# The Abstraction/Implementation Model of Problem Reformulation

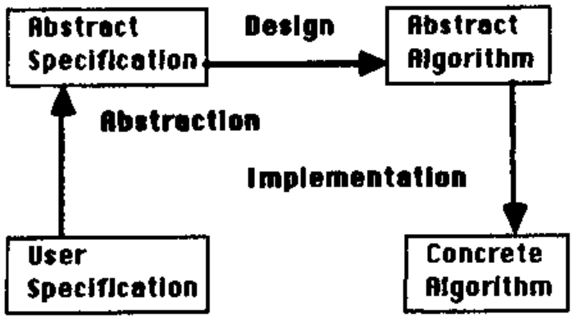
Michael R. Lowry
Stanford Artificial Intelligence Laboratory
Box 3350, Stanford CA 94305 USA
And Kestrel Institute
1801 Page Mill Road, Palo Alto CA 94304 USA

## Abstract

*A good problem representation incorporates important problem constraints while hiding superfluous detail.* This paper presents methods for abstracting a problem representation by making implicit problem properties into explicit properties of the representation. The mathematics of the abstraction search space are given in terms of model theory and universal algebra. The Behavioral Abstraction method uses predefined representation maps to lift a problem representation to an abstract theory. The Behavioral Congruence method *generates* abstract theories and representation maps which incorporate problem constraints expressed as Behavioral Equivalences. Two meta-level methods for generating Behavioral Equivalence theorems are given. The STRATA automatic programming system described in this paper is currently being implemented.

## I Introduction

This paper[1] and its companion AAAI87 paper[Lowry, 1987a] present a framework for problem reformulation and algorithm synthesis. In this framework, both specifications and algorithms are viewed as theories, where an algorithm is a theory about a sequence of state changes. Reformulation is viewed as a representation map between theories. The mathematics of theory mappings can be found in Goguen's work on abstract model theory [Goguen and Burstall, 1985]. This paper explores reformulation as implementation maps between theories, while the AAAI87 paper explores reformulation as instantiation maps between parameterized theories.



[1] This paper describes work done at Stanford University under DARPA contract N00039-84-C-0211, and at the Kestrel Institute under ONR contract N00014-84-C-0473.

The overall framework is presented in the diagram above. First, STRATA *ABSTRACTS* the specification in order to remove superfluous distinctions that are found in the user's initial conceptualization of the problem. This step is explained in this paper. Second, STRATA *DESIGNS* an abstract algorithm by instantiating algorithm schemas which are formalized as parameterized theories. This step is explained in the companion AAAI87 paper, which illustrates the *DESIGN* step by using the parameterized theory of local search to derive the simplex algorithm. Third STRATA *IMPLEMENTS* the abstract algorithm using stepwise refinement. Most work in automatic programming addresses this third step.

STRATA uses the RAINBOW directed inference system developed by Douglas Smith[Smith, 1985] for theorem proving and theorem generation. The third step of *IMPLEMENTATION* is carried out by the *REFINE™* compiler, which embodies a decade's research in automatic programming done at the Kestrel Institute.

The subject of this paper is specification abstraction. Given a problem specification in a problem domain theory, STRATA finds an equivalent problem specification in a more abstract problem domain theory. Globally, the space of theories and implementation mappings form an associative directed multigraph (i.e. a category). There is no global abstraction ordering on theories. However, *with respect to each input-output behavior,* there is an abstraction ordering on theories. For this reason, this overall process is called *Behavioral Abstraction.* The mathematical foundations for the search space of behavioral abstraction is explained in section 3.

There are two cases to consider for behavioral abstraction. In the first case, the multigraph of theories and implementation mappings is predefined, and STRATA performs Behavioral Abstraction by proving *behavioral theorems* and applying *inverse* theory maps. A behavioral theorem is a sentence about an important problem property, and is used as an enabling condition for applying an inverse implementation map. An important subclass of behavioral theorems are behavioral equivalences, that is two objects which behave equivalently with respect to the input-output relation.

