Binding Structures

Carolyn Talcott

Computer Science Department
Stanford University
Stanford, California 94305

1 Introduction

Binding structures enrich traditional abstract data types by providing support for representing binding mechanisms and structures with holes. They are intended to facilitate the representation and manipulation of symbolic structures such as programs, logical formulae, rules, derivations, specifications, and modules.

Practical symbolic computation has its origin in the work of McCarthy. The basic ideas are presented in [17], which is the beginning of the language Lisp. The principal data domain of Lisp is the S-expression domain that is generated by a pairing operation from a set of atomic objects including symbols and numbers. S-expressions were designed as a representation of symbolic information that is easy to read, print, and operate on. An important concept introduced in [18] is that of *abstract syntax.* The abstract syntax of a language specifies the set of expressions and operations for synthesis and analysis abstractly, independent of any printed notation or machine representation. It is defined by giving a signature and axioms. The signature consists of names for constructors of expressions (the synthetic part), and for operations recognizing the various constructions and selecting components (the analytic part). The axioms specify the relations among the constructors, selectors, and recognizers.

The motivation for our work on binding structures comes from an interest in building tools such as theorem provers, transformers, static analyzers, evaluators, rewriters, etc. that manipulate symbolic structures (both automatically and under interactive control). In particular, we are interested in understanding the architecture of such systems, and in identifying generic components that can serve as major building blocks. We briefly review some of the issues of concern. Many symbolic structures embody notions of binding. This means that many rules require hygiene side-conditions to avoid bound/free variable name conflicts. There are other sorts of side conditions to be expressed as well, such as a variable occurring at most or exactly once. It is also important to be able to describe operations or sites of rule applications in terms of sets of occurrences (paths) in a structure. Rules are often expressed informally as schemata, and rule applications are often expressed in terms of allowed syntactic contexts (expressions with holes). In addition, syntactic contexts are important for expressing many properties of expressions and for carrying out symbolic evaluation. Context filling is a mechanism for capturing free variables in contrast to substitution for free variables, which avoids capture. Finally, in developing efficient programs for manipulation of symbolic structures, it is important to be able to express sharing and updating optimizations for algorithms and to have a clear semantics of the structures that support such refinements.

Abstract syntax in the guise of abstract data types and related work in initial algebras (cf. [9]) and algebraic methods (cf. [6]) has proved highly useful in a number of contexts, including programming methodology [10]; generic programming environments (cf. [21, 2]); and programming language semantics (cf. [3]). However, traditional work on abstract data types provides no mechanism for handling binding operators, or for dealing with mutable data.

The notion of binding structure generalizes de Bruijn notation [5] and more recent theories of explicit substitution (cf. [1]) in several ways, providing for named free variables, nameless bound variables, holes, and a corresponding family of replacement operations. The notion of binding operator is reminiscent of NuPrl syntax (cf. [12]). It captures the essence of higher-order abstract syntax (cf. [13, 11, 20]), but is more flexible and does not hide the structure of objects in abstract higher-order entities. Two basic ingredients in the representation of binding structures are binding operators and binding paths. A binding operator comes equipped with a finite set of binding indices (binders), a finite set of argument selectors, and a binding relation on binder-selector pairs. Binders allow for multiple binding (keyword binding positions), argument selectors describe the arity of the operator (keyword arguments), and the binding relation specifies which binders are bound in which arguments. Binding operators generalize the traditional notion of an algebraic signature that has only arity information. Binding structures can be free variables, bound variables, holes, or binding operator applications. The intuition is that a binding structure is a tree with edges from the set of selectors and nodes labeled by node type and additional simple data. A binding path determines the node in the containing structure where a variable is bound. A binding path can be thought of as a sequence of selectors leading from the binding operator node to the bound variable node. Together with a binding index, a binding path determines the binding point of a bound variable. The binding point of a bound variable may lie outside the currently known part of a binding structure. This means that binding structures are closed under subtree selection. A hole has an associated substitution mapping identifiers to binding structures. This is used to establish binding points of free variables (trapping) and to remember substitutions that have been applied to a structure containing the hole. There are three fundamental replacement operations on binding structures: substitution for free variables, unbinding of externally bound variables, and filling of holes. One consequence of having substitutions associated with holes is that replacement operations compose in a natural way, and filling and substitution commute according to a simple law. In addition to the representation of binding structures using binding paths, we also give a representation corresponding to traditional notation, with names for both bound and free variables. To account for externally bound variables, we introduce the notion of naming context. An important consequence of decorating holes with substitutions is that there is a similarity relation on structures with named bound variables that generalizes the usual notion of renaming of bound variables. This is in contrast to the usual treatment of syntactic contexts, where the presence of holes in the scope of a bound variable prevents renaming. We define a map from structures with named bound variables to structures with binding paths. The similarity relation on structures with named bound variables is the equivalence induced by this mapping. The alternative representations of binding structures allow us to describe operations in a variety of ways: locally by structural recursion, globally in terms of tree domains and labels, or as relations on structures with named bound variables. The view of binding structures as trees directly provides for representation of substructure occurrences as tree nodes.

Binding structures have a rich algebraic theory. This theory is the subject of [22]. The theory of binding structures provides a natural starting point for developing generic tools for rewriting and for a more intensional treatment of syntactic structures. There are a number of features to be added to our theory to meet long-term goals. These include: annotations; generic tools for structure walking, matching, and unification; and representation binding structures in terms of mutable structures. To add annotations, one must provide mechanisms for adding annotations, for definition of alternative notions of equality (depending on how annotations are to be accounted for), and for lifting operations to annotated structures or making new annotations transparent to existing operations. A good example of the power of generic structure traversal algorithms is the "code walker" tool used in PCL, a portable implementation of the Common Lisp Object System described in [4]. This tool is used in many ways, including: code analyzers, macro definitions, and transformers. The basic treatment of unification algorithms as transformations on systems of equations [8] generalizes nicely to binding structures. This provides a general framework for designing new unification algorithms and gives rise to many interesting questions. Methods of [15, 14] will be used to develop a theory of mutable binding structures, and to develop systematic methods for refining abstract specifications into efficient implementations.

The remainder of this paper is organized as follows. Section 2 summarizes the mathematical notation and notions used. Section 3 defines the set of binding tree structures and replacement operations and illustrates the utility of the underlying tree structure for giving global characterizations of domains and operations. Section 4 presents the representation of binding structures using bound variable names. A map establishing the connection with binding trees and the similarity relation are defined and specification of operations as relations on structures with named bound variables is illustrated. Section 5 contains a brief discussion of related work.

2 Notation Conventions

Let $X, Y, X_1, \ldots X_n$ be sets. We specify meta-variable conventions in the form: let x range over X, which should be read as the meta-variable x, and decorated variants such as x', x_0, ..., range over the set X. $P_\omega[X]$ is the set of finite subsets of X. X^* is the set of finite sequences from X. $\mathtt{mt}$ is the empty sequence and, for m a natural number, we write $[x_i \mid i < m]$ for the sequence of length m, and $i+1$-st element m_i for $0 \leq i < m$. If $u = [x_i \mid i < m]$, $v = [x_{i+j} \mid j < k]$, we write $[u, v]$ for $[x_i \mid i < k + m]$. Notationally, we do not distinguish elements x of X from singleton sequences $[x]$. Thus, we write $[x, u]$ for $[x, x_0, \ldots, x_{m-1}]$, and $[u, x]$ for $[x_0, \ldots, x_{m-1}, x]$.

