October 1991

Logical and Computational Aspects of Programming With Sets/Bags/Lists

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Ramesh Subrahmanyam
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-29.

Logical and Computational Aspects of Programming With Sets/Bags/Lists

Abstract

We study issues that arise in programming with primitive recursion over non-free datatypes such as lists, bags and sets. Programs written in this style can lack a meaning in the sense that their outputs may be sensitive to the choice of input expression. We are, thus, naturally led to a set-theoretic denotational semantics with partial functions. We set up a logic for reasoning about the definedness of terms and a deterministic and terminating evaluator. The logic is shown to be sound in the model, and its recursion free fragment is shown to be complete for proving definedness of recursion free programs. The logic is then shown to be as strong as the evaluator, and this implies that the evaluator is compatible with the provable equivalence between different set (or bag, or list) expressions. Oftentimes, the same non-free datatype may have different presentations, and it is not clear *a priori* whether programming and reasoning with the two presentations are equivalent. We formulate these questions, precisely, in the context of alternative presentations of the list, bag, and set datatypes and study some aspects of these questions. In particular, we establish back-and-forth translations between the two presentations, from which it follows that they are equally expressive, and prove results relating proofs of program properties, in the two presentations.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-29.

Logical and Computational Aspects Of Programming With Sets/Bags/Lists

MS-CIS-91-29
LOGIC & COMPUTATION 31

Val Breazu-Tannen
Ramesh Subrahmanyam

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Revised
October 1991

This paper has appeared in the *Proceedings of ICALP '91*

Logical and Computational Aspects of Programming with Sets/Bags/Lists [1]

Val Breazu-Tannen *Ramesh Subrahmanyam*

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA
E-mail: val@cis.upenn.edu, ramesh@saul.cis.upenn.edu

Abstract. We study issues that arise in programming with primitive recursion over non-free datatypes such as lists, bags and sets. Programs written in this style can lack a meaning in the sense that their outputs may be sensitive to the choice of input expression. We are, thus, naturally lead to a set-theoretic denotational semantics with partial functions. We set up a logic for reasoning about the definedness of terms and a deterministic and terminating evaluator. The logic is shown to be sound in the model, and its recursion free fragment is shown to be complete for proving definedness of recursion free programs. The logic is then shown to be as strong as the evaluator, and this implies that the evaluator is compatible with the provable equivalence between different set (or bag, or list) expressions . Oftentimes, the same non-free datatype may have different presentations, and it is not clear *a priori* whether programming and reasoning with the two presentations are equivalent. We formulate these questions, precisely, in the context of alternative presentations of the list, bag, and set datatypes and study some aspects of these questions. In particular, we establish back-and-forth translations between the two presentations, from which it follows that they are equally expressive, and prove results relating proofs of program properties, in the two presentations.

1 Introduction

1.1 Motivation

The topic of this paper is programming by structural recursion on datatype presentations whose constructors satisfy equational constraints. Sets, bags (*a.k.a.* multisets) and lists with the "append" presentation, are our primary examples, and we use them in the exposition of the technical results. It should be clear, however, that this work fits in a more general paradigm.

Our motivation came from recent object-oriented and database programming language designs. In the relational and in the complex object data models, most of the programming is done with sets, sometimes with bags. Of great interest is to design database programming languages whose type systems directly represent the underlying data models (see [Atkinson& Buneman 1987, Ohori et al., 1989] and references therein). We are therefore led to the study of languages with set datatypes, as in [Ohori et al., 1989].

How do we program with sets? Past experience with relational algebra [Codd 1970] suggests that an applicative style with a well-chosen collection of primitives is remarkably expressive. Interestingly, programming with lists works out the same way [Bird & Wadler 1988, Backus 1978]. Moreover, for lists, most programs, including the interesting primitives, can be expressed using definitions by *structural recursion*:

[1]This is a slightly revised version of the paper that has appeared in the Proceedings of ICALP'91.The authors are partially supported by ONR Grant NOOO14-88-K-0634, by NSF Grant CCR-90-57570, and by an IBM Graduate Fellowship.

```
let f(Nil)        = N
 |  f(Cons(x,l)) = C(x,f(l))
in ... f ... f ...
```

(Sometimes we need *generalized* structural recursion, where the second clause takes the form

`f(Cons(x,l)) = C(x,f(l),l)`.)

Part of the semantic justification for this programming construct is that (the meaning of) *Nil* and *Cons* *generate* all finite lists. Turning to sets, we note that the empty set, the singleton sets, and set union generate all finite sets. By analogy, here is an attempt to program cardinality in this style:

```
let card(Empty)          = 0
 |  card(SngSet(x))      = 1
 |  card(Union(s1,s2)) = card(s1) + card(s2)
in card(Union(SngSet(A),Union(Empty,SngSet(B))))
```

In a simple operational semantics, for example that of SML, this evaluates to the desired `2`. Unfortunately,

```
card(Union(SngSet(A),Union(Empty,Union(SngSet(A),SngSet(B)))))
```

evaluates to `3`, even though we have a constructor representation of the same set. Perhaps the evaluator could be hacked to transform set representations to ones without repetitions, and thus give a correct answer here? Leaving aside that this seems to require run-time checks of equalities which are not computable, let us argue that hacks that would accommodate programs like `card` would yield a fundamentally unsafe language. Consider the following chain of equalities, in which each step is done according to some axiom that we expect to be true:

```
1 = card(SngSet(x)) = card(Union(SngSet(x),SngSet(x))) =
  = card(SngSet(x)) + card(SngSet(x)) = 1 + 1 = 2
```

We see that reasoning about programs becomes *inconsistent*, and, as a corollary, there is no reasonable denotational semantics for such hacks.

To understand this problem, we must pursue the analogy with lists beyond the surface. The finite lists over a given set form, with cons and nil, an *initial* object in an appropriate category, and the meaning of a function defined by structural recursion is precisely the (unique) morphism that initiality postulates. Note that there are no constraints on cons and nil, and therefore no constraints on the operations that correspond to them in the target of the recursion. We will look for a similar *adjointness (universality)* property as the semantic basis for definitions by structural recursion on sets. Fix a set X, and consider $\mathcal{P}_{fin}(X)$, the set of all finite subsets of X. It's not hard to see that $\mathcal{P}_{fin}(X)$ with union and the empty set is the join-semilattice-with-least-element freely generated by X via the injection of generators given by the singleton set constructor (the morphisms are the join-and-least-element-preserving maps). This is better said in purely algebraic terms: $\mathcal{P}_{fin}(X)$ with union and the empty set is the commutative-idempotent monoid freely generated by X via the singleton set mapping and with respect to monoid homomorphisms.