When STRATA generates a behavioral theorem which does not correspond to a predefined inverse implementation map, then STRATA generates a new, more abstract theory and the implementation map between this new theory and the original theory. This new theory results from incorporating the behavioral theorem into the language and axioms of the new theory. When the behavioral theorem is a behavioral equivalence, generating the new theory is called *behavioral congruence.* Behavioral congruence is the main technical result of this paper.

The method of Behavioral Abstraction is given below:

1. Behavioral Theorems: Find important problem properties.
2. Is the Behavioral Theorem an enabling condition for applying a predefined inverse implementation map?
3. YES - apply the inverse implementation map to obtain a logically equivalent problem specification in a more abstract theory.
4. NO - generate the theory which incorporates the behavioral theorem. For the case of behavioral equivalences, this is called behavioral congruence. Go to Step 3.

The rest of this paper describes the methods used by STRATA to abstract a problem definition into an equivalent abstract reformulation that incorporates constraints of the IO-relation. Section 2 describes the abstraction hierarchy which is searched by behavioral abstraction. In section 3 a simple combinatorial example is introduced to illustrate the abstraction methods. Section 4 describes behavioral equivalence. Section 5 defines the method of behavioral abstraction. Section 6 and 7 contain the main results of behavioral congruence, which is a method for turning a behavioral equivalence in a theory into an equality in a more abstract theory. Only when behavior ally equivalent objects are made identical has the equivalence been incorporated into a new representation.

The pioneering work of Amarel[Amarel, 1968] 20 years ago showed the potential power of reformulation. About 1980, a number of people began investigating methods for searching the space of logically equivalent problem reformulations. The abstraction space formalized in this model is similar to the homomorphic reformulations described in [Korf, 1980]. Another type of search heuristic is to find problem reformulations targeted to a particular problem solving schema such as Divide and Conquer[Smith, 1985], or Heuristic Search[Mostow, 1983]. In [Lowry, 1987a], problem solving schemas are formalized as *parameterized theories,* and hence generalizes the work cited above. In the literature on abstract data types, [Goguen and Meseguer, 1982] it has been shown that abstraction implementation commutes with problem solving schema instantiation.

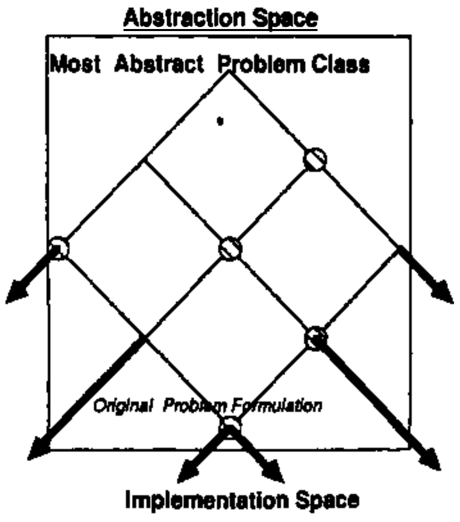
## II Mathematics of the Abstraction Space

This section describes the mathematics of behavioral abstraction from a model theoretic viewpoint in order to understand the behavioral abstraction search space. The model theoretic viewpoint is closely related to the mathematics for the implementation relationship described in [Goguen *et* o/., 1978].

The basic idea behind reformulation is that the class of models for the original problem formulation is isomorphic to the class of models for the reformulated problem definition. This key idea is extended for behavioral abstraction to allow merging models of the concrete theory which are identical with respect to the IO-behavior. This merging comes in two flavors. First, a *derivor* forgets some of the relations and functions which are not directly of interest. For example, the IO-derivor forgets everything except the input/output relation. This is in part how the most abstract problem formulation will be defined. The second flavor of merging is through equivalences. This is the main subject of this paper. Objects which behave equivalently with respect to the IO-relation are merged together into an identity.