$X \to Y$ is the set of (total) functions from X to Y. If the range of f is contained in the domain of f', then $f \circ f'$ is the composition of f and f' and $(f \circ f')(x) = f(f'(x))$ for x in the domain of f. Given two functions, $f \in [X \to Y]$ and $f' \in [X' \to Y']$, we define $f' \triangleleft f$ to be the join with right adjunct taking precedence. Thus $f' \triangleleft f \in X \cup X' \to Y \cup Y'$, $(f' \triangleleft f)(x) = f(x)$ if $x \in X$, and $(f' \triangleleft f)(x) = f'(x)$ if $x \in X' - X$. $\mathit{Fmap}[X, Y]$ is the set of finite maps from X to Y. $\mathtt{mt}$ is the finite map with empty domain, and we write $\{x_1 := y_1, \ldots, x_n := y_n\}$ for the map μ with domain $\{x_1, \ldots, x_n\}$ and range $\{y_1, \ldots, y_n\}$ such that $\mu(x_i) = y_i$ for $1 \leq i \leq n$.

Trees are fundamental in our representation of binding structures. The set of finite trees with edges from X, $\mathit{Tree}[X]$, is the set of finite sets of sequences from X closed under prefix,

$$\mathit{Tree}[X] = \{T \in P_\omega[X^*] \mid (\forall x \in X, u \in X^*)([u, x] \in T \ \rightarrow\ u \in T)\}.$$

We call the elements of a tree T its nodes. We visualize trees as growing down from the root and say that one node is below another if the first is a prefix of the second. Some useful operations on trees are: subtree selection, forming the set of edge selectors at a node, and construction of a tree from a set of edges and associated subtrees. If $T \in \mathit{Tree}[X]$, $u \in T$, and $\mu \in \mathit{Fmap}[X, \mathit{Tree}[X]]$, then $T \wr u$ is the subtree of T at u, $es(T, u)$ is the set of edge selectors of T at u, and $mkT(\mu)$ is the tree T such that $es(T, \mathtt{mt}) = dom(\mu)$ and $T \wr x = \mu(x)$ for $x \in dom(\mu)$. These operations are defined as follows.

Definition (tree operations):

$$T \wr u = \{v \mid [u, v] \in T\},$$

$$es(T, u) = \{x \in X \mid [u, x] \in T\},$$

$$mkT(\mu) = \{\mathtt{mt}\} \cup \{[x, u] \mid x \in dom(\mu)\ \wedge\ u \in dom(\mu(x))\}.$$

The set of L labeled trees over X, $\mathit{Ltree}[X, L]$, is the set of finite maps with domain in $\mathit{Tree}[X]$ and range L.

$$\mathit{Ltree}[X, L] = \mathit{Fmap}[\mathit{Tree}[X], L].$$

The operations on trees are extended naturally to labeled trees. If $\tau \in \mathit{Ltree}[X, L]$ and $v \in dom(\tau)$, then $\tau \wr v$ is the labeled tree with domain $dom(\tau) \wr v$ such that $(\tau \wr v)(u) = \tau([v, u])$ for $[v, u] \in dom(\tau)$. If $l \in L$ and $\mu \in \mathit{Fmap}[X, \mathit{Ltree}[X, L]]$, then $mkLt(l, \mu)$ is the labeled tree with root labeled l and subtree $\mu(x)$ along edge x for $x \in dom(\mu(x))$,

Definition (labeled tree operations):

$$\tau \wr v = \lambda u \in dom(\tau) \wr v.\tau[v, u],$$

$$es(\tau, u) = es(dom(\tau), u),$$

$$mkLt(l, \mu) = \lambda u \in dom(mkT(\lambda x \in dom(\mu).dom(\mu(x)))).$$
$$\begin{cases} l & \text{if } u = \mathtt{mt}, \\ \mu(x)(v) & \text{if } u = [x, v] \text{ with } x \in dom(\mu), \\ & \text{and } v \in dom(\mu(x)). \end{cases}$$

3 Binding Tree Structures

In this section, we describe the representation of binding structures as labeled trees. We begin by describing binding operations and an abstract notion of binding path. Next we define the edge and label domains for binding trees, and define the set of binding trees in terms of operations that generate them. Replacement operations on binding trees are then defined. Examples based on the lambda-c calculus [19] are sprinkled throughout to illustrate the various concepts and relate them to traditional notations. To illustrate one use of the underlying tree structure, we give alternative definitions of the set of binding trees and operations on binding trees directly in terms of tree domains and labeling functions.

We take as given countable sets of identifiers **Id**, indices **Ix**, and selectors **Sel**, and we let *id* range over **Id**, *ix* range over **Ix**, and *s* range over **Sel**. These sets need not be disjoint. They could all be equal to the set of natural numbers, but this would be needlessly parsimonious.

3.1 Binding Operators

We fix a set of binding operators **Op**. For $op \in \mathbf{Op}$, $\mathtt{bdrs}(op) \in P_\omega[\mathbf{Ix}]$ is the set of binding indices (binders), $\mathtt{args}(op) \in P_\omega[\mathbf{Sel}]$ is the set of argument selectors, and $\mathtt{brel}(op) \in P_\omega[\mathtt{bdrs} \times \mathtt{args}]$ is the binding relation. A nonbinding operator is one with `bdrs` empty. We let *op* range over **Op**.

Example. The lambda-c calculus syntax has the operators app, `lam`, and `let` specified by the following table.

name	`bdrs`	`args`	`brel`
app	$\emptyset$	$\{\mathtt{F}, \mathtt{A}\}$	$\emptyset$
`lam`	$\{\mathtt{V}\}$	$\{\mathtt{B}\}$	$\{(\mathtt{V}, \mathtt{B})\}$
`let`	$\{\mathtt{V}\}$	$\{\mathtt{A}, \mathtt{B}\}$	$\{(\mathtt{V}, \mathtt{B})\}$

In the lambda-c syntax,

$$\mathtt{lam}(\mathtt{B} : \mathrm{app}(\mathtt{F} : \mathtt{bv}([\mathtt{B}, \mathtt{F}], \mathtt{V}), \mathtt{A} : \mathtt{bv}([\mathtt{B}, \mathtt{B}, \mathtt{A}], \mathtt{V})))$$

describes a binding tree with domain $\{\mathtt{mt}, \mathtt{B}, [\mathtt{B}, \mathtt{F}], [\mathtt{B}, \mathtt{A}]\}$. The root is labeled by the operator `lam` and the node `B` is labeled by the operator app. The node $[\mathtt{B}, \mathtt{F}]$ is labeled by the bound variable with binding path $[\mathtt{B}, \mathtt{F}]$ and binding index `V`. Following the binding path, in reverse order, up the tree we see that this variable is bound at the root by the binder `V`. The node $[\mathtt{B}, \mathtt{A}]$ is labeled by the bound variable with binding path $[\mathtt{B}, \mathtt{B}, \mathtt{A}]$ and binding index `V`. Following the binding path up the tree from the bound variable node in this case leads out of the tree. Hence, this variable is bound externally one node above the tree under consideration.