Now the source of the problem we encountered above is clear: $(\mathbb{N}, +, 0)$ is a commutative monoid but addition is not idempotent. Thus, there is no reason to expect a (commutative idempotent) monoid homomorphism from $(\mathcal{P}_{fin}(X), \cup, \emptyset)$ to $(\mathbb{N}, +, 0)$ and in fact no such exists (or else 1=2). [2]

Such problems do not arise for list programming with *Nil* and *Cons* because they are *not* equationally constrained. But similar problems would arise for list programming with *Nil* , *SngList* , and *App* since the operations that correspond to *Nil* and *App* must form a monoid. In general, choosing constructors whose intended semantics has equational constraints gives rise, at a minimum, to the following issues:

- Datatype values may have more than one representation as constructor expressions in the language. One must prove that the evaluator is *compatible* with constructor expression equivalence (takes equivalent inputs to equivalent outputs).
- Not all programs that parse and typecheck have a meaning: for that, each of the definitions by structural recursion that they make must be correct, a property that depends on the meaning of component program phrases. Since these may contain global variables (free in the component phrases but bound in the program as a whole) any denotational semantics must deal with *partiality* even though with structural recursion all computations terminate.
- Checking definedness (meaning exists) is in general not decidable. Only restricted cases might be amenable to compile time definedness checking, similarly to type checking. Checking definedness is in general not even r.e. We can hope, however that programmers can *prove* the correctness of most practical definitions by structural recursion. An (inevitably incomplete) logic of programs would be useful to that effect. Such a logic must have at a minimum definedness and equational formulas, hence it will also serve as a logic of program equivalence. Moreover, structural induction should be available in this logic, as the principal means of proving properties of functions defined by structural recursion.

Investigating these issues is our primary motivation. Our secondary motivation comes from the examples of such datatypes that we have in mind, which are presented in Figure 1. It is natural to ask in what way the two alternative presentations of say, lists, are related. Is there a uniform way of translating programs using structural recursion on *Nil* and *Cons* into programs using structural recursion on *Nil* , *SngList* , and *App* ? And vice-versa? The same questions arise for the alternative presentations of bags and sets. Moreover, the presentations of bags can be seen as hierarchically dominated by those of lists, and those of sets as dominated by those of bags, since, for example, the commutative-idempotent monoids determine a full subcategory in the category of commutative monoids, which is a full subcategory of the category of monoids. Do the above mentioned connections between the alternative presentations of lists uniformly specialize to connections between the two presentations of bags, and further, of sets?

We conclude this section with the observation that all these presentations seem to be very useful. For example, the list reverse function has a more immediate and elegant definition in the *Nil* , *SngSet* , and *App* presentation, than in the *Nil* and *Cons* presentation. In [Breazu-Tannen et al., 1991], we have a program that computes transitive closure using structural recursion on the *Empty* and *Ins* presentation of sets. It is not at all obvious how to directly discover a program for transitive closure using the *Empty* , *SngSet* , and *Union* presentation (although by the work in Section 3 we know that there is a uniform translation).

[2] How then do we program cardinality in this style? Note that addition is still commutative, so an analogous program would have worked fine for computing the cardinality of bags. It is then sufficient to program the fundamental function that coerces a set into a bag. To do this by structural recursion we need the appropriate commutative-idempotent monoid structure on bags. This is given by the "max" operation, and implementing it requires that the elements of the underlying type have equality. (Not surprisingly, since $A = B$ is equivalent to $card(Union\ (SngSet\ (A), SngSet\ (B))) = 1$.)

1.2 Other related work

Pascal has a type of sets but they are restricted to sets of integers or similar, and hence cannot be used for relational database programming. Pascal-R [Schmidt 1977] is a clean and powerful extension that adds a type of relations (sets of tuples) together with some primitive operations on relations and a "for-loop" whose "index" ranges over all tuples in a given relation. This is the imperative analogue of the functional structural recursor in our *ISet* presentation (see Figure 1), and it has therefore the safety problems that we are concerned with (probably only those that appear because of the use of noncommutative counterparts to *Ins* , since we suspect that Pascal-R's operational semantics maintains "canonical" representations for relations). SETL [Schwartz et al., 1986] is a language that lacks the strong typing of Pascal-R but deals with more general kinds of sets. It has however, a similar for-loop construct and hence faces the same problems. One could try to apply our ideas to these languages, but we suspect that the possibility of side-effects in the body of the for-loops will complicate the semantics substantially.

The idea of alternative presentations of lists, and programming using structural recursion in all these presentations also appears in [Wadler 1987]. Wadler also notes that the usefulness of programming with the *AList* presentation has been suggested for reasons of mathematical elegance by Meertens and of efficiency by Sleep .

In programming language theory most of the research on issues involving set semantics has concentrated on powerdomains, with the primary motivations coming from studying nondeterminism and concurrency [Gunter 1986]. A powerdomain semantics for relational and complex object databases is discussed in [Buneman et al., 1989].

1.3 Overview

In Section 2 we present general results about languages built on top of the simply typed lambda calculus with products, by adding datatype presentations like those in Figure 1. The results are stated for an example, that of sets constructed from the empty set, singleton sets and union (the *USet* presentation) but they hold for a general class of such languages. In subsection 2.2 we give a semantics based on sets and partial functions, which nonetheless models the equational constraints for the constructors and the adjointness property used to express the meaning of structural recursion. Programs that are not defined (do not have a meaning) in this model should be rejected. In subsection 2.3 we show that a simple evaluator, deterministic and terminating, and which otherwise ignores the constraints on the constructors, is sound in the previously given denotational semantics, and as a consequence is compatible with the equivalence between different expressions denoting the same set (or bag, or list). In subsection 2.4 we first argue that definedness is in general undecidable, in fact, not even recursively axiomatizable, and then we give a logic that approaches definedness well enough to cover most of the examples we studied. Reasoning about the definedness of terms in turn requires reasoning about term equality. The logic is shown to be sound in the model, and its recursion free fragment is shown to be complete for proving definedness of recursion free programs. The logic is then shown to be as strong as the evaluator, and this implies that the evaluator is compatible with the provable equivalence between different set (or bag, or list) expressions.

In Section 3 we investigate the relation between the alternative presentations of lists, bags, and sets (see Figure 1). In subsection 3.1 we establish back and forth connections between the initiality situations on which these presentations are based. The connections are established first for lists, then they are uniformly specialized to bags, and further, to sets. In subsection 3.2, we present the syntactic counterparts of these semantic connections, in the form of effective translations between the corresponding languages. We end with some proof-theoretic compatibility properties between the translations and the logics introduced in subsection 2.4

CList	*AList*
$Nil_\sigma : CList(\sigma)$ $x:\sigma\ ,\ l: CList(\sigma) \vdash Cons_\sigma(x,l) : CList(\sigma)$	$Nil_\sigma : AList(\sigma)$ $x:\sigma \vdash SngList_\sigma(x) : AList(\sigma)$ $l_1, l_2 : AList(\sigma) \vdash App_\sigma(l_1,l_2) : AList(\sigma)$ $App(l_1, App(l_2,l_3)) = App(App(l_1,l_2),l_3)$ $App(l, Nil) = l = App(Nil, l)$
IBag	*SBag*
$Zero_\sigma : IBag(\sigma)$ $x:\sigma\ ,\ b: IBag(\sigma) \vdash Inc_\sigma(x,b) : IBag(\sigma)$ $Inc(x, Inc(y,b)) = Inc(y, Inc(x,b))$	$Zero_\sigma : SBag(\sigma)$ $x:\sigma \vdash One_\sigma(x) : SBag(\sigma)$ $b_1, b_2 : SBag(\sigma) \vdash Sum_\sigma(b_1,b_2) : SBag(\sigma)$ $Sum(Sum(b_1,b_2),b_3) = Sum(b_1,(Sum\ b_2,b_3))$ $Sum(b, Zero) = b$ $Sum(b_1,b_2) = Sum(b_2,b_1)$
ISet	*USet*
$Empty_\sigma : ISet(\sigma)$ $x:\sigma\ ,\ s: ISet(\sigma) \vdash Ins_\sigma(x,s) : ISet(\sigma)$ $Ins(x, Ins(y,s)) = Ins(y, Ins(x,s))$ $Ins(x, Ins(x,s)) = Ins(x,s)$	$Empty_\sigma : USet(\sigma)$ $x:\sigma \vdash SngSet_\sigma(x) : USet(\sigma)$ $s_1, s_2 : USet(\sigma) \vdash Union_\sigma(s_1,s_2) : USet(\sigma)$ $Union(Union(s_1,s_2),s_3) = Union(s_1, Union(s_2,s_3))$ $Union(s, Empty) = s$ $Union(s_1,s_2) = Union(s_2,s_1)$ $Union(s,s) = s$