The most abstract class of models for a given problem formulation is obtained by first forgetting everything except the io-relation (or io-relations/functions if there is more than one), and then identifying all objects which behave equivalently. This most abstract class is the top of a lattice structure whose bottom is the original problem formulation. Some of the points in this lattice will not have finite axiomatizations, because taking a derivor does not necessarily preserve finite axiomatizability.

The space of tractable problem abstractions, corresponding to the classes of models which have finite axiomatizations, have a semi-lattice structure with the original problem formulation at the bottom. [Lowry, 1987b] describes this structure in more detail. The objective of behavioral abstraction is to move upwards in this space of tractable problem abstractions. The diagram below shows the space of abstract problem reformulations, the circles denote tractable abstractions. The space of implementations is obtained by unwinding the multigraph of implementations.



The definition of an implementation link is as a representation map from an abstract theory to a concrete theory, with an abstraction function for mapping concrete objects to abstract objects. The diagram below shows an implementation link between bags and lists, where the abstract theory of bags and the concrete theories of lists are axiomatized using equational logic.

| *Bag Theory* | *Map* | *List Theory* |
|---|---|---|
| *BagofOperators* | → | *ListofOperators* |
| *Operators* | → | *Operators* |
| *NullBag* | → | *Nil* |
| *MakeBag* | → | *MakeList* |
| *BagUnion* | → | *Append* |
| *AddBag* | → | *Cons* |
| *BagUnion Axioms* |  | *Append Axioms* |
| *Identity* |  | *Identity* |
| *Associativity* |  | *Associativity* |
| *Commutative* |  |  |

Abstraction Map from lists to bags:

*Abs(Nil) = NullBag*
*Abs(MakeList(x)) = MakeBag(x)*
*Abs(Append(Ll,L2) = BagUnion{Abs(Ll), Abs(L2))*

In general there are many implementation links between two theories, going in both directions. It is assumed that a knowledge base has some of these implementation links predefined. Subsequent sections show how to automatically use predefined implementation links for behavioral abstraction when they are available, and also to generate new implementation links when given a novel problem constraint. The next page has the equational theory for lists. The equational theory for bags is obtained by renaming append to bagunion and making it commutative. The equational theory for sets is obtained by renaming bagunion to set union and making it idempotent (a set unioned with itself yields the same set). A different equational theory for sets is obtained by renaming bagunion to symmetric-set-difference and making it self-conjugate (a set differenced with itself yields the null set). This demonstrate how adding an equality to a theory can generate an abstract theory. These theories will be used in the example.

Equational Theory of Lists

| *sorts* | *example* |
|---|---|
| *operators* | $O1$ |
| *ListsofOperators* | $[O1, O3, O4, O2, O4]$ |
| ***functions*** | ***example*** |
| *nil* | $[]$ |
| *MakeList* | $MakeList(O1) \rightarrow [O1]$ |
| *Append* | $Append([O1, O3, O4], [O2, O4]) \rightarrow [O1, O3, O4, O2, O4]$ |
| ***Axiom*** | ***Definition*** |
| *identity* | $Append(nil, L) = L$ |
| *associative* | $Append(L1, Append(L2, L3)) = Append(Append(L1, L2), L3)$ |

## III A Simple Example

This example will be used to illustrate the methods described in subsequent sections.

1101001 *Start String*
01100 *Operator*
$\boxed{11000}$
*etc.*
$\boxed{11011}$ *Goal String*

The arrow puzzle [2] is a state space search problem where the states are bit strings and the operators are bit strings which are applied by bit-wise XORing with a state to yield a new state. The input sort is a StartString GoalString pair, and the output sort is a list(sequence) of operators. The input /output relation *SOLVE* is defined as:
*SOLV E({StartString, GoalString}, OpList) = Apply(Reduce(XOR,OpList), StartString) = GoalString*

*Reduce* takes a binary operator and a list of arguments, iteratively applying the operator to the accumulated result and the next element of the list. *Apply* takes an operator and bitwise XORs it with a bitstring. Reduce and Apply are defined using conditional equations as follows:
*if list = null then Reduce(binop,hst) = identity*
*if list / null then*
*Reduce(bmopjist) = car(list)binopReduce(binop,cdr(hst))*
*Apply (operator, Bit String) — operator XORBitStrtng*