3.2 Binding Paths and Binding Points

A binding point is a binding path together with a binding index. Binding paths are used to describe the node of a binding tree at which a variable is bound. The binding index describes which binder of the operator labeling the binding node is doing the binding.

We let **Path** be the set of paths, **Bp** the set of binding paths, and we let p, q range over **Path**. We will represent paths as finite sequences of selectors and binding paths as non-empty paths. In particular, using our convention that elements are also singleton sequences, a selector s represents a binding path (describing the node directly above) and any binding path can be represented in the form $[s, p]$. Representing paths as sequences of selectors gives a natural geometrical meaning to the manipulation of binding paths and structures. Since only the path length matters in determining the referenced node, we could equally well use natural numbers for paths. To capture this abstract view of paths, we let $\leq$ ($<$) be the reflexive (irreflexive) pre-ordering on sequences according to length, and let $\equiv$ be the corresponding equivalence relation.

When a binding tree is moved relative to its external binding context, external segments of binding points must be correspondingly adjusted to preserve binding relations. For example, `let` elimination, $\mathtt{let}\{x := e_1\}e_0 \mapsto (\lambda x.e_0)e_1$, is represented by

$$\mathtt{let}(\mathtt{A}:\tau_1, \mathtt{B}:\tau_0) \mapsto \mathtt{app}(\mathtt{F}:\mathtt{lam}(\mathtt{B}:\tau_0{\downarrow}(\mathtt{F},\mathtt{B})), \mathtt{A}:\tau_1),$$

where $\tau{\downarrow}(p', p)$ adjusts for motion down path p' of a structure containing τ at p. We define adjustment operations $p_1{\downarrow}(q,p)$ and $p_1{\uparrow}p$. These operations act on paths modulo $\equiv$. If p is strictly smaller than p_1, then $p_1{\downarrow}(q,p)$ inserts q before the tail with length that of p in p_1. $p_1{\uparrow}p$ deletes the edge corresponding to s from p_1 if $[s,p]$ is equivalent to a proper tail of p_1. Thus, if in some containing structure the subtree at the node τ described by p is moved down along path q, then within that subtree $p_1{\downarrow}(q,p)$ describes the same node in the transformed tree as p_1 described in the original tree. Similarly, if the edge above τ is deleted, then $p_1{\uparrow}p$ describes the same node in the transformed tree as p_1 described in the original tree.

Definition (binding path adjustment):

$$p_1{\downarrow}(q,p) \equiv \begin{cases} p_1 & \text{if } p_1 \leq p, \\ [s, p_2, q, p] & \text{if } p_1 \equiv [s, p_2, p], \end{cases}$$

$$p_1{\uparrow}p \equiv \begin{cases} p_1 & \text{if } p_1 \leq [s, p], \\ [s', p_2, p] & \text{if } p_1 \equiv [s', p_2, s, p]. \end{cases}$$

We write $p_1{\downarrow}p$ for $p_1{\downarrow}(p, \mathtt{mt})$. The adjustment operations compose and commute according to the following equations:

$$q{\downarrow}(p_0, p){\downarrow}(p_1, p) \equiv q{\downarrow}([p_0, p_1], p),$$

$$p_0{\downarrow}(q, [p, p']){\downarrow}(q', p') \equiv p_0{\downarrow}(q', p'){\downarrow}(q, [p_0, q', p']),$$

$$q{\downarrow}(s, p){\uparrow}p \equiv q.$$

3.3 Binding Trees

A binding tree is a labeled tree. Labels consist of a tag identifying the node constructor together with additional simple data. The possible tags are `FV` (free variable), `BV` (bound variable), `HOL` (hole), and `BO` (binding operator). The additional data for a `FV` tag is an identifier, for a `BV` tag is a binding point (a binding path together with a binding index), for a `HOL` tag is an identifier, and for a `BO` is an operator.

The substitution associated with a hole is represented as a family of subtrees indexed by identifiers. Thus, the edges of a binding tree are selectors and identifiers,

$$\mathbf{Edg} = \mathbf{Sel} \cup \mathbf{Id},$$

$$\mathbf{Lab} = \mathtt{FV}{:}\mathbf{Id} \cup \mathtt{BV}{:}[\mathbf{Bp} \times \mathbf{Ix}] \cup \mathtt{HOL}{:}\mathbf{Id} \cup \mathtt{BO}{:}\mathbf{Op}.$$

We let π range over $\mathbf{Edg}^*$.

The set of binding trees is the subset of $\mathit{Ltree}[\mathbf{Edg}, \mathbf{Lab}]$ generated by the labeled tree constructions `fv`, `bv`, `hol`, and `bo`. `fv` constructs a free variable node from an identifier. `bv` constructs a bound variable node from a binding point. `hol` constructs a hole node from an identifier and a finite map from identifiers to binding trees. `bo` constructs a binding operator node from a binding operator and a map associating argument selectors of the binding operator to binding trees.

Definition (binding tree constructors):

$$\mathtt{fv} = \lambda id \in \mathbf{Id}.\lambda\pi \in \{\mathtt{mt}\}.\mathtt{FV}{:}id,$$

$$\mathtt{bv} = \lambda(p, ix) \in \mathbf{Bp} \times \mathbf{Ix}.\lambda\pi \in \{\mathtt{mt}\}.\mathtt{BV}{:}(p, ix),$$

$$\mathtt{hol} = \lambda(id, \sigma) \in \mathbf{Id} \times \mathit{Fmap}[\mathbf{Id}, \mathit{Ltree}[\mathbf{Edg}, \mathbf{Lab}]].\mathit{mkLt}(\mathtt{HOL}{:}id, \sigma),$$

$$\mathtt{bo}(op, \mu) = \lambda(op, \sigma) \in \mathbf{Op} \times \mathit{Fmap}[\mathbf{Sel}, \mathit{Ltree}[\mathbf{Edg}, \mathbf{Lab}]].\mathit{mkLt}(\mathtt{BO}{:}op, \mu).$$

Definition (binding tree generation): **Bt** is the least set B closed under the:

(fv) $\quad id \in \mathbf{Id} \rightarrow \mathtt{fv}(id) \in B,$

(bv) $(p, ix) \in \mathbf{Bp} \times \mathbf{Ix} \to \mathtt{bv}(p, ix) \in B$,

(hol) $id \in \mathbf{Id} \wedge \sigma \in Fmap[\mathbf{Id}, B] \to \mathtt{hol}(id, \sigma) \in B$,

(bo) $op \in \mathbf{Op} \wedge \mu \in [\mathbf{args}(op) \to B] \to \mathtt{bo}(op, \mu) \in B$.

We let τ range over $\mathbf{Bt}$. Now we make precise the sense in which binding points describe binding relations. We define the path $\widehat{\pi}$, associated with a node π in a binding tree, to be the subsequence of selectors occurring in π (i.e., the result of erasing all identifier occurrences from π). Let π be a bound variable occurrence of τ, i.e., $\pi \in dom(\tau)$ and $\tau(\pi) = \mathtt{BV}{:}(p, ix)$ for some binding point (p, ix). If $p > \widehat{\pi}$, then we say that the variable at π is externally bound. Otherwise we can uniquely decompose π such that $\pi = [\pi_0, s, \pi_1]$ and $p \equiv [s, \widehat{\pi}_1]$. In this case, π_0 is the binding node of π in τ. Note that by the definition of binding tree there must be some operator op such that $\tau(\pi_0) = \mathtt{BO} : op$.