Figure 1: Datatypes, Constructors and Equational Constraints

2 Syntax, Semantics and Logic

2.1 Syntax

We consider languages built on top of the simply typed lambda calculus with products, and, say, some unspecified base type ι and some unspecified constants of base type $c_0, c_1, \ldots$. To this, we add datatype presentations like those in Figure 1, plus, for each such datatype, a construct for expressing functions defined by structural recursion. We only specify the notation in the case of the *USet* presentation, which is our generic example. Instead of the syntactically sugared expression in the style of Section 1.1

```
let f(Empty)          = E
 |  f(SngSet(x))      = S x
 |  f(Union(s1,s2))   = U <f s1 , f s2>
in M
```

we use $M[USetRec(E,S,U)/f]$ where

$$\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash S : \sigma \to \tau \quad \Gamma \vdash U : (\tau \times \tau) \to \tau}{USetRec_{(\sigma,\tau)}(E,S,U) : USet(\sigma) \to \tau}$$

Typechecking rules are as in the simply typed lambda calculus, and in our treatment we will assume that the terms under consideration typecheck, but we will routinely omit types and type subscripts.

From the six datatypes that are described in Figure 1, we obtain in this way six different languages, all of which, except for the one given by *CList*, raise the issues discussed in Section 1.1. In subsections 2.2, 2.3, and 2.4, we state our results only for the *USet* language, but it is clear that they generalize to the other four languages of interest as well as a whole class of similar problems.

2.2 Denotational Semantics

Potential, undefinedness has only one source, namely the meaning of structural recursion. In the presence of lambda abstraction however, partiality affects the entire language. Types are interpreted as sets, as follows:

$[\![\iota]\!] \triangleq$ an unspecified nonempty set.
$[\![\sigma \to \tau]\!] \triangleq$ the set of all partial functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$.
$[\![\sigma \times \tau]\!] \triangleq$ the cartesian product of $[\![\sigma]\!]$ and $[\![\tau]\!]$.
$[\![\mathit{USet}\ (\tau)]\!] \triangleq \mathcal{P}_{fin}([\![\tau]\!])$, the set of all finite subsets of $[\![\tau]\!]$.

We call the collection of sets (indexed by types) defined here the *Partial Type Hierarchy* over $[\![\iota]\!]$, notation $\mathcal{P}$. *Environments* are total, type-preserving, functions that map variables to values in $\mathcal{P}$. The meaning of a term t is a partial function $[\![t]\!]$ that maps environments to values in $\mathcal{P}$, defined as follows (the notation $\overset{\Delta}{\simeq}$ means that the left hand side is defined as the right hand side whenever the latter exists, and undefined otherwise).

$[\![x]\!]\rho \triangleq \rho(x)$ $\qquad [\![c_i]\!]\rho \triangleq$ an unspecified element of $[\![\iota]\!]$

$[\![t_1 t_2]\!]\rho \overset{\Delta}{\simeq} [\![t_1]\!]\rho([\![t_2]\!]\rho)$ $\qquad [\![\lambda y.t]\!]\rho \triangleq \Psi$, where $\Psi(a) \overset{\Delta}{\simeq} [\![t]\!]\rho[y/a]$

$[\![\langle t_1, t_2\rangle]\!]\rho \overset{\Delta}{\simeq} \langle [\![t_i]\!]\rho, [\![t_i]\!]\rho\rangle$ $\qquad [\![\pi_i(t)]\!]\rho \overset{\Delta}{\simeq} e_i \;\; where [\![t]\!]\rho = \langle e_1, e_2\rangle).$

$[\![\mathit{Empty}\ ]\!]\rho \triangleq \emptyset$ $\qquad [\![\mathit{SngSet}\ (t)]\!]\rho \overset{\Delta}{\simeq} \{[\![t]\!]\rho\}$ $\qquad [\![Union(t_1, t_2)]\!]\rho \overset{\Delta}{\simeq} [\![t_1]\!]\rho \cup [\![t_2]\!]\rho$

The denotation of a term of the form $\mathit{USetRec}_{(\sigma,\tau)}(E, S, U)$ needs some care. According to the semantic paradigm suggested in subsection 1.1, its meaning in environment ρ should be the unique homomorphism h from $(\mathcal{P}_{fin}([\![\sigma]\!]), \emptyset, \cup)$ to $([\![\tau]\!], [\![E]\!]\rho, [\![U]\!]\rho)$ satisfying $h(\{a\} = [\![S]\!]\rho(a)$, provided that $([\![\tau]\!], [\![E]\!]\rho, [\![U]\!]\rho)$ is a commutative idempotent monoid. In the context of the overall semantics however, $[\![E]\!]\rho$ may be undefined, and $[\![S]\!]\rho$ or $[\![U]\!]\rho$ may be partial, so we need to check that the semantic paradigm carries through.

Define *strong equality* as follows, $e_1 \simeq e_2$ holds if and only if either both e_1 and e_2 are defined and are equal, or they are both undefined. Using this, we can talk about *partial* monoids and homomorphisms.

Lemma 1 *Fix a set X. For any partial commutative idempotent monoid $(M, \cdot, e)$, and any partial function $f : X \longrightarrow M$, there is a unique partial homomorphism $h : (\mathcal{P}_{fin}(X), \cup, \emptyset) \longrightarrow (M, \cdot, e)$ such that $h(\{x\}) \simeq f(x)$*

This suggests that the existence of the meaning of $\mathit{USetRec}_{(\sigma,\tau)}(E, S, U)$ in ρ should be conditioned on $([\![\tau]\!], [\![U]\!]\rho, [\![E]\!]\rho)$ being a partial commutative idempotent monoid. It turns out however, that this is sometimes too strong a condition. More economically, we only need to have a structure of commutative idempotent monoid on those elements of $[\![\tau]\!]$ which will end up in the range of $\mathit{USetRec}(E, S, U)$. This extra flexibility is exploited in Section 3.2 when we translate *ISet* structural recursion via *USet* programs. It is also reflected in the logic of definedness and equality as the basis for the soundness of the "range" induction principles (see subsection 2.4) which are used, among other things, to prove the definedness of the above mentioned translation.

We therefore define $GEN(E, S, U, \rho)$, which we refer to as the *range set*, to be the smallest subset of $[\![\tau]\!]$ with the following properties:

if $[\![E]\!]\rho$ is defined then $[\![E]\!]\rho \in GEN(E, S, U, \rho)$;

if $a \in [\![\sigma]\!]$ and $[\![S]\!]\rho(a)$ is defined then $[\![S]\!]\rho(a) \in GEN(E, S, U, \rho)$;

if $b, c \in GEN(E, S, U, \rho)$ and $[\![U]\!]\rho(b, c)$ is defined then $[\![U]\!]\rho(b, c) \in GEN(E, S, U, \rho)$.