The properties of XOR, denoted ⊗, assumed to be defined in the knowledge base, are used to reformulate the problem:

| | |
|---|---|
| *Commutative* | $A \otimes B = B \otimes A$ |
| *Associative* | $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ |
| *Identity* | $A \otimes 0 = A$ |
| *SelfConjugate* | $A \otimes A = 0$ |

The commutativity and associativity of XOR means that a list of operators can be arbitrarily re-ordered, which is used to lift the output sort from list of operators to bags of operators. An alternative representation for a bag is an exponential notation, which maps each operator into its number of occurrences in the bag. The self-conjugate property of XOR means that applying an operator twice is equivalent to not applying it at all. In other words, only the even/odd parity of the number of operator occurrences is relevant. Thus the exponential notation is collapsed to the characteristic function for a set by mapping the natural numbers to mod2. Similarly, StartString GoalString pairs can be collapsed into a single bit-string by XORing them together. This single bit-string represents the class of StartString GoalString pairs that have the same solutions. The following transformations illustrate these abstractions:

Lists to Bags $[O1, O3, O4, O2, O4] \Rightarrow \{O1, O2, O3, O4, O4\}$
Exponents $\{O1, O2, O3, O4, O4\} \Rightarrow \{O1^1, O2^1, O3^1, O4^2\}$

[2]Korf presented this puzzle with arrows pointed up or down to represent bits. He made additional assumptions about the set of operators which are irrelevant to the abstraction presented here.

Bags **to Sets** $\{O1^1, O2^1, O3^1, O4^2\} \Rightarrow \{O1^t, O2^t, O3^t, O4^f\}$

Pairs to bit-strings $< 10111, 00101 > \Rightarrow 10010$

Note that certain choices of operators might not span the space of all possible Start String GoalString pairs. The search will not terminate unless the search space can be made finite. There are an infinite number of possible operator lists and operator bags. However, there are only a finite (but exponential) number of operator sets; thus the search is guaranteed to terminate for the abstract problem reformulation.

## IV Behavioral Equivalence Theorems

The first step in Behavioral Abstraction is to find a problem property which will then be incorporated in a new representation for the problem. Behavioral Equivalences are an important subclass of problem properties. Two objects are behaviorally equivalent when they behave identically with respect to the 10 relation. In the schema given below, terms of the input sort are prefixed with /n, terms of the output sort are prefixed with *Out,* and the 10 relation is designated *R.* The first schema is for input objects that behave identically, the second schema is for output objects that behave identically.[3] (The symbol *=Bth* denotes equivalence with respect to the IO-relation).

Behavioral Equivalence Schemas

$$In1 \cong_{Beh} In2 \text{ iff } \forall Out\ R(In1, Out) \leftrightarrow R(In2, Out)$$
$$Out1 \cong_{Beh} Out2 \text{ iff } \forall In\ R(In, Out1) \leftrightarrow R(In, Out2)$$

Two general meta-level methods for generating behavioral equivalence theorems, the kernel method and the homomorphism method, are given in [Lowry, 1987b]. In universal algebra, the kernel of a function f whose domain is sort A and whose range is sort B is defined as the equivalence relation on A of elements which are mapped to the same element in B. A homomorphism is a function f from A to B that maps a function g on A to a function h on B. The kernel of g is mapped to the kernel of h (or a subset of the kernel of h). The generalization of homomorphism to many sorted logics can be found in [Goguen *ti* a/., 1978].