Example.

$$\tau = \mathtt{lam}(\mathtt{B} : \mathtt{app}(\mathtt{F} : \mathtt{bv}([\mathtt{B}, \mathtt{B}, \mathtt{F}], \mathtt{V}), \mathtt{A} : \mathtt{hol}(0, \{\mathtt{x} := \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V})\}))),$$

$$\tau' = \mathtt{lam}(\mathtt{B} : \tau).$$

In τ the variable occurrence at $[\mathtt{B}, \mathtt{F}]$ is externally bound, the subtree at $[\mathtt{B}, \mathtt{A}]$ is a hole, and the variable occurrence at $[\mathtt{B}, \mathtt{A}, \mathtt{x}]$ is bound at the root. In τ' the variable at $[\mathtt{B}, \mathtt{B}, \mathtt{F}]$ is bound at the root and the variable occurrence at $[\mathtt{B}, \mathtt{B}, \mathtt{A}, \mathtt{x}]$ is bound at the node $\mathtt{B}$. Note that in concrete notation we write $\mathtt{lam}(\mathtt{B} : \tau)$ for $\mathtt{bo}(\mathtt{lam}, \{\mathtt{B} := \tau\})$ and similarly for other operators. We will often use standard argument order and omit selectors. Thus, $\mathtt{let}(\tau_1, \tau_0)$ abbreviates $\mathtt{let}(\mathtt{A} : \tau_1, \mathtt{B} : \tau_0)$, and so on.

3.4 Operations on Binding Trees

We define operations on binding trees by structural recursion. As a first example, we define a simple binding operation $bd \in [\mathbf{Id} \times \mathbf{Bp} \times \mathbf{Ix} \times \mathbf{Bt} \to \mathbf{Bt}]$. $bd(id, p, ix, \tau)$ replaces occurrences of free variables in τ, whose identifier is id, by variables bound externally at (p, ix).

Definition (bd):

$$bd(id, p, ix, \mathtt{fv}(id)) = \mathtt{bv}(p, ix),$$

$$bd(id, p, ix, \mathtt{fv}(id')) = \mathtt{fv}(id') \qquad \text{if } id \neq id',$$

$$bd(id, p, ix, \mathtt{hol}(id, \sigma)) = \mathtt{hol}(id, \lambda x \in dom(\sigma).bd(id, p, ix, \sigma(x))),$$

$$bd(id, p, ix, \mathtt{bo}(op, \mu)) = \mathtt{bo}(op, \lambda s \in dom(\mu).bd(id, p{\downarrow}s, ix, \sigma(s))).$$

Using bd, we give a correspondence $\hookrightarrow$ from traditional notation for lambda-c expressions to binding trees. This correspondence will be formalized in section 4, where a representation of binding structures corresponding directly to traditional notation is defined.

Example.

$$x \hookrightarrow \mathtt{fv}(\mathtt{x}),$$

$$[\![\ ]\!] \hookrightarrow \mathtt{hol}(\mathtt{0}, \mathtt{mt}),$$

$$e_0 \hookrightarrow \tau_0 \wedge e_1 \hookrightarrow \tau_1 \rightarrow e_0(e_1) \hookrightarrow \mathtt{app}(\tau_0, \tau_1),$$

$$e \hookrightarrow \tau \rightarrow \lambda x.e \hookrightarrow \mathtt{lam}(bd(\mathtt{x}, \mathtt{B}, \mathtt{V}, \tau)),$$

$$e_0 \hookrightarrow \tau_0 \wedge e_1 \hookrightarrow \tau_1 \rightarrow \mathtt{let}\{x := e_0\}e_1 \hookrightarrow \mathtt{let}(\tau_0, bd(\mathtt{x}, \mathtt{B}, \mathtt{V}, \tau_1)),$$

where $[\![\ ]\!]$ represents a hole, the basic building block for expression contexts, and we adopt the convention that x is a variable with identifier $\mathtt{x}$, etc. The following specific examples illustrate the role of free and bound variables, holes, and binding operator application.

Example.

$$\begin{aligned} e_0 &= \lambda x.f(x) \hookrightarrow \tau_0, \\ &= \mathtt{lam}(bd(\mathtt{x}, \mathtt{B}, \mathtt{V}, \mathtt{app}(\mathtt{fv}(\mathtt{f}), \mathtt{fv}(\mathtt{x})))), \\ &= \mathtt{lam}(\mathtt{app}(\mathtt{fv}(\mathtt{f}), \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V}))), \\ e_1 &= \lambda x.f([\![\ ]\!]) \hookrightarrow \tau_1, \\ &= \mathtt{lam}(bd(\mathtt{x}, \mathtt{B}, \mathtt{V}, \mathtt{app}(\mathtt{fv}(\mathtt{f}), \mathtt{hol}(\mathtt{0}, \mathtt{mt})))), \\ &= \mathtt{lam}(\mathtt{app}(\mathtt{fv}(\mathtt{f}), \mathtt{hol}(\mathtt{0}, \{\mathtt{x} := ([\mathtt{B}, \mathtt{A}], \mathtt{V})\}))). \end{aligned}$$

We lift the adjustment operations on binding paths to binding trees and substitution maps as follows.

Definition ($\downarrow$):

$$\mathtt{fv}(id){\downarrow}(q, p) = \mathtt{fv}(id),$$

$$\mathtt{bv}(p', ix){\downarrow}(q, p) = \mathtt{bv}(p'{\downarrow}(q, p), ix),$$

$$\mathtt{hol}(id, \sigma){\downarrow}(q, p) = \mathtt{hol}(id, \sigma{\downarrow}(q, p)),$$

$$\mathtt{bo}(op, \mu){\downarrow}(q, p) = \mathtt{bo}(op, \lambda s \in dom(\mu).\mu(s){\downarrow}(q, [p, s])),$$

$$\sigma{\downarrow}(q, p) = \lambda i \in dom(\sigma).\sigma(i){\downarrow}(q, p).$$

Definition ($\uparrow$):

$$\mathtt{fv}(id){\uparrow}p = \mathtt{fv}(id),$$

$$\mathtt{bv}(p', ix)\uparrow p = \mathtt{bv}(p'\uparrow p, ix),$$

$$\mathtt{hol}(id, \sigma)\uparrow p = \mathtt{hol}(id, \sigma\uparrow p),$$

$$\mathtt{bo}(op, \mu)\uparrow p = \mathtt{bo}(op, \lambda s \in dom(\mu).\mu(s)\uparrow[p, s]),$$

$$\sigma\uparrow p = \lambda i \in dom(\sigma).\sigma(i)\uparrow p.$$

We write $\tau\downarrow q$ for $\tau\downarrow(q, \mathtt{mt})$. Adjustments lifted to trees and substitution maps compose and commute according to the same laws as for adjustment of binding paths.

Three fundamental replacement operations on binding trees are substitution for free variables, unbinding of bound variables, and filling of holes. Substitution, $sb \in [Fmap[\mathbf{Id}, \mathbf{Bt}] \times \mathbf{Bt} \to \mathbf{Bt}]$, replaces free variables using a map from free variable identifiers to binding trees. Substitution preserves binding relative to some common external context for the substituents and the substitutee. At a hole, the substitution being applied is composed (on the left) with the substitution associated with the hole.