Note that the range of $[\![S]\!]\rho$ is included in $GEN(E, S, U, \rho)$. Now, if for all $a, b, c \in GEN(E, S, U, \rho)$ the following strong equalities hold:

$$
\begin{array}{ll}
\text{(assoc)} & [\![U]\!]\rho([\![U]\!]\rho(a,b),c) \simeq [\![U]\!]\rho(a,[\![U]\!]\rho(b,c)) \\
\text{(ident)} & [\![U]\!]\rho([\![E]\!]\rho,a) \simeq a \simeq [\![U]\!]\rho(a,[\![E]\!]\rho) \\
\text{(comm)} & [\![U]\!]\rho(a,b) \simeq [\![U]\!]\rho(b,a) \\
\text{(idemp)} & [\![U]\!]\rho(a,a) \simeq a
\end{array}
$$

then $(GEN(E, S, U, \rho), [\![U]\!]\rho, [\![E]\!]\rho)$ is a partial monoid and we define $[\![USetRec(E, S, U)]\!]\rho$ to be the unique partial homomorphism $h : \mathcal{P}_{fin}([\![\sigma]\!]) \longrightarrow GEN(E, S, U, \rho)$ such that $h(\{a\}) \simeq [\![S]\!]\rho(a)$, whose existence is insured by Lemma 1. Otherwise, $[\![USetRec(E, S, U)]\!]\rho$ is undefined.

2.3 Operational Semantics

An operational semantics for the language is presented in Appendix B. As usual, only closed terms are evaluated. If $P \Downarrow v$ then v is called a *value*. The product and set constructors are evaluated eagerly. We use a call-by-name rule, but this is not particularly important, since all programs terminate and hence, an evaluator employing call-by-value would be essentially equivalent. The evaluator is deterministic:

Lemma 2 *If $P \Downarrow v_1$ and $P \Downarrow v_2$ then $v_1 \equiv v_2$*

And always terminating:

Theorem 3 (Strong Normalization) *For every closed term P there is a value v such that $P \Downarrow v$.*

This is shown by extending the proof of strong normalization of Gödel's System T (see [Barendregt 1984] or [Girard et al., 1989]).

Next we see that even though the evaluator is oblivious to the equational constraints on the constructors, it is nonetheless sound in the semantics of the previous subsection, which validates these constraints. In what follows, when we say that a closed term is *defined*, we mean that the term has a meaning (in the null environment) in $\mathcal{P}$, according to the semantics in the previous subsection.

Theorem 4 (Soundness of Operational Semantics) *If P is defined, and $P \Downarrow v$, then v is defined, and $[\![P]\!] = [\![v]\!]$ in $\mathcal{P}$.*

The desired compatibility of the evaluator with the datatype constraints follows immediately.

Corollary 5 (Compatibility) *Let P be a defined closed term of type USet $(\sigma) \rightarrow \tau$, and let S_1 and S_2 be two defined closed terms of type USet (σ), denoting the same set. If $PS_1 \Downarrow v_1$ and $PS_2 \Downarrow v_2$ then $[\![v_1]\!] = [\![v_2]\!]$.*

An analysis of the operational semantics shows the following two facts. Any term of type ι evaluates to some constant c_i. Any term of type *USet* (τ) evaluates to a term of the form A$[t_1,..,t_m]$, where $A[]$ is a purely algebraic context composed of $Union$, $SngSet$ and $Empty$, the t_i's are closed terms of type τ. What is usually called *computational adequacy* is an immediate consequence of the strong normalization theorem and the soundness theorem, that if a term is semantically defined, then (a) if its meaning is $c_i \in$ B, then $t \Downarrow c_i$, and (b) if its meaning is a set $\{e_1,..,e_n\}$, then it evaluates to a term of the form A$[t_1,..,t_m]$, where $A[]$ is a purely algebraic context composed of $Union$, $SngSet$ and $Empty$, with $\{[\![t_i]\!] : 1 \leq i \leq m\} = \{e_1,..,e_n\}$.

We also get as an immediate consequence that if two program *phrases* (i.e. two not necessarily closed terms) are semantically equal in all environments, then they are *observationally equivalent*, that is, when put in closed contexts of "observable type" yielding defined terms, they evaluate to values that are semantically equal. Observable types could be, for example, the types of complex objects, which are constructed out of set types, product types and ι. For these, semantic equality of values, is easily seen to be decidable, and completely axiomatizable by the equational constraints on the constructors, hence the same as provable equality in the logic of subsection 2.4.

2.4 Logic of Equality and Definedness

As we have seen in the previous section, the programs that are defined (their meaning in the model of subsection 2.2 exists) evaluate correctly. Moreover, we can reason consistently about such programs, provided that our reasoning is sound in the same model. We advocate, of course, that definedness should be established statically, prior to evaluation. It would be nice if we could deal with definedness checking in the same way we deal with most type checking: automatically and at compile time. Unfortunately:

Theorem 6 *For each of the languages based on AList , IBag , or SBag , checking whether a closed term is defined is Π^0_1-hard.*

We do not know at this time how hard the problem is for the recursion constructs over *ISet* and *USet* . In any case, once we add, say, the natural numbers and recursion over them, one cannot hope for definedness to be recursively axiomatizable. One can, however, hope to prove that given individual programs are correctly defined, and, in fact, we have accumulated a certain amount of experience with such proofs (see, for example, Section 3.2). We think that it is valuable to design formal systems for such proofs, on one hand to serve as rigorous inspiration for informal verifications, on the other to explore the possibility of automating some of the arguments. While such a formal proof system is necessarily incomplete, we see it as evolving and becoming richer in interaction with programming practice, with soundness in the denotational semantics as the only theoretical limitation. The logic we consider in this paper is presented in Appendix B.

Reasoning about the definedness of terms in turn requires reasoning about term equality. Hence, the formulae are universally quantified term equalities, and universally quantified definedness assertions. Up to a point our problems are similar to those of the call-by-value lambda calculus, for instance β and η are not sound in our model. Thus, for reasoning about the recursion free fragment of our languages (base, function, and product types) we have a logic that is essentially equivalent to Moggi's unconditional

λ_p-calculus [Moggi 1985], though we additionally have product types and a couple of rules are stated slightly differently. It is the structural induction principles that give to our logic its distinctive flavor. To deal with induction, we use a notation reminiscent of intuitionistic sequents, but the logic is best viewed as a natural deduction proof system, the left hand side of the sequents containing the premises of the deduction. The premises of induction steps contain eigenvariables that occur, in general, on both sides of the sequent. These variables may be considered to be quantified over the entire sequent.