For the arrow puzzle, the kernel method yields a behavioral equivalence on the input sort, which is a Start-String GoalString pair. The function XOR maps a pair of bit-strings to a single bit-string. This defines an equivalence class on pairs of bit strings. The following behavioral equivalence theorem describes this equivalence class:

$$ss1 \otimes gs1 = ss2 \otimes gs2 \Rightarrow$$
$$\{ss1, gs1\} \cong_{Beh} \{ss2, gs2\}$$

For the arrow puzzle, the homomorphism method yields two behavioral equivalences on the output sort, which is a list of operators. The homomorphism function in this case is REDUCE, which maps a list of operators to a single bit string representing the composite operator. The homomorphism maps the properties of XOR, particularly commutativity and self-conjugate, to behavioral equivalences of append:

$$\text{append}(Lop1, Lop2) \cong_{Beh} \text{append}(Lop2, Lop1)$$
$$\text{append}(Lop1, Lop1) \cong_{Beh} \text{nil}$$

## V Behavioral Abstraction

The basic idea of behavioral abstraction is to apply stepwise implementation in reverse. The representation map of an implementation link has all the primitive functions and relations of the abstract theory as its source and often derived functions and relations in the concrete theory as its target. Stepwise implementation simply translates a complex relation in the abstract theory through the representation map to a complex relation in the concrete theory. In general, the representation map is only partially invertible, so a problem formulated in the concrete theory cannot always be simply mapped to the abstract theory. For example, in the bag theory to list theory representation map given in section 3, there is no inverse map for CAR or CDR. This is because CAR and CDR are non-determinate when applied to bags.

When a problem formulated in the concrete theory is defined solely in terms of the image of a representation map, then the inverse map can be directly applied to derive an abstract problem reformulation. Often, it is possible to prove that a problem has some abstract property which implies that it can be reformulated in a more abstract theory even though the problem definition is not in terms of the image of the appropriate representation map. In this case, it is necessary to derive an equivalent problem formulation stated in terms of the image of the representation map. This is in itself a reformulation or operationalization - the basic inference step is finding an equivalence. Given this equivalent problem formulation and the appropriate predefined implementation link, by applying the inverse representation map the abstract problem reformulation is obtained. Subsequent sections show how to generate a new implementation link when an appropriate one does not already exist in the knowledge base.

Abstracting through Predefined Implementation

1. Find properties of a problem.
2. Find an implementation link which uses some of these properties in the abstraction direction.
3. Find an equivalent problem formulation stated in terms of the image of the representation map.
4. Apply the inverse of the representation map to derive the abstract formulation.

These steps are illustrated with the reformulation of the arrow puzzle from lists of operators to bags of operators. In STEP 1. the homomorphism method generates

[3]This easily generalizes to the case where the behavior is defined as an arbitrary sublanguage of the problem domain theory[Lowry, 1087b]

several behavioral equivalence theorems including the comrautativity of append *with respect to the 10-behavior,*

$$SOLVE(\{SS,GS\}, append(Lop1, Lop2)) = SOLVE(\{SS,GS\}, append(Lop2, Lop1))$$

*In* STEP 2. the knowledge base is searched for an appropriate implementation link which incorporates the problem constraint found in step 1. For behavioral equivalence constraints, this means finding an implementation link with an abstraction function whose kernel is a subset of the behavioral equivalences found in step 1. The abstraction function for the bag to list implementation that maps bagunion to append is:

$abs(append(L1, L2)) = bagunion(abs(L1), abs(L2))$

Since bagunion is append with commutativity, the kernel of the abstraction function contains the behavioral equivalence of commutativity of append.

In STEP 3. the problem is reformulated to an expression in the target of the representation map. Since cons is in the target whereas car and cdr are not, the car/cdr recursion which defines the REDUCE operator is replaced by an existential construction using cons:

if *list = null* then *Reduce(binoplist) = identity*
if *list* $\neq$ **null then** $\exists x\ L\ cons(x, L) = list$ AND
*Rcduct(binop,list) = x binop Reduce(binop, L)*

in STEP 4. this new problem formulation is mapped through the inverse implementation link:

if *bag = nullbag* then *ReduceBag (binop, bag) = identity*
if *bag* $\neq$ *nullbag* then $\exists x\ B$ *addbag*($x_y B$) - *bag* AND
*Reduce(binop,bag) = x binop Rtduct(binop,B)*
*SolveBag({StartString. GoalString},OpBag) =*
*Apply(rcduceBag(XOROpbag), Start String) = GoalString*