Definition (sb):

$$sb(\sigma, \mathtt{fv}(id)) = \begin{cases} \sigma(id) & \text{if } id \in dom(\sigma), \\ \mathtt{fv}(id)) & \text{if } id \notin dom(\sigma), \end{cases}$$

$$sb(\sigma, \mathtt{bv}(p, ix)) = \mathtt{bv}(p, ix),$$

$$sb(\sigma, \mathtt{hol}(id, \sigma')) = \mathtt{hol}(id, \sigma \lhd \lambda i \in dom(\sigma').sb(\sigma, \sigma'(i))),$$

$$sb(\sigma, \mathtt{bo}(op, \mu)) = \mathtt{bo}(op, \lambda s \in dom(\mu).sb(\sigma\downarrow s, \mu(s)).$$

Unbinding, $ub \in [Fmap[\mathbf{Bp} \times \mathbf{Ix}, \mathbf{Bt}] \times \mathbf{Bt} \to \mathbf{Bt}]$, replaces externally bound variables according to an unbinding map from binding points to binding structures. At a hole, the associated substitution is unbound pointwise.

Definition (ub):

$$ub(\zeta, \mathtt{fv}(id)) = \mathtt{fv}(id),$$

$$ub(\zeta, \mathtt{bv}(p, ix)) \begin{cases} \zeta(p, ix) & \text{if } (p, ix) \in dom(\zeta), \\ \mathtt{bv}(p, ix) & \text{if } (p, ix) \notin dom(\zeta), \end{cases}$$

$$ub(\zeta, \mathtt{hol}(id, \sigma)) = \mathtt{hol}(id, \lambda i \in dom(\sigma).ub(\zeta, \sigma(i))),$$

$$ub(\zeta, \mathtt{bo}(op, \mu)) = \mathtt{bo}(op, \lambda s \in dom(\mu).ub(\zeta\Downarrow s, \mu(s)),$$

where

$$\mathtt{mt}\Downarrow(q, p) = \mathtt{mt},$$

$$\zeta\{(p', ix) := \tau\}\Downarrow(q, p) = (\zeta\Downarrow(q, p))\{(p'\downarrow(q, p), ix) := (\tau\downarrow(q, p))\}.$$

Filling, $\mathit{fil} \in [\mathit{Fmap}[\mathbf{Id}, \mathbf{Bt}] \times \mathbf{Bt} \to \mathbf{Bt}]$, replaces holes using a map from hole identifiers to binding trees. Filling preserves binding relations except at holes in the domain of the filling map. At such a hole, the associated substitution is filled (pointwise) using the filling map, and the resulting substitution is applied to the filler obtained by applying the filling map to the hole identifier to obtain the hole replacement.

Definition (*fil*):

$$\mathit{fil}(\varphi, \mathtt{fv}(id)) = \mathtt{fv}(id),$$

$$\mathit{fil}(\varphi, \mathtt{bv}(p, ix)) = \mathtt{bv}(p, ix),$$

$$\mathit{fil}(\varphi, \mathtt{hol}(id, \sigma')) =$$

$$\begin{cases} sb(\lambda id' \in dom(\sigma').\mathit{fil}(\varphi, \sigma'(id')), \varphi(id)) & \text{if } id \in dom(\varphi), \\ \mathtt{hol}(id, \lambda id' \in dom(\sigma').\mathit{fil}(\varphi, \sigma'(id'))) & \text{if } id \notin dom(\varphi), \end{cases}$$

$$\mathit{fil}(\varphi, \mathtt{bo}(op, \mu)) = \mathtt{bo}(op, \lambda s \in dom(\mu).\mathit{fil}(\varphi{\downarrow}s, \mu(s))).$$

Now we continue the lambda-c example to illustrate application of the adjustment, substitution, unbinding, and filling operations.

Example.

$$e_1[\![x]\!] \hookrightarrow \mathit{fil}(\{\mathtt{0} := \mathtt{fv(x)}\}, \tau_1),$$

$$= \mathtt{lam}(\mathtt{app}(\mathtt{fv(f)}, \mathit{fil}(\{\mathtt{0} := \mathtt{fv(x)}\}, \mathtt{hol}(\mathtt{0}, \{\mathtt{x} := ([\mathtt{B}, \mathtt{A}], \mathtt{V})\})))),$$

$$= \mathtt{lam}(\mathtt{app}(\mathtt{fv(f)}, sb(\{\mathtt{x} := ([\mathtt{B}, \mathtt{A}], \mathtt{V})\}, \mathtt{fv(x)}))),$$

$$= \tau_0,$$

$$sb(\{\mathtt{f} := \tau\}, \tau_0) = \mathtt{lam}(\mathtt{app}((\tau{\downarrow}\mathtt{B}){\downarrow}\mathtt{F}), \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V})),$$

$$= \mathtt{lam}(\mathtt{app}((\tau{\downarrow}[\mathtt{B}, \mathtt{F}]), \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V}))),$$

$$= \mathtt{lam}(\mathtt{app}((\tau{\downarrow}[\mathtt{B}, \mathtt{F}]), \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V}))).$$

The beta rule can be expressed in terms of binding trees as follows:

$$\mathtt{app}(\mathtt{lam}(\tau_b){\downarrow}\mathtt{F}, \tau_a) \xrightarrow{\beta} ub(\{(\mathtt{B}, \mathtt{V}) := \tau_a\}, \tau_b){\uparrow}\mathtt{B}.$$

The adjustments in this rule reflect that fact that external bindings in the subtrees τ_b, τ_a should refer to the same external points before and after application of the transformation. Specifically, two edges are deleted from the path leading to τ_b and one edge is removed from the path leading to τ_a. One of the edges deleted from the path leading to τ_b is accounted for by the $\downarrow$F. The other must be accounted for after unbinding has removed binding points referring to the `lam` node. The result of unbinding is thus a binding tree with an extra edge in the path leading to it. Deletion of this edge is accounted for by applying $\uparrow\mathtt{B}$ to the result of unbinding.

Example. As an example of beta reduction, using the tree τ_0 constructed above and any other tree τ, we have

$$\mathbf{app}(\tau_0, \tau) \xrightarrow{\beta} \mathbf{app}(\mathtt{fv}(\mathtt{f}), \tau),$$

since, using the definitions of adjustment we have

$$\begin{aligned}&ub(\{(\mathtt{B}, \mathtt{V}) := \tau\}, \mathbf{app}(\mathtt{fv}(\mathtt{f}), \mathtt{bv}([\mathtt{B}, \mathtt{A}], \mathtt{V})))\uparrow\mathtt{B}\\ &\quad = \mathbf{app}(\mathtt{fv}(\mathtt{f}), \tau{\downarrow}\mathtt{A})\uparrow\mathtt{B} = \mathbf{app}(\mathtt{fv}(\mathtt{f}), \tau{\downarrow}\mathtt{A}\uparrow\mathtt{B}) = \mathbf{app}(\mathtt{fv}(\mathtt{f}), \tau).\end{aligned}$$

An important consequence of our representation of holes using associated substitutions is that applications of filling maps can be uniformly moved across substitutions.