The usual structural induction principle $\mathit{IND}_{\mathit{USet}}$ (see Appendix B) appears to be insufficient for certain useful arguments in which the induction is valid not on entire types, but only on certain subsets (see subsection 3.2), so we add a "range" induction principle

$$\frac{\Gamma \vdash q[E] \quad \Gamma \vdash q[S\,x] \quad \Gamma,\, q[m]\, q[n] \vdash q[U\,m\,n]}{q[\mathit{USetRec}(E,S,U)s]}(\mathit{RNG} - \mathit{IND}_{\mathit{USet}})$$

where x, m, n are fresh variables. The soundness of this principle makes use of the complication involving "range sets" in the the denotational semantics of structural recursion in subsection 2.2. The range set complication in the model is then reflected in the reasoning about the definedness of the structural recursion construct, namely $\mathit{USetRec}(E,S,U)$. Instead of simply requiring, for example, that $U\langle x, U\langle y,z\rangle\rangle = U\langle U\langle x,y\rangle, z\rangle$ we require that $U\langle Hs_1, U\langle Hs_2, Hs_3\rangle\rangle = U\langle U\langle Hs_1, Hs_2\rangle, Hs_3\rangle$ where $H \equiv \mathit{USetRec}(E,S,U)$. Apparently therefore, proving the definedness of $U\mathit{SetRec}(E,S,U)$ requires proving some equational properties this terms already has! Luckily, these can be shown using the range induction principle.

Theorem 7 (Soundness) *The logic is sound with respect to the denotational semantics in $\mathcal{P}$.*

While the entire logic must be incomplete for reasoning about $\mathcal{P}$, its recursion free fragment could conceivably do better. Moggi shows that the λ_p-calculus is complete for reasoning over all partial cartesian closed categories [Moggi 1985]. We are, of course, not interested in reasoning about models in which we cannot interpret the datatypes with structural recursion. For reasoning in $\mathcal{P}$, Moggi has an *incompleteness* result, but it involves more complex formulas than the ones we are considering [Moggi 1985]. In fact,, we can show the following.

Theorem 8 (Restricted Completeness) *Let t be a term in the recursion free fragment of the language. If t is defined in all environments, then $\mathit{Def}(t)$ is provable in the logic.*

Finally, we investigate the relationship between our logic and the operational semantics. Since the logic is denotationally sound and semantic equality implies observational equivalence, the logic can also be used for reasoning about observational equivalence. Moreover, we can show a few facts that do not follow from this by directly relating the logic and the evaluator.

Theorem 9 *Let P be a closed term. If $\mathit{Def}(P)$ is provable in the logic and $P \Downarrow v$ then $P = v$ is provable in the logic.*

This gives us a proof-theoretic version of the compatibility result in Corollary 5:

Corollary 10 (Provable Compatibility) *Let P be closed term of type USet $(\sigma) \to \tau$, and let S_1 and S_2 be two provably equal closed terms of type USet (σ), such that PS_i are provably defined. If $PS_1 \Downarrow v_1$ and $PS_2 \Downarrow v_2$ then v_1 and v_2 are provably equal.*

3 Relating *Cons* and *App* , *Inc* and *Sum* , *Ins* and *Union*

3.1 The semantic connection

Definition 11 *An X-dynamic is given by an ordered triple (M, δ_M : $X \times M \to M$,e), where M is a non-empty set, and e $\in$ M. A morphism between two X-dynamics (M_1, e_1, δ_{M_1}) and (M_2, e_2, δ_{M_2}), called an X-dynamorphism, is a function g:$M_1 \to M_2$ such that g(e_1) = e_2, and the following diagram commutes:*

$$\begin{tikzcd}
X \times M_1 \arrow[r, "\delta_{M_1}"] \arrow[d, "id \times g"'] & M_1 \arrow[d, "g"] \\
X \times M_2 \arrow[r, "\delta_{M_2}"'] & M_2
\end{tikzcd}$$

For a fixed set X, the category of X-dynamics and X-dynamorphisms is called **DYN**(X). The full subcategory of **DYN**(X) with δ satisfying (1 below) is called **CDYN**(X). The full subcategory of X-dynamics with the δ function satisfying (1) and (2 below) is called **ICDYN**(X).

$$\delta(x, \delta(y, M)) = \delta(y, \delta(x, m)) \tag{1}$$

$$\delta(x, \delta(x, m)) = \delta(x, m) \tag{2}$$

Definition 12 *Given a set X, the category* **MON***(X) is defined as follows: the objects of this category are quadruples (M,$\cdot$,e,η) where (M,$\cdot$,e) is a monoid and η : $X \to M$. A morphism between two such objects (M,$\cdot$,e,η) and (M',$\cdot$',e',η') is a monoid homomorphism h:M$\to$M' with the additional property that the following diagram commutes:*

$$\begin{tikzcd}
 & X \arrow[dl, "\eta"'] \arrow[dr, "\eta'"] & \\
M \arrow[rr, "h"'] & & M'
\end{tikzcd}$$

The full subcategories of **MON**(X), where the underlying monoids of the objects are commutative and commutative idempotent, are called **CMON**(X) and **ICMON**(X), respectively.

Note that ($[\![$*CList* $(\sigma)]\!]$, *Nil* , *Cons*) is an initial object in the category **DYN**($[\![\sigma]\!]$), and that ($[\![$*AList* $(\sigma)]\!]$, *Nil* , *SngList* , *App*) is an initial object in the category **MON**($[\![\sigma]\!]$). A similar relationship holds between the obvious algebras corresponding to *IBag* and *SBag* , on the one hand, and **CDYN**(X) and **CMON**(X), and similarly between the *ISet* and *USet* algebras, on the one hand, and the categories **ICDYN**(X) and **ICMON**(X). In what follows, we will see that, for the case of the list algebras, these two semantics can be constructed from each other, and moreover, that these constructions will uniformly specialize to relate the semantics of *IBag* and *SBag* , and further, of *ISet* and *USet* . More importantly, we will see that these constructions also relate the unique morphisms postulated by initiality, hence the semantics of the structural recursors.

To every object **M** = $(M, \cdot, e, \eta)$ in **MON**(X) we associate an object Φ(**M**) = (M, e, δ) in **DYN**(X), where $\delta(x, m) = \eta(x) \cdot m$.

Theorem 13 **M** *is initial in* **MON**(X) *if and only if* $\Phi(\mathbf{M})$ *is initial in* **DYN**(X).

Proof Sketch. We omit the "only if" part (but warn that it uses induction on the X-dynamics structure on M, and the validity of induction is a consequence of initiality). For the "if" part, let (N, f, ε) be an arbitrary X-dynamics. To exploit the initiality of **M**, we must "fabricate" a monoid out of N. The idea is to take the monoid of (total) functions $N \to N$, with composition and the identity map. In order for the proof to specialize uniformly to bags and sets, we take the extra step (not really necessary for lists) of considering the submonoid of $N \to N$ generated by the set of functions $F = \{\lambda n.\varepsilon(x,n) \mid x \in X\}$, call it F^*. By the initiality of **M** there is a unique **MON**(X)-morphism h from **M** to $(F^*, \circ, id, \lambda x.\lambda n.\varepsilon(x,n))$. Define $g : M \to N$ by $g(m) = h(m)(f)$. It is easy to verify that g is an X-dynamorphism. The uniqueness of g follows from the observation that for every X-dynamorphism q from $\Phi(\mathbf{M})$ to (N, f, ε) we have $q(m) = h(m)(f)$, which is proved by induction on the monoid-generated-by-X structure on M (again the validity of this induction principle follows from initiality). ∎

When we specialize this proof to, say, sets, we show additionally that if $\cdot$ is commutative and idempotent, then so is δ, and that if ε is commutative and idempotent, then so is $\circ$ on F^* ($\circ$ is, in general, neither commutative, nor idempotent on $N \to N$, and this explains the extra step taken in the construction).