## VI Making the World Safe for Equality

When a problem property does not correspond to a predefined implementation link, then for behavioral abstraction STRATA must generate a new representation theory which incorporates the property. From a mathematical viewpoint, the central issue*is how to turn an equivalence into an equality. In particular, a behavioral equivalence in the concrete theory is turned into an equality in the abstract theory. Equal objects are identical - they can be freely substituted for each other everywhere. Equivalent objects are not identical; by definition behaviorally equivalent objects can be substituted for each other in the IO-relation, but not necessarily elsewhere. *Only when behaviorally equivalent objects are made identical has the equivalence been incorporated into a new representation.* This is illustrated by the theories which are the abstract data types for lists, bags, and sets.

In order to turn a behavioral equivalence into an equality, a new representation language is generated, so that behaviorally equivalent objects are freely substitutable. The requirement of free substitution is the constraint used in the generator of the new representation language. This section discusses some of the conceptual background needed for generating the new representation language, further details can be found in [Lowry, 1987b]. A sort is PROTECTED against new congruences when a congruence would lead to identifying objects that are not behaviorally equivalent, A sort is SEPARABLE into separate copies *for* different functions if the problem definition does not link the different functions. This is the basis of the SPLITTING strategy. Finally, a function is SAFE with respect to a BEHAVIORAL EQUIVALENCE if propagating the equivalence through the function as an equality does not lead to defining new congruences on PROTECTED sorts. Deleting UNSAFE functions is the basis of the PROTECTION strategy.

Definition: A sort is PROTECTED iff:

Case 1 It is in the input or output of the IO-relation.

Case 2 It is a free parameter to the problem definition.

In the arrow puzzle, the following sorts are protected by case 1: lists of operators, StartString GoalString pairs. Case 2 protects the sort operators, because the arrow puzzle is defined with respect to any set of operators (bit strings which are XORed with the states). The basic idea behind a protected sort is that no congruence can be defined on a protected sort unless it is a behavioral equivalence with respect to the io-relation.

In general each sort is in the domain or range of many functions. Congruences are propagated through sorts that are in the domain of a function to the sort which is the range of a function. This range sort is in turn the input to other functions, and in this way a congruence is propagated through the daisy chain of functions in the problem domain theory. However, only some of this daisy chaining is intrinsic to the axiomatization and definitions of the functions in the problem domain theory. This leads to the definition of a SEPARABLE sort with respect to two functions:

Definition: A sort S is separable with respect to a function Fl whose range is S and a function F2 which has an input argument of sort S IFF Fl does not appear as part of a term in an expression whose head is F2 in the definition of the problem domain theory.

The intuition of this definition is that a SEPARABLE sort can be split into two sorts, thus blocking the propagation of a congruence from Fl to F2. Another viewpoint is that sort S is separable if the problem domain theory could have been given with the output of Fl being sort Si and the input of F2 being a distinct sort S2 (SI equal S2, but SI NOT eq S2). The mathematical machinery for this viewpoint is described at length in [Burstall and Goguen, 1977] and [Goguen and Burstall, 1985].

Definition: A function is SAFE iff:

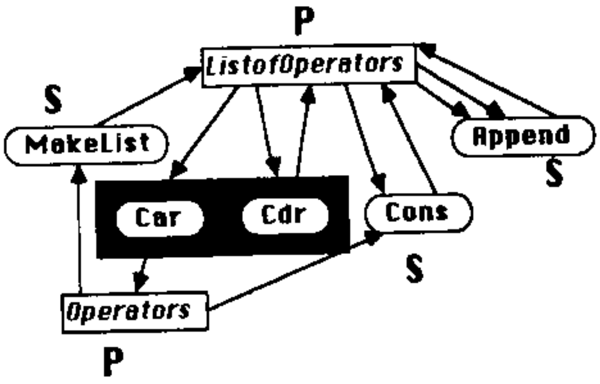
Case 1 *The range is the input or output sort of the io-relation.* Behaviorally equivalent inputs to the function result in behaviorally equivalent outputs.