Theorem (fil.sb):

$$\mathit{fil}(\varphi, sb(\sigma, \tau)) = sb(\lambda id \in dom(\sigma).\mathit{fil}(\varphi, \sigma(id)), \mathit{fil}(\varphi, \tau)).$$

3.5 Working with a Global View

We conclude this section by illustrating the use of tree domains and labeling functions to directly describe the class of binding trees and operations on binding trees.

Theorem (Bt): The set of binding trees **Bt** is characterized as follows.

$$\begin{aligned}&\tau \in \mathbf{Bt} \leftrightarrow \tau \in \mathit{Ltree}[\mathbf{Edg}, \mathbf{Lab}] \wedge\\ &\quad (\forall \pi \in dom(\tau))(\\ &\qquad (\exists id)(\tau(\pi) = \mathtt{FV}\!:id \wedge es(\tau, \pi) = \{\ \}) \vee\\ &\qquad (\exists p, ix)(\tau(\pi) = \mathtt{BV}\!:(p, ix) \wedge es(\tau, \pi) = \{\ \}) \vee\\ &\qquad (\exists id)(\tau(\pi) = \mathtt{HOL}\!:id \wedge es(\tau, \pi) \in P_\omega[\mathbf{Id}]) \vee\\ &\qquad (\exists op)(\tau(\pi) = \mathtt{BO}\!:op \wedge es(\tau, \pi) = \mathbf{args}(op))).\end{aligned}$$

Theorem (adj): Adjustment $\downarrow$ on binding structure trees has the following global characterization.

$$\begin{aligned}&\tau{\downarrow}(p_0, p_1) = \tau' \leftrightarrow dom(\tau) = dom(\tau') \wedge\\ &\qquad (\forall \pi \in dom(\tau))(\tau'(\pi) = \tau(\pi){\downarrow}(p_0, [p_1, \widehat{\pi}])),\end{aligned}$$

where

$$\mathtt{BV}:(p', ix)\!\downarrow\!(q,p) = \mathtt{BV}:(p'\!\downarrow\!(q,p), ix),$$

$$\mathtt{FV}: id\!\downarrow\!(q,p) = \mathtt{FV}: id,$$

$$\mathtt{HOL}: id\!\downarrow\!(q,p) = \mathtt{HOL}: id,$$

$$\mathtt{BO}: op\!\downarrow\!(q,p) = \mathtt{BO}: op.$$

Our final example is the specification of an operation that applies a top-level conversion r to a set of independent occurrences P in a binding tree. Let r be a top-level reduction; for example, the beta rule. For simplicity, we assume r is the identity on trees not of the correct form. We specify $pa(r, P, \tau)$, the parallel application of r at sites P in τ, as follows. The precondition for applicability, $okpa(P, \tau)$, says that P is a subset of the domain of τ and that no node in P is is a proper prefix of any other node of P. $pa(r, P, \tau)$ agrees with τ on $dom(\tau)/P$, the nodes in the domain of τ not below any node of P, and the subtree of $pa(r, P, \tau)$ at each node in P is the result of applying r to the subtree of τ at that node,

$$okpa(P, \tau) \leftrightarrow$$

$$P \subset dom(\tau) \wedge (\forall \pi, \pi' \in P)(\forall \pi_0)(\pi' = [\pi, \pi_0] \to \pi_0 = \mathtt{mt}),$$

$$dom(\tau)/P = \{\pi \in dom(\tau) \mid \neg(\exists \pi' \in P, \pi_0)\pi = [\pi'.\pi_0]\},$$

$$okpa(P, \tau) \to \tau' = pa(r, P, \tau) \leftrightarrow$$

$$dom(\tau') = dom(\tau)/P \cup \{[\pi, \pi'] \mid \pi \in P \wedge \pi' \in dom(r(\tau \wr \pi))\} \wedge$$

$$(\forall \pi \in dom(\tau)/P)\tau(\pi) = \tau'(\pi) \wedge (\forall \pi \in P)\tau' \wr \pi = r(\tau \wr \pi).$$

4 Binding via Bound Variable Names

In this section, we define the set **Asb** of abstract structures with binding (ASBs) and the set **Ncx** of naming contexts. ASBs are a representation of binding structures that corresponds to traditional notation. Thus, binding relations are determined by variable names, and whether or not a variable is bound depends on the context in which it occurs. Naming contexts permit us to distinguish between free and externally bound variables. We define a map from ASBs with naming contexts to binding tree structures. Two structures map to the same binding tree just if they only differ by renaming of bound variables. When descending into a structure at a binding node, a choice of names for bound variables must be fixed. Naming contexts provide a means of recording naming choices while preserving the ability to rename bound variables. We conclude by showing how operations on binding structures can be specified by relations on ASBs with associated naming contexts.

4.1 ASBs

ASBs are either variables, holes, or binding operator applications. Variables have names. A hole has a name and a substitution mapping identifiers to ASBs. A binding operator application has an operator, a naming map from binders to variable names, and a map from argument selectors to ASBs. The naming map, together with the operator binding relation, determine the names of the variables bound by each binder in the argument substructures. In order that each bound variable have a uniquely determined binding point, we require that the naming map be injective.

Definition (Asb): The set **Asb** is generated by the operations `var`, `hol`, `bo` according to the following equation.

$$\begin{aligned}\mathbf{Asb} = {} & \mathtt{var}(\mathbf{Id}) \cup \mathtt{hol}(\mathbf{Id}, \mathit{Fmap}[\mathbf{Id}, \mathbf{Asb}]) \cup \\ & \mathtt{bo}(op \in \mathbf{Op}, [\mathtt{bdrs}(op) \to \mathbf{Id}]_{\mathrm{inj}}, [\mathtt{args}(op) \to \mathbf{Asb}]).\end{aligned}$$

A naming context is an injective finite map from identifiers to (external) binding points,

$$\mathbf{Ncx} = \mathit{Fmap}[\mathbf{Id}, \mathbf{Bp} \times \mathbf{Ix}]_{\mathrm{inj}}.$$

We let a range over **Asb** and ξ range over **Ncx**.

a2b maps ASBs to binding trees. It is extended to ASBs with associated naming contexts by first carrying out the ASB conversion and then applying the substitution obtained by composing the bound variable constructor with the naming context. At a binding operator op, the binding substitution $bds(op, s, \eta)$ transforms binding in the substructure along s described by the naming map η into corresponding binding points. $bds(op, s, \eta)$ maps binder names $\eta(ix)$ to bound variables with binding point (s, ix) if (ix, s) is in the binding relation of op.

Definition (induced binding substitution):

$$bds(op, s, \eta) = \{id := \mathtt{bv}(s, ix) \mid (ix, s) \in \mathtt{brel}(op) \wedge \eta(ix) = id\}.$$

The map *a2b* is defined as follows.

