{\bullet}(m{\leftrightarrow}m') \mid Q{\bullet}(m{\leftrightarrow}m')$

$n{\bullet}(m{\leftrightarrow}m') = n$ if $n{\neq}m$ and $n{\neq}m'$ $\qquad$ $n[P]{\bullet}(m{\leftrightarrow}m') = n{\bullet}(m{\leftrightarrow}m')[P{\bullet}(m{\leftrightarrow}m')]$

$((\nu n)P){\bullet}(m{\leftrightarrow}m') = (\nu n{\bullet}(m{\leftrightarrow}m'))P{\bullet}(m{\leftrightarrow}m')$

Transpositions are used in the definition of α-congruence and capture-avoiding substitution. Structural congruence is analogous to the standard definition for π-calculus [27]; the "scope extrusion" rule for ν over -|- is written in an equivalent equational style.

2-3 Definition: α-Congruence and Structural Congruence on Tree Values.

α-congruence, $\equiv_\alpha$, is the least congruence relation on tree values such that:

$(\nu n)P \equiv_\alpha (\nu m)(P{\bullet}(n{\leftrightarrow}m))$ $\quad$ where $m{\notin}na(P)$

Structural congruence, $\equiv$, is the least congruence relation on tree values such that:

$P \equiv_\alpha Q \Rightarrow P \equiv Q$ $\qquad$ $(\nu n)0 \equiv 0$

$P \mid Q \equiv Q \mid P$ $\qquad$ $(\nu n)m[P] \equiv m[(\nu n)P]$ if $n{\neq}m$

$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ $\qquad$ $(\nu n)(P \mid (\nu n)Q) \equiv ((\nu n)P) \mid ((\nu n)Q)$

$P \mid 0 \equiv P$ $\qquad$ $(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$

N.B.: This notion of α-congruence can be shown equivalent to the standard one.

2-4 Definition: Free Name Substitution on Tree Values.

$0\{n{\leftarrow}m\} = 0$ $\qquad$ $(P \mid Q)\{n{\leftarrow}m\} = P\{n{\leftarrow}m\} \mid Q\{n{\leftarrow}m\}$

$p[P]\{n{\leftarrow}m\} = p\{n{\leftarrow}m\}[P\{n{\leftarrow}m\}]$

$((\nu p)P)\{n{\leftarrow}m\} = (\nu q)((P{\bullet}(p{\leftrightarrow}q))\{n{\leftarrow}m\})$ $\quad$ for $q{\notin}na((\nu p)P){\cup}\{n,m\}$

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

We next define high values and transpositions over them (see also Definition 3-1 for the syntax of terms t). A stack ρ is a list of pairs of the form $\emptyset[x_1{\leftarrow}F_1]...[x_n{\leftarrow}F_n]$, where x_i are variables (distinct from names), F_i are high values, $\rho(x_i) \triangleq F_j$ where j is the largest index such that $x_i{=}\,x_j$, and $dom(\rho) \triangleq \{x_1, ..., x_n\}$. Variables are not affected by transpositions.

$\langle\rho, x, t\rangle{\bullet}(n{\leftrightarrow}n') \triangleq \langle\rho{\bullet}(n{\leftrightarrow}n'), x, t{\bullet}(n{\leftrightarrow}n')\rangle$

$\emptyset{\bullet}(n{\leftrightarrow}n') \triangleq \emptyset$

$\rho[x{\leftarrow}F]{\bullet}(n{\leftrightarrow}n') \triangleq \rho{\bullet}(n{\leftrightarrow}n')[x{\leftarrow}F{\bullet}(n{\leftrightarrow}n')]$

2-5 Definition: High Values and Stacks

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

3 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 [16, 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.