Case 2 *The range is not the input or output sort.* Behaviorally equivalent inputs have identical outputs.

CAR and CDR are not SAFE with respect to commutativity of append, because their respective output sorts, operators and list of operators, are protected. However, CONS is safe because it is covered by case 1.

## VII Behavioral Congruence - Protection and Splitting

The *protection* and *splitting* methods transform the language of the problem domain theory in minimal ways so that adding the behavioral equivalence theorem as a new equality preserves the problem semantics. The goal is to make the representation map as invertible as possible, so that the problem definition needs to be only minimally reformulated before being lifted to the new abstract problem domain theory. The *protection method* deletes functions whose range is a protected sort which are not safe. The *splitting method* makes copies of separable sorts so that functions which would not be safe after propagating the behavioral equivalence as a congruence become safe.



The protection method is illustrated above for the case of transforming the behavioral commutativity of append into an equality. In the diagram, sorts are represented by boxes, and functions are represented by ovals. The inputs and outputs of a function are represented by arrows leading from/to the appropriate sorts. The protected sorts are *ListofOperators* and *Operators,* labeled with a P. The sort *ListofOperators* is used to define the behavioral equivalence theorems of commutativity and self-conjugate, any function which uses this sort as an input argument must be safe. The functions APPEND and CONS are safe, and labeled with an S. In contrast the functions CAR and CDR are not safe and are surrounded by black to indicate that they are deleted by the *protection method.* The function MakeList is safe because its input is the protected sort *operators* upon which no equality can be defined, as it is a free parameter.

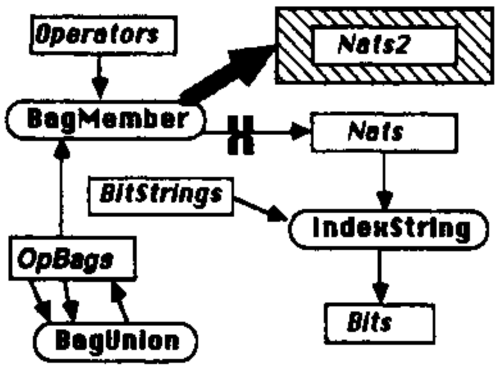
To derive the abstract theory from the concrete theory of lists with the behavioral equivalence of commutativity of append, the protection method renames the behavioral sorts and the safe functions which reference them, copies the axioms and then adds the behavioral equivalence as an equality. The representation map takes a renamed function in the abstract theory into the corresponding function in the original theory. The *protection method* generates the theory of bags shown in section 2, along with the corresponding implementation link from bags to lists. Note that behavioral abstraction must still be used to lift the problem definition SOLVE into the new theory, as explained in section 5.

The *protection method* could be applied once again for the behavioral equivalence of self-conjugate of append. Bag-Union (previously append) becomes symmetric set difference. Addbag (previously cons) is also transformed to XorAdd, its new action is to delete an element if it already exists, otherwise it adds the element.

An alternative derivation path which uses the behavioral equivalence of self-conjugate of append is given by the *splitting method.* In this case, the theory of bags is first isomorphically transformed to exponential notation, as discussed in [Lowry, 1987b]. In this isomorphic theory given below, equality of bags is defined in terms of bag-member. Bag-member is a function which takes an element and a bag and returns the number of occurrences of the element in the bag.

$$BagMember(x, NullBag) = 0$$
$$BagMember(x, MakeBag(x)) = 1$$
$$\text{If } y \neq x \text{ then } BagMember(y, MakeBag(x)) = 0$$
$$BagMember(x, BagUnion(B1, B2)) =$$
$$BagMember(x, B1) + BagMember(x, B2)$$
$$B1 = B2 \text{ iff } \forall x BagMember(x, B1) = BagMember(x, B2)$$

In this new isomorphic representation, bag equality is defined with the sort *natural numbers* through the function bag-member. In the arrow puzzle, the natural numbers are used both as the range of bag-member and as an index into bit-strings. Propagating the congruence of self-conjugate through the natural numbers causes them to collapse to Mod2 (isomorphic to the booleans). If this were then propagated to the index into bit-strings, it would cause a collapse of bit-strings into the even and odd indices. Unlike the elements of operator-lists, which are protected, the natural numbers are not protected and can be split. This prevents the propagation of the congruence through the range of bag-member to the index of bit-strings.