Now for the converse construction. For every object $\mathbf{D} = (D, e, \delta)$ in **DYN**(X), we construct, as in the proof of Theorem 13, the submonoid of $D \to D$ generated by the set of functions $F = \{\lambda d.\delta(x,d) \mid x \in X\}$, call it F^*. We associate to **D** the object $\Psi(\mathbf{D}) = (F^*, \circ, id, \lambda x.\lambda d.\delta(x,d))$ in **MON**(X).

Theorem 14 **D** *is initial in* **DYN**(X) *if and only if* $\Psi(\mathbf{D})$ *is initial in* **MON**(X).

Again, this result and its proof uniformly specialize to bags and sets.

Now, if **M** is initial in **MON**(X) then by Theorems 13 and 14, $\Psi(\Phi(\mathbf{M}))$ is also initial, hence isomorphic to **M** . Similarly, if **D** is initial in **DYN**(X), $\Phi(\Psi(\mathbf{D}))$ is isomorphic to **D**.

3.2 Relating languages

The attractive semantic connection presented in subsection 3.1 has a syntactic counterpart, effectively relating the language based on *CList* to that based on *AList* , and similarly for bags and sets. Taking the case of lists, we can (almost), out of the proofs and definitions in subsection 3.1, extract a translation that takes *AList* programs to *CList* programs, and a translation that does the converse. Almost, because the Ψ construction taken literally, would require to define *AList* (σ) as a "subset" of *CList* $(\sigma) \to$ *CList* (σ) and we don't know how to fit this in the system of simple types. Luckily,

Lemma 15 *If* $\mathbf{D} = (D, e, \delta)$ *is initial in* **DYN**(X), *then* $\Psi(\mathbf{D})$ *is isomorphic to* $(D, \cdot, e, \lambda x.\delta(x,e))$ *where* $d_1 \cdot d_2 = g_{d_1}(d_2)$ *with* g_{d_1} *being the unique* X*-dynamorphism from* **D** *to* (D, d_1, δ) .

This suggests that the types can stay the same and that we can get an effective translation of the *AList* constructors by *CList* programs. Moreover, this lemma specializes uniformly to bags and sets.

Figure 2 gives the important definitional clauses of these translations for the *ISet* and *USet* languages. Note that in order to translate structural recursion on *ISet* by a *USet* program, we still make use of the (syntactic counterpart of the) Ψ construction: since in general the (meaning of the) target of the recursion is not initial, Lemma 15 cannot be used.

$$\Phi(\mathit{USet}) = \mathit{ISet}$$
$$\Phi(\mathit{Union}\,(T_1,T_2)) = \mathit{ISetRec}(\Phi(T_2), \lambda\langle x,s\rangle.\mathit{Ins}\,(x,s))\Phi(T_1)$$
$$\Phi(\mathit{SngSet}\,(A)) = \mathit{Ins}\,(\Phi(A),\mathit{Empty}\,) \qquad \Phi(\mathit{Empty}\,) = \mathit{Empty}$$
$$\Phi(\mathit{USetRec}(E,S,U)) = \mathit{ISetRec}(\Phi(E), \lambda\langle x,z\rangle.\Phi(U)(\Phi(S)x,z))$$
$$\Psi(\mathit{ISet}\,) = \mathit{USet}$$
$$\Psi(\mathit{Empty}\,) = \mathit{Empty} \qquad \Psi(\mathit{Ins}\,(A,T)) = \mathit{Union}\,(\mathit{SngSet}\,(\Psi(A)),\Psi(T))$$
$$\Psi(\mathit{ISetRec}(E,I)) = \lambda s.(\mathit{USetRec}(\mathit{id}, \lambda x.\lambda z.\Psi(I)(x,z), \circ)\, s\, \Psi(E))$$

Figure 2: Translations

Let $\mathcal{L}_{\mathit{ISet}}$ and $\mathcal{L}_{\mathit{USet}}$ stand for the languages with *ISet* and *USet*, respectively, as the datatypes. Let *Th* stand for the rules and axioms in the recursion free fragment. Let $\mathit{Th}_{\mathit{ISet}}$ stand for *Th* plus rules for *ISetRec* (see Appendix B) without the two induction rules. Let $\mathit{Th}_{\mathit{USet}}$ stand for *Th* plus rules for *USetRec* (see Appendix B) with out the two induction rules. The following theorem establishes a proof theoretic relationship between equations in the Language $\mathcal{L}_{\mathit{ISet}}$, and their translates in $\mathcal{L}_{\mathit{USet}}$ via Ψ, and equations in the language $\mathcal{L}_{\mathit{USet}}$ and their $\mathcal{L}_{\mathit{ISet}}$ translates via Φ.

Theorem 16 *(i) If* $\mathit{Th}_{\mathit{ISet}} \vdash t_1 = t_2$ *then* $\mathit{Th}_{\mathit{USet}} + \mathit{IND}_{\mathit{USet}} \vdash \Psi(t_1) = \Psi(t_2)$.

(ii) If $\mathit{Th}_{\mathit{USet}} \vdash t_1 = t_2$ *then* $\mathit{Th}_{\mathit{ISet}} + \mathit{IND}_{\mathit{ISet}} \vdash \Phi(t_1) = \Phi(t_2)$.

(iii) For any term t in the language $\mathcal{L}_{\mathit{USet}}$, $\mathit{Th}_{\mathit{USet}} + \mathit{IND}_{\mathit{USet}} \vdash t = \Psi(\Phi(t))$.

(iv) For any term s in the language $\mathcal{L}_{\mathit{ISet}}$, $\mathit{Th}_{\mathit{ISet}} + \mathit{IND}_{\mathit{ISet}} \vdash s = \Psi(\Phi(s))$

In subsections 2.4 and 2.2 we motivated the range set construction and the range induction principle, respectively, to deal with structural recursions $\mathit{USetRec}_{(\sigma,\tau)}(E,S,U)$ in which $([\![\tau]\!], [\![U]\!], [\![E]\!], [\![U]\!])$ is not a monoid, while $(GEN_\tau(E,S,U,\rho), [\![U]\!]\rho, [\![E]\!]\rho, [\![U]\!]\rho)$ is. Dealing with such recursions is important in Theorem 16. For instance, consider the translation (via Ψ) of the (provably defined) closed term $\mathit{ISetRec}(E,I)$ (Here, F provably has the property that $I(x,I(y,z)) = I(y,I(x,z))$, ($z$ ranging over the set $GEN(E,I)$.) Proving the definedness of the Ψ-translation this term requires proving the definedness of $\mathit{USetRec}(\mathit{id},\Phi(I),\circ,))$. It is clear, that in general $\circ$ is not commutative, but it is commutative over the set $GEN_\tau(\mathit{id},\Phi(I),\circ,\rho)$ (and similarly for idempotence).

4 Further research

The reader may have noticed that we have avoided fixed point recursion. One of our main concerns is the choice of basic constructs for programming with sets and bags. For datatypes like *CList* or binary trees, for example, there is a simple minded *translation* of structural recursion into fixed point recursion which uses "inverses" to the constructors (*head* and *tail* for *CList* or taking the root, and the left and right subtree for binary trees). In general, no such inverses exist for datatypes with equationally constrained constructors. Some languages implement "non-deterministic" inverses, for example "choose" as an inverse to set insert, "partition" as an inverse to bag sum. It seems that to program with sets using "choose", one would need a "delete" function as well, restricting one to deal with equality types. That aside, a clean semantics for these seems to be considerably more complicated than that of structural recursion, and the presence of fixed point recursion suggests powerdomains. An in depth comparison of such an approach with the one that we take in this paper seems interesting.