3-1 Definition: Syntax

$\mathcal{N},\mathcal{M},\mathcal{P},\mathcal{Q}$ ::=	*Name Expressions*
$x,\ n,\ \mathcal{N}\llparenthesis\mathcal{M}\leftrightarrow\mathcal{M}'\rrparenthesis$	name variable, name constant, name transposition
$\mathcal{A},\mathcal{B}$ ::=	*Tree Types*
$\mathbf{0},\ \mathcal{N}[\mathcal{A}],\ \mathcal{A}\mid\mathcal{B},\ \mathsf{H}\underline{x}.\mathcal{A},\ ©\mathcal{N},$	void, location, composition, hiding, occurrence,
$\mathbf{F},\ \mathcal{A}\wedge\mathcal{B},\ \mathcal{A}\Rightarrow\mathcal{B},\ \mathcal{A}\llparenthesis\mathcal{M}\leftrightarrow\mathcal{M}'\rrparenthesis$	false, conjunction, implication, type transposition
$\mathcal{F},\mathcal{G},\mathcal{H}$ ::=	*High Types*
$\mathcal{A},\ \mathbf{N},$	tree types, name type,
$\mathcal{F}\to\mathcal{G},\ \Pi\underline{x}.\mathcal{G}$	function types ($\mathcal{F}\neq\mathbf{N}$), dependent types ($x$:**N**)
t,u,v ::=	*Terms*
$0,\ \mathcal{N}[u],\ t\mid u,\ (\nu\underline{x})t,$	void, location, composition, restriction,
$t\div(\mathcal{N}[\underline{y}:\mathcal{A}]).u,\ t\div(\underline{x}:\mathcal{A}\mid\underline{y}:\mathcal{B}).u,$	location match, composition match,
$t\div((\nu\underline{x})\underline{y}:\mathcal{A}).u,\ t?(\underline{x}:\mathcal{A}).u,v,$	restriction match, tree type test,
$t\llparenthesis\mathcal{M}\leftrightarrow\mathcal{M}'\rrparenthesis,$	term transposition,
$x,\ \mathcal{N},\ \lambda\underline{x}:\mathcal{F}.t,\ t(u)$	high variable, name expr, function, application

Underlined variables indicate binding occurrences. The scoping rules should be clear, except that: in location match $\underline{y}$ scopes u; in composition match $\underline{x}$ and $\underline{y}$ scope u; in restriction match $\underline{x}$ scopes $\mathcal{A}$ and u, and $\underline{y}$ scopes u; in tree type test $\underline{x}$ scopes u and v. We define name sets, such as $na(\mathcal{A})$, and actual transpositions on all syntax, such as $t\bullet\llparenthesis n\leftrightarrow m\rrparenthesis$, 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: $\mathcal{N}\{x\leftarrow\mathcal{M}\}$, $\mathcal{A}\{x\leftarrow\mathcal{M}\}$, and $\mathcal{F}\{x\leftarrow\mathcal{M}\}$.

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

4 Satisfaction

The satisfaction relation, written $\vDash$, 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 notion of satisfaction from modal logic. Over tree types we have essentially the relation studied in [16, 10], extended to hiding and transpositions. Satisfaction is then generalized to high types, where it depends on the operational semantics $\Downarrow_\rho$ of Section 5, which depends on satisfaction at tree types only.

4-1 Definition: Satisfaction

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

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

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

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

On High Types: $F \vDash_H \mathcal{F}$, for $\mathcal{F}$ 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 $F \vDash_T \mathcal{A}$
$H \vDash_H \mathcal{F}\rightarrow\mathcal{G}$ ($\mathcal{F}\neq\mathbf{N}$)	iff $H = \langle \rho, z, t\rangle$ and $\forall F,G.\ (F \vDash_H \mathcal{F} \wedge t \Downarrow_{\rho[z\leftarrow F]} G) \Rightarrow G \vDash_H \mathcal{G}$
$H \vDash_H \Pi x.\,\mathcal{G}$	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 $©\mathcal{N}$ 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 2-3). The type $\mathsf{H}w.©w$ is the type of non-redundant restrictions, with the quantifier $\mathsf{H}w$ revealing a restricted name n, and $©w$ 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 $\mathcal{F}\rightarrow\mathcal{G}$ if, on any input satisfying $\mathcal{F}$, every output satisfies $\mathcal{G}$; similarly for $\Pi x.\,\mathcal{G}$.