The *splitting method* is illustrated above, by the addition of the new sort *natural numberst* as the output of BagMember The output of BagMember is no longer the same sort as the input to IndexString The propagation of the behavioral equivalence of self-conjugate of bag-union through bag-member causes an additional axiom to be added to the sort natural numbers2, which results in a theory which is syntactically identical to the theory for Mod2 or the booleans. The derivation below shows how the additional axiom which defines Mod2 (or the booleans) is obtained by propagating the behavioral equivalence through substitution:

*BagUnion* $(S.S) \cong_{B.\cdot} NullBag$

*BagMcmber(x, BagUnion(S, S))* =

*BagMember(x, NullBag)*

*B agM ember (x,S)* + *BagMember*$(x,S) = 0$

$Y + Y = 0$

In the new theory BagUnion becomes Symmetric Set Difference and the behavioral equivalence of self-conjugate of BagUnion becomes an equality. BagMember becomes SetMember, and the sort naturalnumbers2 becomes the booleans through the addition of the equality $Y + Y = 0$.

## VIII Summary

*A good representation incorporates the problem constraints.* This paper has presented the abstraction implementation model of problem reformulation. The search space of abstract problem reformulations is formalized model theoretically as a lattice structure. Behavioral abstraction is a general method for incorporating a problem constraint into an abstract problem reformulation, by reversing an implementation link. Behavioral congruence is a method for *generating* new representations which incorporate behavioral equivalence constraints. The first step is to transform the representation language through protection and splitting so that all functions are safe. Then the behavioral equivalence constraint is made into an equality, and propagated throughout the new representation language to generate new constraints. The kernel method and the homomorphism method are meta-level inference techniques for efficiently generating behavioral equivalence constraints. These methods are being implemented in an automatic programming system called STRATA.

## IX Acknowledgment

This paper benefitted from the technical and editing help provided by Joseph Goguen, Laura Jones, and Douglas Smith. Discussions with the following people have also contributed to the development of these ideas: Phil Agre, Tom Binford, Bruce Buchanan, Raul Duran, Mike Genesereth, David MacAllester, Patricia Riddle, Jeff Schrager, David Smith, Steve Tappel, and Daniel Weld.

## References

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. *Machine Intelligence 3,* 1968.

[Burstall and Goguen, 1977] Rod M. Burstall and Joseph Goguen. Putting theories together to make specifications. In *IJCAI* 5, pages 1045-1058, 1977.

[Goguen and Meseguer, 1982] Joseph Goguen and Jose Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *ICALP,* Springer Verlag, 1982.

[Goguen *et* a/., 1978] Joseph Goguen, Jim Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology,* pages 80-149, Prentice-Hall, 1978.

[Goguen and Burstall, 1985] Joseph A. Goguen and Rod M. Burstall. *Institutions: Abstract Model Theory for Computer Science.* Technical Report CSLI- 85-30, CSLI, 1985.

[Korf, 1980] Richard E. Korf. Towards a model of representation change. *Artificial Intelligence,* 14(1), April 1980.

[Lowry, 1987a] Michael R. Lowry. Algorithm synthesis through problem reformulation. In *AAAI-81,* July 1987.

[Lowry, 1987b] Michael R. Lowry. *Algorithm Synthesis through Problem Reformulation.* PhD thesis, Stanford University, 1987.

[Mostow, 1983] Jack Mostow. Machine transformation of advice into a heuristic search procedure. In *Machine Learning, An Artificial Intelligence Approach,* chapter 12, Tioga Press, 1983.

[Smith, 1985] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence,* 27(1), September 1985.