So far, we did not develop our treatment enough to cover generalized structural recursion (see subsection 1.1) but this is clearly of great practical importance and is next in line. Further down the line, it would be interesting to see the interaction of structural recursion on datatypes with general recursion in the language. Is the logic we propose in this paper still sound for proving observational equivalence?

Our investigation here raises a number of questions. We would like to understand the proof theory better. For example, is the λ_p-calculus, which is part of our logic, conservatively extended by our logic? Are there more restricted completeness results or normal form theorems (for either the entire logic, or for interesting fragments of it)? We leave open the question of decidability of definedness and semantic equality for the languages with *USet* and *ISet* datatypes (see Theorem 6 in subsection 2.4).

Yet another issue is a precise understanding of the intuition that any program on sets can also be viewed *mutatis mutandis* as a program on bags and further down on lists. For example, a definition of the cartesian product function on *ISet* , after merely replacing the recursor and constructor names by the recursor and constructor names in the language containing *CList* , will become a list cartesian product function (though there are more than one list cartesian product functions).

Finally we remark that Theorem 16 is restricted to the translation of equalities that have induction free proofs, even though induction is used in the proof of the translated equalities. Based on the semantic results in subsection 3.1, we expect that inductive proofs can also be translated, but it seems that we cannot stay within an equational logic while doing this.

Acknowledgements: We would like to thank Peter Buneman for inspiration and many useful discussions. We thank Francisco Corella for his criticisms on an earlier draft. Val Breazu-Tannen thanks Octavian Stănăşilă for many discussions on X-dynamics and monoids.

References

[Atkinson& Buneman 1987] M. P. Atkinson. Types and Persistence in Database Programming Languages. ACM Computing Surveys, June 1987.

[Backus 1978] J. Backus. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*,21:613-641, August 1978.

[Barendregt 1984] H. P. Barendregt . The Lambda Calculus: Its Syntax and Semantics. Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North Holland, Amsterdam, Second Edition, 1984.

[Bird & Wadler 1988] R. Bird and P. Wadler. Introduction to Functional Programming. *Series in Computer Science*, Prentice-Hall International, 1988.

[Breazu-Tannen et al., 1991] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. Proceedings of the 3rd International Workshop on Database Programming Languages, Naphlion, Greece, August 1991, to appear.

[Buneman et al., 1989] P. Buneman, A. Jung, and A. Ohori. Using Powerdomains to Generalize Relational Databases. In *Theoretical Computer Science*, August 1989.

[Codd 1970] E. F. Codd. A Relational Model for Large Shared Databank. In *Communications of the ACM*, 13(6):377-387, 1970.

[Girard et al., 1989] J. Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Gunter 1986] C. A. Gunter. Sets and the Semantics of Bounded Non-Determinism. Manuscript, University of Cambridge, 1986.

[Moggi 1985] E. Moggi. Categories of Partial Morphisms and the λ_p-calculus. In *Proceedings of Category Theory and Computer Programming*, Springer-Verlag, 1985.

[Ohori et al., 1989] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli - a Polymorphic language with Static Type Inference. In *Proceedings of the ACM SIGMOD Conference*, pages 46-57, 1989.

[Schmidt 1977] J. W. Schmidt. Some High-Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3):247-261, September 1977.

[Schwartz et al., 1986] J. T. Schwartz, R.B.K Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Itroduction to SETL*, Springer-Verlag, New York, 1986.

[Wadler 1987] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the Conference on Principles of Programming Languages*, pages 307-313, 1987.

APPENDIX A : OPERATIONAL SEMANTICS

In the following H abbreviates $USetRec(E, S, U)$.

$$c \Downarrow c \qquad Empty \Downarrow Empty \qquad \lambda x.M \Downarrow \lambda x.M \qquad H \Downarrow H$$

$$\frac{M \Downarrow v}{SngSet\,(M) \Downarrow SngSet\,(v)} \qquad \frac{M \Downarrow v_1 \quad N \Downarrow v_2}{Union\,(M,N) \Downarrow Union\,(v_1,v_2)} \qquad \frac{M \Downarrow (\lambda x.L) \quad L[N/x] \Downarrow v}{MN \Downarrow v}$$

$$\frac{M \Downarrow \langle N_1,N_2\rangle \quad N_1 \Downarrow v}{\pi_1(M) \Downarrow v} \qquad \frac{M \Downarrow \langle N_1,N_2\rangle \quad N_2 \Downarrow v}{\pi_2(M) \Downarrow v} \qquad \frac{M \Downarrow u \quad N \Downarrow v}{\langle M,N\rangle \Downarrow \langle u,v\rangle}$$

$$\frac{M \Downarrow H \quad N \Downarrow Empty \quad E \Downarrow v}{MN \Downarrow v} \qquad \frac{M \Downarrow H \quad N \Downarrow SngSet\,(P) \quad SP \Downarrow v}{MN \Downarrow v}$$

$$\frac{M \Downarrow H \quad N \Downarrow Union\,(P_1,P_2) \quad U\,\langle (HP_1),(HP_2)\rangle \Downarrow v}{MN \Downarrow v}$$

APPENDIX B :LOGIC

Recursion Free Logic

$\Gamma \vdash q \quad (q \in \Gamma)$ $\qquad$ $\Gamma \vdash \mathit{Def}(x)$ $\qquad$ $\Gamma \vdash t \approx t$

$\Gamma \vdash \mathit{Def}(\lambda x.t)$ $\qquad$ $\Gamma \vdash (\lambda x.t)x \approx t$ $\qquad$ $\Gamma \vdash \lambda x.yx \approx y$ (y is a variable)

$\mathit{Def}(t_1 t_2) \vdash \mathit{Def}(t_1)$ $\qquad$ $\mathit{Def}(t_1 t_2) \vdash \mathit{Def}(t_1)$

$$\frac{\Gamma \vdash q_1 \;\ldots\; \Gamma \vdash q_n \qquad q_1,\ldots,q_n \vdash q}{\Gamma \vdash q} \qquad \frac{\Gamma_1 \vdash \mathit{Def}(t_i)\ (i \le n) \quad \Gamma_2 \vdash q}{\Gamma_2[\vec{x}/\vec{t}] \cup \Gamma_1 \vdash q[\vec{x}/\vec{t}]} \qquad \frac{\Gamma \vdash t_1 \approx t_2}{\Gamma \vdash t_2 \approx t_1}$$

$$\frac{\Gamma \vdash t_1 \approx t_2 \quad \Gamma \vdash t_2 \approx t_3}{\Gamma \vdash t_1 \approx t_3} \qquad \frac{\Gamma \cup \{\mathit{Def}(t_1)\} \vdash t_1 \approx t_2 \quad \Gamma \cup \{\mathit{Def}(t_2)\} \vdash t_1 \approx t_2}{\Gamma \vdash t_1 \approx t_2}$$

$$\frac{\Gamma \vdash t_1 \approx t_2}{\Gamma \vdash C[t_1] \approx C[t_2]} \qquad \frac{\Gamma \vdash t_1 \approx t_2 \quad \Gamma \vdash \mathit{Def}(t_1)}{\Gamma \vdash \mathit{Def}(t_2)}$$