4-2 Proposition: Tree Satisfaction Under Structural Congruence.
If $P \vDash \mathcal{A}$ and $P \equiv Q$ then $Q \vDash \mathcal{A}$.

4-3 Lemma: Name and 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 $P\bullet(m\leftrightarrow m') \vDash \mathcal{A}\bullet(m\leftrightarrow m')$.

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.

5-1 Definition: Operational Semantics

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

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

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

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

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

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

The operations (Red ÷ -) and (Red ? -) can execute run-time type tests on dependent types that are run-time instantiated; e.g., note the role of x in $\lambda x:\mathbf{N}.\ t?(y:x[\mathbf{0}]).u,v$. Here, $\rho(\mathcal{A})$ 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: the type rules of Section 7 guarantee this condition.

The matching reductions are nondeterministic and, in a big step semantics, avoid divergent paths if convergent paths are possible.

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.

5-2 Lemma: 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 [22,11]. A crucial equivalence rule, (EqN ↔ Apart), 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*.

6-1 Definition: Freshness Signature

A *freshness signature* ϕ is an ordered list of distinct variables annotated with either ∀ or H, and of names. (For example: $p,\forall x,\mathsf{H}y,n,m,p,\mathsf{H}z,\forall w$.) Notation: $dom(\phi)$ is the set of variables in ϕ; $na(\phi)$ is the set of names in ϕ; $\phi(x)$ is the symbol ∀ 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 $\phi\supseteq\mathcal{F}$.

Next we define three equivalence relations between name expressions, $\sim_\mathrm{N}$, tree types, $\sim_\mathrm{T}$, and high types, $\sim_\mathrm{H}$ (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.

6-2 Definition: Equivalence and Apartness

$\mathcal{N} \sim_{\mathrm{N}\phi} \mathcal{M}$ (a congruence, abbrev. $\mathcal{N} \sim_\phi \mathcal{M}$), and $\mathcal{N} \#_\phi \mathcal{M}$ (a symmetric relation) are the least such relations on name expressions such that $\phi\supseteq\mathcal{N},\mathcal{M}$, and:

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

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

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

they are congruences including α-conversion; we highlight:
$\mathcal{N} \sim_{N\phi} \mathcal{N}'$ and $\mathcal{A} \sim_{\phi} \mathcal{A}' \;\Rightarrow\; \mathcal{N}'[\mathcal{A}'] \sim_{\phi} \mathcal{N}'[\mathcal{A}']$ (EqT $\mathcal{N}$[] Congr)
$\mathcal{A} \sim_{(\phi,\mathsf{H}x)} \mathcal{A}' \;\Rightarrow\; \mathsf{H}x.\, \mathcal{A} \sim_{\phi} \mathsf{H}x.\, \mathcal{A}'$ (EqT H Congr)
$\mathsf{H}x.\, \mathcal{A} \sim_{\phi} \mathsf{H}y.\, \mathcal{A}\{x \leftarrow y\}$ with $y \notin fv(\mathcal{A})$ (EqT H-α)

they distribute transpositions over all type constructors; we highlight:
$\mathbf{0}(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_{\phi} \mathbf{0}$ (EqT $\mathbf{0}$ ↔)
$(\mathcal{N}[\mathcal{A}])(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_{\phi} \mathcal{N}(\mathcal{M} \leftrightarrow \mathcal{M}')[\mathcal{A}(\mathcal{M} \leftrightarrow \mathcal{M}')]$ (EqT $\mathcal{N}$[] ↔)
$(\mathsf{H}x.\mathcal{A})(\mathcal{M} \leftrightarrow \mathcal{M}') \sim_{\phi}$ (EqT H ↔)
$\quad \mathsf{H}x.(\mathcal{A}\{x \leftarrow x(\mathcal{M} \leftrightarrow \mathcal{M}')\}(\mathcal{M} \leftrightarrow \mathcal{M}'))$ with $x \notin fv(\mathcal{M},\mathcal{M}')$