Definition (*a2b*):

$$a2b(\mathtt{var}(id)) = \mathtt{fv}(id),$$

$$a2b(\mathtt{hol}(id, \sigma)) = \mathtt{hol}(id, \lambda x \in dom(\sigma).a2b(\sigma(x))),$$

$$a2b(\mathtt{bo}(op, \eta, \mu)) = \mathtt{bo}(op, \lambda s \in dom(\mu).sb(bds(op, s, \eta), a2b(\mu(s)))),$$
$$a2b(a, \xi) = sb(\mathtt{bv} \circ \xi, a2b(a)).$$

The substitution associated with holes decouples trapping and binding and allows for renaming of bound variables even when there is a hole in their scope. This contrasts with the usual formulation of contexts as expressions with undecorated holes. For example, the structure obtained by lambda-binding the variable named x in a hole with empty substitution $\mathtt{lam(x, hol(0, mt))}$ can be renamed to $\mathtt{lam(y, hol(0, \{x := var(y)\}))}$ without changing its meaning, while the context $\mathtt{lam(x, hol(0, mt))}$ is clearly different from the $\mathtt{lam(y, hol(0, mt))}$. The easiest way to see this is to look at the binding trees to which these expressions map:

$$a2b(\mathtt{lam(x, hol(0, mt))}) = \mathtt{lam(hol(0, \{x := bv(B, V)\}))},$$
$$a2b(\mathtt{lam(y, hol(0, \{x := var(y)\}))}) = \mathtt{lam(hol(0, \{x := bv(B, V)\}))},$$
$$a2b(\mathtt{lam(y, hol(0, mt))}) = \mathtt{lam(hol(0, \{y := bv(B, V)\}))}.$$

We say that the binding point (p, ix) is admissible at π in τ if it is externally bound $(\widehat{\pi} < p)$ or if it is bound at a node π_0 labeled by a binding operator op such that (ix, s) is in the binding relation of op and π is below $[\pi_0, s]$. **aBt** is the set of binding trees with only admissible bindings.
Definition (aBt):

$$\begin{aligned}
&\tau \in \mathbf{aBt} \;\leftrightarrow\; \tau \in \mathbf{Bt} \,\wedge \\
&\quad (\forall \pi \in dom(\tau))(\forall p, ix)(\tau(\pi) = \mathtt{BV}{:}(p, ix) \;\rightarrow\; \widehat{\pi} < p \,\vee \\
&\qquad (\exists op, \pi_0, s, \pi_1)(\pi = [\pi_0, s, \pi_1] \,\wedge \\
&\qquad\qquad \tau(\pi_0) = \mathtt{BO}{:}\,op \;\wedge\; p = [s, \widehat{\pi}_1] \;\wedge\; (ix, s) \in \mathtt{brel}(op))).
\end{aligned}$$

The binding trees that arise in practice are generally admissible. In particular, the binding trees with only admissible bindings are exactly those that correspond to binding structures represented using named bound variables.
Theorem (asb.admis): **aBt** is the image of $\mathbf{Asb} \times \mathbf{Ncx}$ under the map $a2b$.

As we saw in the example above, the map from ASBs with associated naming contexts to binding trees is not 1–1. The equivalence relation $\stackrel{s}{\simeq}$ (structural similarity) induced by this map corresponds to the usual notion of equivalence up to renaming of bound variables. To define the similarity relation, we use the operation, $xncx \in \mathbf{Ncx} \times \mathbf{Op} \times \mathbf{Sel} \times Fmap[\mathbf{Ix}, \mathbf{Id}] \rightarrow \mathbf{Ncx}$. $xncx(\xi, op, s, \eta)$ adjusts the naming context ξ to preserve external binding relations and extends it to record the choice of bound variable names when moving from an ASB constructed with operator op using naming map η to the substructure selected by s.

Definition ($xncx$):

$$xncx(\xi, op, s, \eta) = (\xi{\downarrow}s) \lhd bds(op, s, \eta),$$

where

$$(p_1, ix){\downarrow}(p, q) = (p_1{\downarrow}(p, q), ix),$$

$$\xi{\downarrow}(p, q) = \lambda id \in dom(\xi).\xi(id){\downarrow}(p, q).$$

Definition (structural similarity): Structural similarity is the least relation, $a; \xi \overset{s}{\simeq} a'; \xi'$, on ASBs with naming contexts such that the following hold:

$$id \notin (dom(\xi) \cup dom(\xi')) \rightarrow \mathtt{var}(id); \xi \overset{s}{\simeq} \mathtt{var}(id); \xi',$$

$$id \in dom(\xi) \land id' \in dom(\xi') \land \xi(id) = \xi'(id')$$

$$\rightarrow \mathtt{var}(id); \xi \overset{s}{\simeq} \mathtt{var}(id'); \xi',$$

$$1_\xi \lhd \sigma; \xi \overset{s}{\simeq} 1'_\xi \lhd \sigma'; \xi' \rightarrow \mathtt{hol}(id, \sigma); \xi \overset{s}{\simeq} \mathtt{hol}(id, \sigma'); \xi',$$

$$(\forall s \in \mathtt{args}(op))\mu(s); xncx(\xi, op, s, \eta) \overset{s}{\simeq} \mu'(s); xncx(\xi', op, s, \eta')$$

$$\rightarrow \mathtt{bo}(op, \eta, \mu); \xi \overset{s}{\simeq} \mathtt{bo}(op, \eta', \mu'); \xi',$$

$$\sigma; \xi \overset{s}{\simeq} \sigma'; \xi' \leftrightarrow$$

$$dom(\sigma) = dom(\sigma') \land (\forall id \in dom(\sigma))(\sigma(id); \xi \overset{s}{\simeq} \sigma'(id); \xi'),$$

where $1_\xi = \lambda id \in dom(\xi).\mathtt{var}(id)$ is the identity variable substitution with domain that of ξ.

Theorem (a2b.sim): $a; \xi \overset{s}{\simeq} a'; \xi'$ just if $a2b(a, \xi) = a2b(a', \xi')$.

Corollary (a2b): **aBt** is isomorphic to **Asb** $\times$ **Ncx** modulo $\overset{s}{\simeq}$.

We conclude by giving a specification of substitution as a relation on ASBs with associated naming contexts. The first clause in the definition says that variables with identifier in the domain of the naming context are not free in the associated ASB. The key clause is that for binding operators. It expresses exactly the hygiene conditions required for renaming prior to substitution. $\mathtt{bo}(op, \eta_1, \mu_1); \xi$ is a renaming of $\mathtt{bo}(op, \eta, \mu); \xi$ such that no variables bound according to the naming η_1 trap free variables of σ ($\mathrm{Rng}(\eta_1) \cap \mathrm{FV}(\sigma) = \emptyset$), and no externally bound variables are trapped by the naming η_1 ($\mathrm{Rng}(\eta_1) \cap dom(\xi) = \emptyset$).

Definition ($isSb$): The substitution relation $isSb$ is the least relation on $Fmap[\mathbf{Id}, \mathbf{Asb}] \times \mathbf{Asb} \times \mathbf{Ncx} \times \mathbf{Asb} \times \mathbf{Ncx}$ such that