$$\frac{\Gamma \vdash \mathit{Def}(t_1) \quad \Gamma \vdash \mathit{Def}(t_2)}{\Gamma \vdash \mathit{Def}(\langle t_1, t_2 \rangle)} \qquad \frac{\Gamma \vdash \mathit{Def}(\langle t_1, t_2 \rangle)}{\Gamma \vdash \mathit{Def}(t_i)} i = 1,2 \qquad \frac{\Gamma \vdash \mathit{Def}(t)}{\Gamma \vdash \mathit{Def}(\pi_i(t))} i = 1,2$$

$$\frac{\Gamma \vdash \mathit{Def}(\pi_i(t))}{\Gamma \vdash \mathit{Def}(t)} i = 1,2 \qquad \Gamma \vdash \langle \pi_1(t), \pi_2(t) \rangle \approx t \qquad \frac{\Gamma \vdash \mathit{Def}(\langle t_1, t_2 \rangle)}{\Gamma \vdash \pi_i(\langle t_1, t_2 \rangle) \approx t_i} i = 1,2$$

Rules For *USetRec* and associated Constructors

$\Gamma \vdash \mathit{Union}\,(s_1, \mathit{Union}\,(s_2, s_3)) \approx \mathit{Union}\,(\mathit{Union}\,(s_1, s_2), s_3)$ $\quad$ $\Gamma \vdash \mathit{Union}\,(s, s) \approx s$

$\Gamma \vdash \mathit{Union}\,(s, \mathit{Empty}) \approx s$ $\quad$ $\Gamma \vdash \mathit{Union}\,(s_1, s_2) \approx \mathit{Union}\,(s_2, s_1)$ $\quad$ $\Gamma \vdash \mathit{Def}(\mathit{Empty})$

$$\frac{\Gamma \vdash \mathit{Def}(\mathit{SngSet}\,(t))}{\Gamma \vdash \mathit{Def}(t)} \qquad \frac{\Gamma \vdash \mathit{Def}(t)}{\Gamma \vdash \mathit{Def}(\mathit{SngSet}\,(t))}$$

$$\frac{\Gamma \vdash \mathit{Def}(\mathit{Union}\,(t_1, t_2))}{\Gamma \vdash \mathit{Def}(t_i)} \qquad \frac{\Gamma \vdash \mathit{Def}(t_1) \quad \Gamma \vdash \mathit{Def}(t_2)}{\Gamma \vdash \mathit{Def}(\mathit{Union}\,(t_1, t_2))}$$

In the following H abbreviates $\mathit{USetRec}(E, S, U)$:

$$\frac{\Gamma \vdash \mathit{ICMON}(E, S, U)}{\Gamma \vdash \mathit{Def}(H)} \qquad \frac{\Gamma \vdash \mathit{Def}(H)}{\Gamma \vdash H\,\mathit{Empty} \approx E} \qquad \frac{\Gamma \vdash \mathit{Def}(H)}{\Gamma \vdash H\,\mathit{SngSet}\,(t) \approx S\,t}$$

$$\frac{\Gamma \vdash \mathit{Def}(H)}{\Gamma \vdash H\,\mathit{Union}\,(t_1, t_2) \approx U\langle H\,t_1, H\,t_2 \rangle}$$

Here $\Gamma \vdash \mathit{ICMON}(E, S, U)$ abbreviates the following premisses (x, y, z are fresh variables):

(*ident*) $\Gamma \vdash U\langle E, H\,x \rangle \approx H\,x \approx U\langle H\,x, E \rangle$

(*assoc*) $\Gamma \vdash U\langle H\,x, U\langle H\,y, H\,z \rangle\rangle \approx U\langle U\langle H\,x, H\,y \rangle, H\,z \rangle$

(*comm*) $\Gamma \vdash U\langle H\,x, H\,y \rangle \approx U\langle H\,y, H\,x \rangle$

(*idemp*) $\Gamma \vdash U\langle H\,x, H\,x \rangle \approx H\,x$

$$\frac{\Gamma \vdash e[\mathit{Empty}/z] \quad \Gamma \vdash e[\mathit{SngSet}\,(x)/z] \quad \Gamma, e[s_1/z], e[s_2/z] \vdash e[\mathit{Union}\,(s_1, s_2)/z]}{\Gamma \vdash e}(\mathit{IND}_{\mathit{USet}})$$

$(x, s_1, s_2$ are fresh$)$

$$\frac{\Gamma \vdash q[E] \quad \Gamma \vdash q[S\,x] \quad \Gamma,\, q[m]\,q[n] \vdash q[U\,m\,n]}{q[\mathit{USetRec}(E, S, U)s]}(\mathit{RNG} - \mathit{IND}_{\mathit{USet}}) \qquad (x, m, n \text{ are fresh})$$

APPENDIX C : **RULES FOR *ISetRec***

$$\Gamma \vdash \mathit{Def}(\mathit{Empty}) \qquad \frac{\Gamma \vdash \mathit{Def}(\mathit{Ins}\,(t_1, t_2))}{\Gamma \vdash \mathit{Def}(t_i)} \qquad \frac{\Gamma \vdash \mathit{Def}(t_1) \quad \Gamma \vdash \mathit{Def}(t_2)}{\mathit{Def}(\mathit{Ins}\,(t_1, t_2))}$$

$$\Gamma \vdash \mathit{Ins}\,(x, \mathit{Ins}\,(y, s)) \approx \mathit{Ins}\,(y, \mathit{Ins}\,(x, s)) \qquad \Gamma \vdash \mathit{Ins}\,(x, \mathit{Ins}\,(x, s)) \approx \mathit{Ins}\,(x, s)$$

In the following H abbreviates $\mathit{ISetRec}(E, I)$.

$$\frac{\Gamma \vdash \mathit{ICDYN}(E, I)}{\Gamma \vdash \mathit{Def}(H)} \qquad \frac{\Gamma \vdash \mathit{Def}(H)}{\Gamma \vdash H\,\mathit{Empty} \approx E} \qquad \frac{\Gamma \vdash \mathit{Def}(H)}{\Gamma \vdash H\,\mathit{Ins}\,(x, s) \approx I\langle x, (H\,s)\rangle}$$

Here $\Gamma \vdash \mathit{ICDYN}(E, I)$ abbreviates the following sequents (x, y, z are fresh):

$(\mathit{comm}) \quad \Gamma \vdash I\langle x, \langle y, H\,z\rangle\rangle \approx I\langle y, \langle x, H\,z\rangle\rangle$

$(\mathit{idemp}) \quad \Gamma \vdash I\langle x, \langle x, H\,z\rangle\rangle \approx I\langle x, H\,z\rangle$

$$\frac{\Gamma \vdash e[\mathit{Empty}/z] \quad \Gamma, e[s/z] \vdash e[\mathit{Ins}\,(x, s)/z]}{\Gamma \vdash e}(\mathit{IND}_{\mathit{ISet}}) \qquad (x, s \text{ are fresh})$$

$$\frac{\Gamma \vdash q[E] \quad \Gamma,\, q[m] \vdash q[I\,x\,m]}{q[\mathit{ISetRec}(E, I)s]}(\mathit{RNG} - \mathit{IND}_{\mathit{ISet}}) \qquad (x, m \text{ are fresh})$$