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

they are congruences including α-conversion; we highlight the cases for Π:
$\mathcal{F} \sim_{(\phi,\forall x)} \mathcal{G} \;\Rightarrow\; \Pi x.\, \mathcal{F} \sim_{\phi} \Pi x.\, \mathcal{G}$ (EqH Π Congr)
$\Pi x.\, \mathcal{G} \sim_{H\phi} \Pi y.\, \mathcal{G}\{x \leftarrow y\}$ with $y \notin fv(\mathcal{G})$ (EqH Π-α)

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

6-3 Definition: Valuation.

If $\varepsilon : Var \rightarrow \Lambda$ is a finite map, then we say that ε is a *valuation*.
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 $x \notin dom(\varepsilon)$.
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 $\mathcal{F}$.

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

6-4 Definition: 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 $\forall y \in dom(\varepsilon).\ (x <_{\phi} y \wedge \phi(y) = \mathsf{H}) \Rightarrow (x \in dom(\varepsilon) \wedge \varepsilon(x) \neq \varepsilon(y))$

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).

The following soundness result requires some careful build-up: lemmas for instantiations of equivalence and apartness under partial valuations, for closure of satisfaction under closed equivalence, and substitution lemmas. We omit the details.

6-5 Proposition: Soundness of Equivalence and Apartness.

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

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 write $E,x{:}\mathcal{T}$ and $E,\mathsf{Q}x{:}\mathbf{N}$ for the extension of E with a new 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. Moreover, in Definition 7-1 below we extract the freshness signature associated with an environment:

7-1 Definition: Freshness Signature of an Environment

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

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

7-2 Definition: Type Rules

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

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env n)
$$\frac{E \vdash \diamond}{E,\ n \vdash \diamond}$$

(Env $x{\in}\mathbf{N}$)
$$\frac{E \vdash \diamond \quad \mathsf{Q}{\in}\{\forall,\mathsf{H}\} \quad x{\notin}dom(E)}{E,\ \mathsf{Q}x{:}\mathbf{N} \vdash \diamond}$$

(Env $x{\notin}\mathbf{N}$)
$$\frac{E \vdash \mathcal{T} \quad \mathcal{T}{\neq}\mathbf{N} \quad x{\notin}dom(E)}{E,\ x{:}\mathcal{T} \vdash \diamond}$$

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

(NExpr n)
$$\frac{E \vdash \diamond \quad n{\in}na(E)}{E \vdash_{\mathrm{N}} n}$$

(NExpr x)
$$\frac{E \vdash \diamond \quad E(x){=}\mathbf{N}}{E \vdash_{\mathrm{N}} x}$$

(NExpr $\leftrightarrow$)
$$\frac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash_{\mathrm{N}} \mathcal{N}[\mathcal{M}{\leftrightarrow}\mathcal{M}']}$$

Types. Rules for $E \vdash_{\mathrm{T}} \mathcal{A}$ and $E \vdash \mathcal{T}$ (that is, $\mathcal{A}$ is a tree type and $\mathcal{T}$ is a type in E).
The rules are naturally syntax-driven, we highlight:

(Type H)
$$\frac{E,\ \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} \mathcal{A}}{E \vdash_{\mathrm{T}} \mathsf{H}x.\mathcal{A}}$$

(Type $\to$)
$$\frac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T}{\neq}\mathbf{N}}{E \vdash \mathcal{T}{\to}\mathcal{G}}$$