$$
\begin{aligned}
& a = \mathtt{var}(id) \ \wedge \ id \in dom(\xi) \ \wedge \ a;\xi \overset{s}{\simeq} a';\xi' \ \rightarrow \ isSb(\sigma, a, \xi, a', \xi'), \\
& id \in (dom(\sigma) - dom(\xi)) \ \wedge \ \sigma(id);\xi \overset{s}{\simeq} a';\xi' \ \rightarrow \ isSb(\sigma, \mathtt{var}(id), \xi, a', \xi'), \\
& id \notin (dom(\sigma) \cup dom(\xi) \cup dom(\xi')) \wedge \ \rightarrow \ isSb(\sigma, \mathtt{var}(id), \xi, \mathtt{var}(id), \xi'), \\
& \mu' = 1_{\xi'} \lhd \sigma' \wedge \\
& \qquad (\forall x \in dom(\sigma_1)) isSb(\sigma, \sigma_1(x), \xi, \mu'(x), \xi') \wedge \\
& \qquad (\forall x \in dom(\xi) - dom(\sigma_1)) isSb(\sigma, \mathtt{var}(x), \xi, \mu'(x), \xi') \wedge \\
& \qquad (\forall x \in dom(\sigma) - (dom(\xi) \cup dom(\sigma_1)))(\sigma(x);\xi \overset{s}{\simeq} \mu'(x);\xi') \\
& \qquad\qquad \rightarrow \ isSb(\sigma, \mathtt{hol}(id, \sigma_1), \xi, \mathtt{hol}(id, \sigma'), \xi'), \\
& \mathtt{bo}(op, \eta, \mu);\xi \overset{s}{\simeq} \mathtt{bo}(op, \eta_1, \mu_1);\xi \wedge \\
& \quad \mathbf{Rng}(\eta_1) \cap (\mathbf{FV}(\sigma) \cup dom(\xi)) = \emptyset \wedge \\
& \quad (\forall s \in \mathbf{args}(op)) \\
& \qquad isSb(\sigma, \mu_1(s); xncx(\xi, op, s, \eta_1), \mu'(s); xncx(\xi', op, s, \eta')) \\
& \quad \rightarrow \ isSb(\sigma, \mathtt{bo}(op, \eta, \mu);\xi, \mathtt{bo}(op, \eta', \mu');\xi'),
\end{aligned}
$$

where FV computes the free variables and is defined most easily by mapping to binding trees,

$$
\begin{aligned}
& \mathrm{FV}(a) = \{id \in \mathbf{Id} \mid (\exists \pi \in dom(a2b(a))))(a2b(a)(\pi) = \mathtt{FV}: id)\}, \\
& \mathrm{FV}(\sigma) = \bigcup_{id \in dom(\sigma)} \mathrm{FV}(\sigma(id)).
\end{aligned}
$$

Theorem (a2b.sb):

$$
\begin{aligned}
& isSb(\sigma, a, \xi, a', \xi') \ \leftrightarrow \\
& \quad sb(\lambda id \in dom(\sigma).a2b(\sigma(id), \xi), a2b(a, \xi)) = a2b(a', \xi').
\end{aligned}
$$

5 Related Work

Related work includes the de Bruijn notation, higher-order abstract syntax, and explicit substitution theories. De Bruijn notation [5] was designed to simplify the implementation of Automath, which is based on the typed lambda calculus. An analog to naming contexts was used for translation between external and internal notations. This notation has very clean algebraic properties and is well-suited to mechanization. Our theory generalizes the de Bruijn notation to allow for generalized binding operations, holes, and for the presence of both free and bound variables.

In NuPrl (cf. [12]), the arity of an operator θ is a tuple of numbers $<k_1, \ldots, k_n>$. n is the number of arguments, and ith component indicates the number of variables bound in the ith argument. Well-formed expressions are obtained by applying the operator to a list of binding terms, $\theta(\ldots, (x_1, \ldots, x_{k_i})e_i, \ldots)$. This is analogous to our use of the binding substitution in the map from ASBs to binding trees.

In the higher-order approach, expressions are represented as terms of a typed lambda calculus, object variables are represented as lambda-variables, all binding is reduced to lambda-binding, and substitution is represented as lambda-application. The LF system [11] uses lambda calculus with dependent types to represent syntax and rules in defining formal systems (logics). Pfenning [20] introduces higher-order abstract syntax based on a polymorphic typed lambda calculus. This work generalizes ideas of Huet and Lang [13] for specifying program transformations using high-order notation. Higher-order notations provide elegant solutions to many problems, such as expressing program transformation schemes, defining logical inference rules, etc. Variables are not object-level entities, and hygiene is taken care of by the underlying lambda calculus. Higher-order notation does not give access to actual structure; thus, it is difficult to specify conditions involving properties such as the number of occurrences of a free variable. Specific expression contexts can be represented as higher-order functions, but classes of contexts are a problem (cf. Mason [16] for a discussion of the problem of representing Hoare's Logic).

Theories of explicit substitutions have been a recent focus of attention (cf. [1, 7]). The objective is to give an elementary account of lambda-calculi, in particular beta-reduction, that can be directly implemented. The problem is that substitutions appear as meta-level objects rather than theory level objects in traditional formulations. To solve this problem, substitutions are made explicit and can be manipulated at the theory level. In [1] an equational theory with a Church–Rosser set of rewrite rules is presented. A de Bruijn-style notation is used and their substitutions correspond to a composition of our adjustment and unbinding operations. If meta-variables for expressions (holes) are allowed, normal form terms include terms corresponding to holes decorated with substitutions. This aspect is not pursued, and the hole-filling operation is not treated.

Acknowledgments

The author would like to thank Ian Mason for carefully reading earlier versions of this paper, and our group seminar members for many fruitful discussions during the course of the research. Special thanks are due to John McCarthy for laying the foundation for research in mathematical theory of computation and for providing many key insights into the nature of the problems and their solutions.

This research was partially supported by DARPA contract N00039-84-C-0211 and by NSF grants CCR-8718605 and CCR-8917606.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. Technical Report Rapports de Recherche No. 777, INRIA, 1987.

[3] M. Broy, M. Wirsing, and P. Pepper. On the Algebraic Definition of Programming Languages. *ACM Transactions on Programming Languages and Systems*, 9(1), 1987.

[4] Pavel Curtis. Algorithms: the PCL Code Walker. *Lisp Pointers*, 3(1):50–61, 1990.

[5] N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[7] J. Field. On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15, 1990.

[8] Jean H. Gallier and Wayne Snyder. Designing Unification Procedures Using Transformations: A Survey. *ETACS Bulletin*, 40:273–326, 1990.

[9] J. A. Goguen and J. Meseguer. Initiality, Induction, and Computability. In M. Nivat and J. C. Reynolds, editors, *Applications of Algebra to Language Definitions and Compilation.* Cambridge University Press, 1983.

[10] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* Ph. D. Thesis, University of Toronto, 1975.

[11] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE, 1987.

[12] Douglas J. Howe. Equality in Lazy Computation Systems. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE, 1989.

[13] Gérard Huet and Bernard Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11:31–55, 1987.

[14] I. A. Mason and C. L. Talcott. Inferring the equivalence of (first-order) functional programs that mutate data. *Theoretical Computer Science*, to appear, 199?

[15] I. A. Mason and C. L. Talcott. Reasoning about Programs with Effects. In *Programming Language Implementation and Logic Programming, PLILP'90*, volume 456 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1990.

[16] I.A. Mason. Hoare's Logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987.

[17] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Comm. ACM*, 3:184–195, 1960.

[18] J. McCarthy. Towards a Mathematical Science of Computation. In C. M. Popplewell, editor, *Information Processing: Proceedings of the IFIP Congress 62*, pages 21–28. North-Holland, Amsterdam, 1963.

[19] E. Moggi. Computational Lambda-Calculus and Monads. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE, 1989.

[20] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988.

[21] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *Proceedings of th ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48. ACM, 1984.

[22] Carolyn L. Talcott. Towards a Theory of Binding Structures. In *Second International Conference on Algebraic Methodology and Software Technology, AMAST*, 1991.