(Type Π)
$$\frac{E,\ \forall x{:}\mathbf{N} \vdash \mathcal{G}}{E \vdash \Pi x.\ \mathcal{G}}$$

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

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

(Sub Equiv)
$$\frac{E \vdash \mathcal{T} \quad E \vdash \mathcal{G} \quad \mathcal{T} \sim_{fs(E)} \mathcal{G}}{E \vdash \mathcal{T} <: \mathcal{G}}$$

(Sub $\mathbf{N}$)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{N} <: \mathbf{N}}$$

(Sub $\to$)
$$\frac{E \vdash \mathcal{T}' <: \mathcal{T} \quad E \vdash \mathcal{G} <: \mathcal{G}' \quad \mathcal{T}{,}\mathcal{T}'{\neq}\mathbf{N}}{E \vdash \mathcal{T}{\to}\mathcal{G} <: \mathcal{T}'{\to}\mathcal{G}'}$$

(Sub Π)
$$\frac{E,\ \forall x{:}\mathbf{N} \vdash \mathcal{G} <: \mathcal{G}'}{E \vdash \Pi x.\ \mathcal{G} <: \Pi x.\ \mathcal{G}'}$$

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)
$$\frac{E \vdash \diamond}{E \vdash 0 : \mathbf{0}}$$

(Term $\mathcal{N}$[])
$$\frac{E \vdash_{\mathrm{N}} \mathcal{N} \quad E \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash \mathcal{N}[t] : \mathcal{N}[\mathcal{A}]}$$

(Term |)
$$\frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{T}} u : \mathcal{B}}{E \vdash t \mid u : \mathcal{A} \mid \mathcal{B}}$$

(Term ν)

$$\frac{E, \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} t : \mathcal{A}}{E \vdash (\nu x)t : \mathsf{H}x.\mathcal{A}}$$

(Term ↔)

$$\frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \quad E \vdash_{\mathrm{N}} \mathcal{M} \quad E \vdash_{\mathrm{N}} \mathcal{M}'}{E \vdash t\langle\mathcal{M}{\leftrightarrow}\mathcal{M}'\rangle : \mathcal{A}\langle\mathcal{M}{\leftrightarrow}\mathcal{M}'\rangle}$$

(Term $\div\mathcal{N}[]$)

$$\frac{E \vdash_{\mathrm{T}} t : \mathcal{N}[\mathcal{A}] \quad E, y{:}\mathcal{A} \vdash u : \mathcal{F}}{E \vdash t{\div}(\mathcal{N}[y{:}\mathcal{A}]).u : \mathcal{F}}$$

(Term $\div|$)

$$\frac{E \vdash_{\mathrm{T}} t : \mathcal{A} \mid \mathcal{B} \quad E, x{:}\mathcal{A}, y{:}\mathcal{B} \vdash u : \mathcal{F}}{E \vdash t{\div}(x{:}\mathcal{A} \mid y{:}\mathcal{B}).u : \mathcal{F}}$$

(Term $\div\nu$)

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

(Term ?)

$$\frac{E \vdash_{\mathrm{T}} t : \mathcal{B} \quad E, x{:}\mathcal{A} \vdash u : \mathcal{F} \quad E, x{:}\neg\mathcal{A} \vdash v : \mathcal{F}}{E \vdash t?(x{:}\mathcal{A}).u,v : \mathcal{F}}$$

(Term x)

$$\frac{E \vdash \diamond \quad x \in dom(E)}{E \vdash x : E(x)}$$

(Term $\mathcal{N}$)

$$\frac{E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash \mathcal{N} : \mathbf{N}}$$

(Term λ)

$$\frac{E, x{:}\mathcal{F} \vdash t : \mathcal{G} \quad \mathcal{F} \neq \mathbf{N}}{E \vdash \lambda x{:}\mathcal{F}.t : \mathcal{F} \to \mathcal{G}}$$

(Term App)

$$\frac{E \vdash t : \mathcal{F} \to \mathcal{G} \quad E \vdash u : \mathcal{F}}{E \vdash t(u) : \mathcal{G}}$$

(Term Depλ)

$$\frac{E, \forall x{:}\mathbf{N} \vdash t : \mathcal{G}}{E \vdash \lambda x{:}\mathbf{N}.t : \Pi x.\, \mathcal{G}}$$

(Term DepApp)

$$\frac{E \vdash t : \Pi x.\, \mathcal{G} \quad E \vdash_{\mathrm{N}} \mathcal{N}}{E \vdash t(\mathcal{N}) : \mathcal{G}\{x \leftarrow \mathcal{N}\}}$$

(Subsumption)

$$\frac{E \vdash t : \mathcal{F} \quad E \vdash \mathcal{F} <: \mathcal{G}}{E \vdash t : \mathcal{G}}$$

Notes: • 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* $\mathcal{A},\mathcal{B}$) $\Rightarrow \forall P.\ P \vDash \varepsilon(\mathcal{A}) \Rightarrow P \vDash \varepsilon(\mathcal{B})$).

• 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 1.3. 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 $\mathcal{F}$, since $\mathcal{F}$ can depend only on variables of type $\mathbf{N}$. 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 $\mathbf{N}$, 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 $\mathbf{N}$, 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). Moreover, we require ρ to satisfy the freshness constraints implicit in E. We write $\rho\backslash x$ for the restriction of ρ to $dom(\rho)$-$\{x\}$.

7-3 Definition: Environment Satisfaction

$\rho \vDash E$ (where E is normalized with all the names to the left) is defined as follows:

$\rho \vDash n_1, ..., n_k$ 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 $\exists\rho',x,n.\ \rho = \rho'[x \leftarrow n] \wedge x \notin dom(\rho') \wedge \rho' \vDash E' \wedge n \notin na(E') \wedge n \notin na(\rho')$

7-4 Theorem: Subject Reduction.

(1) If $E \vdash \mathcal{F} <: \mathcal{G}$ and $\rho \vDash E$ and $F \vDash_{\mathrm{H}} \rho(\mathcal{F})$ then $F \vDash_{\mathrm{H}} \rho(\mathcal{G})$.

(2) If $E \vdash_{\mathrm{N}} \mathcal{N}$ and $\rho \vDash E$ and $\mathcal{N} \downarrow_{\rho} n$ then $n \vDash_{\mathrm{N}} \rho(\mathcal{N})$.

(3) If $E \vdash t : \mathcal{F}$ and $\rho \vDash E$ and $t \Downarrow_{\rho} F$, then $F \vDash_{\mathrm{H}} \rho(\mathcal{F})$.

Proof

We show the (Term ν) case of (3), which is by induction on the derivation of $E \vdash t : \mathcal{F}$. 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) $E, \mathsf{H}x{:}\mathbf{N} \vdash_{\mathrm{T}} t : \mathcal{A}$, 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, $P{\bullet}\{n\leftrightarrow n'\} \vDash \rho\backslash x(\mathcal{A})\{x\leftarrow n'\}$. Since $F \equiv (\nu n')P{\bullet}\{n\leftrightarrow n'\}$ and $n' \notin na(\rho\backslash x(\mathcal{A}))$, by Definition 4-1, $F \vDash \mathsf{H}x.\rho\backslash x(\mathcal{A})$. 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, $t{\div}(\underline{x}[\underline{y}{:}\mathcal{A}]).u$, 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:

$$\begin{array}{lcl} \lambda x{:}\mathbf{N}.\ \lambda y{:}x[\mathbf{T}].\ x[y] & : & \Pi x.\ (x[\mathbf{T}] \rightarrow x[x[\mathbf{T}]]) \\ \lambda z{:}(\mathsf{H}x.x[\mathbf{T}]).\ z{\div}((\nu x)y{:}x[\mathbf{T}]).\ x[y] & : & (\mathsf{H}x.x[\mathbf{T}]) \rightarrow (\mathsf{H}x.x[x[\mathbf{T}]]) \end{array}$$

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($\underline{x}$: $\mathbf{T}$): Result =
    (test x as $\underline{w}{:}\mathcal{A}$ then w else 0) |
    (test x as $\underline{w}{:}\mathbf{0}$ then 0 else
     match x as ($\underline{y}{:}\neg\mathbf{0} \mid \underline{w}{:}\neg\mathbf{0}$) then collect(y) | collect(w) else
     match x as ($\underline{y}[\underline{w}{:}\mathbf{T}]$) then collect($\underline{w}$) else
     match x as (($\nu\underline{y}$)$\underline{w}$:©y) then collect(w) else 0)
```

Recall that, in the last match, a (νy) is automatically wrapped around the result; hence the $\mathsf{H}x.X$ in the definition of *Result*. The typing $\underline{w}$:©y (instead of $\underline{w}{:}\mathbf{T}$) is used to reduce nondeterminism by forcing the analysis of non-redundant restrictions. Similarly, the pattern $\neg\mathbf{0} \mid \neg\mathbf{0}$ 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 $\mathcal{A} <: \mathbf{T}$, $\mathcal{A} <: \mathcal{A}\vee\mathcal{B}$ 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: (νn) ... $addr[n[0]]$ ... $ptr[n[0]]$ ... . 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 $Somewhere(addr[n[\mathbf{0}]])$, where $Somewhere(\mathcal{A}) \triangleq \textbf{\textit{rec}}\ X.\ (\mathcal{A} \mid \mathbf{T}) \vee \exists y.(y[X] \mid \mathbf{T})$.

```
let rec $deDangle(\underline{x}: \mathbf{T}): \mathbf{T}$ =
    match $x$ as $((\nu\underline{y})\underline{w}{:}©y)$ then $deDangle(w)$ else $f(x, x)$
and $f(\underline{x}: \mathbf{T}, \underline{root}: \mathbf{T}): \mathbf{T}$ =
    test $x$ as $\underline{w}{:}\mathbf{0}$ then $0$ else
    match $x$ as $(\underline{y}{:}\neg\mathbf{0} \mid \underline{w}{:}\neg\mathbf{0})$ then $f(y, root) \mid f(w, root)$ else
    match $x$ as $(ptr[\underline{y}[\mathbf{0}]])$ then
        test $root$ as $\underline{w}{:}Somewhere(addr[y[\mathbf{0}]])$ then $ptr[y[0]]$ else $0$ else
    match $x$ as $(\underline{z}[\underline{w}{:}\mathbf{T}])$ 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 $\mathcal{A} <: \mathbf{T}$.

9 Conclusions and Acknowledgments

We have introduced a language and a rich type system for manipulating semistructured data with hidden labels and scope extrusion, via pattern matching and transpositions.

As advocated in [23,30], our formal development could be carried out within a metatheory with transpositions; then, Lemmas 4-3 and 5-2 would fall out of the metatheory, and one could be less exposed to mistakes in α-conversion issues. We have not gone that far, but we should seriously consider this option in the future.

We are not aware of previous uses of transpositions in structural operational semantics, although this falls within the general framework of [30]. We believe ours is the first calculus or language with explicit transpositions in the syntax of terms and types.

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**. Proc. TACS 2001, Naoki Kobayashi and Benjamin C. Pierce (Eds.). LNCS. 2215. Springer, 2001, pp 1-37. To appear in I&C.

[11] L. Caires, L. Cardelli: **A Spatial Logic for Concurrency: Part II**. Proc. CONCUR'02, 2002.

[12] C. Calcagno, L. Cardelli, A.D. Gordon, **Deciding Validity in a Spatial Logic for Trees**. Draft.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

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

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

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

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

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

[30] 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.

[31] 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.