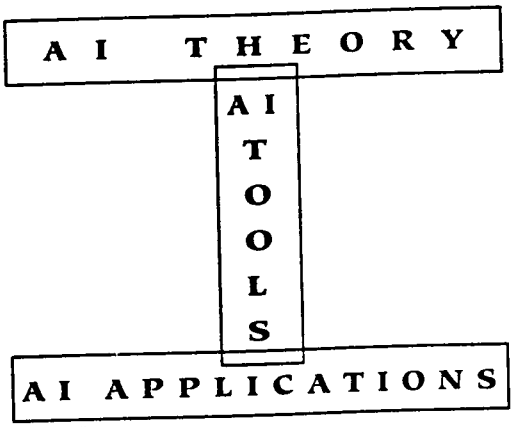
INTERNATIONAL JOURNAL ON

ARTIFICIAL INTELLIGENCE TOOLS

(Architectures, Languages, Algorithms)



VOLUME 4 NUMBERS 1 & 2 JUNE 1995

Editor-in-Chief
NIKOLAOS G. BOURBAKIS
Co-Editor-in-Chief
JEFFREY J. P. TSAI

World Scientific
Singapore • New Jersey • London • Hong Kong

International Journal of Artificial Intelligence Tools, Vol. 4, Nos. 1 & 2 (1995) 33–53
© World Scientific Publishing Company

A FRAMEWORK FOR HETEROGENEOUS ASSOCIATIVE LOGIC PROGRAMMING

ARVIND K. BANSAL

Department of Mathematics and Computer Science
Kent State University, Kent, OH 44242 - 0001, USA
E-mail: arvind@mcs.kent.edu

Received 10 March 1995
Revised 29 May 1995

ABSTRACT

Associative computation is characterized by seamless intertwining of search-by-content and data parallel computation. The search-by-content paradigm is natural to scalable high performance heterogeneous computing since the use of tagged data avoids the need for explicit addressing mechanisms. In this paper, the author presents an algebra for associative logic programming, an associative resolution scheme, and a generic framework of an associative abstract instruction set. The model is based on the integration of data alignment and the use of two types of bags: data element bags and filter bags of Boolean values to select and restrict computation on data elements. The use of filter bags integrated with data alignment reduces computation and data transfer overhead, and the use of tagged data reduces overhead of preparing data before data transmission. The abstract instruction set has been illustrated by an example. Performance results are presented for a simulation in a homogeneous address space.

Keywords: Artificial intelligence, Associative computing, Data-parallel computing, Heterogeneous computing, Parallel computing, Scalable high performance computing, Logic programming.

1. Introduction

Associative computation is characterized by seamless intertwining of search-by-content and data-parallel computation [15]. This intertwining facilitates the integration of logic programming paradigm and data-parallel computations on a heterogeneous collection of massive parallel machines. This work generalizes the associative computation model which started with our earlier work to exploit associative computation on SIMD architectures [1,2].

Heterogeneous collections of machines have different address spaces, and there is no uniform mechanism to handle this heterogeneity. Current popular approaches to handle heterogeneity [14,16] use a message passing paradigm for information transfer. However, structured data, if transferred across a heterogeneous address space, has to be linearized (unpacked) in one address space, and restructured

(packed) in the other:

1. The unpacking algorithm goes through a chain of references to linearize the data,
2. The packing algorithm has to build up a chain of references, and
3. The implicit attributes have to be tagged during the message transfer.

In order to reduce this overhead, the associative computing paradigm maximizes the use of sedentary data, uses duplicate copies of tagged data, manipulates and transfers bit-vectors, and exploits intertwining of the search-by-content paradigm and data-parallel computation. The motivation for this paper is to formalize the rules for an associative computation model of logic programming, and develop a framework of an associative abstract instruction set which will facilitate implementation of the associative model of logic programming on a heterogeneous collection of massive parallel architectures. The major contribution in this paper is the extension of associative computation formalism to resolution process of logic programming, and description of a framework of associative abstract instruction set. This framework is the basis for our effort to develop heterogeneous abstract machines suitable for a heterogeneous collection of high performance machines having a heterogeneous address space.

The associative computational model has direct applications for a large class of problems which require the integration of high performance symbolic computing, knowledge retrieval, and scientific computation. Some examples are geographical knowledge bases, genome sequence analysis, and intelligent simulation of complex engine processes.

The rest of the paper is as follows. Section 2 describes the necessary terminology for logic programming, notations, a generic architecture for which the model is applicable, and the abstract data representations used in the associative computation model. Section 3 describes an algebra for associative computation and its application to data-parallel computation on distributed data. Section 4 describes an extended algebra for associative logic programming, a framework of associative model of logic programming, and an associative resolution strategy. Section 5 describes a generic description of an abstract instruction set. Section 6 illustrates the abstract instruction set using an example. Section 7 describes the performance evaluation of a simulation of the framework in homogeneous address space. The last section concludes the paper.

2. Preliminaries

2.1. *Notations*

In this paper, the author will denote a bag or a set within curly brackets $\{ \ldots \}$, a tuple within angular brackets $< \ldots >$, implication by $\Rightarrow$, equivalence by $\equiv$, membership by $\in$, and complement by $\neg$. A subscripted definition of the form

$definition|_{condition}$ will be used to specify a definition and the required conditions. An empty set is represented by ϕ. The upper case letters (possibly subscripted) X, Y, Z denote variables, the lower case letters (possibly subscripted) t, u, v, w denote values, the representations X/t or Y/w represent variable bindings, the lower case Greek letters σ, θ, δ represent a set of variable bindings, the upper case Greek letters Σ, Θ denote a bag of set of bindings.

2.2. *Logic programming terminology*

The basic definitions of logic programming are available in [11,12]. A term is a variable, a constant, or an n-ary function $f(t_1, ..., t_N)$ such that $t_{I(1\leq I\leq N)}$ is a term. An atom is n-ary predicate of the form $p(t_1, ..., t_N)$. A literal is an atom or a negation of an atom.

A logic program is a set of Horn clauses of the form $A :- B_1, ..., B_N$ ($N \geq 0$). Facts are clauses with empty clause-bodies. Rules are clauses with non-empty clause-bodies. Successful solution of a logical query is based on repeated reduction of a goal to conjunction of subgoals using unification – a combination of a pattern matching and a value binding process – until the goal can be unified with one of the facts.

A *multiple-occurrence variable* occurs more than once in a literal. A *shared variable* occurs in two or more subgoals of a clause. A *producer* is the first occurrence of a shared variable, and generates a bag of values. A *consumer* is an occurrence of a shared variable other than the producer. There can be only one producer. However, there can be multiple consumers. *Aliased variables* share the same value: binding one of the aliased variables automatically binds others with the same value. Aliasing, denoted by the symbol $\simeq$, is an equivalence relationship.

A *simple fact* has no multiple occurrence variables, and a *complex fact* has at least one multiple occurrence variable. A complex clause is either a complex fact, or a clause with a non-empty clause-body.

A substitution is of the form $\{X_1/t_1, ..., X_N/t_N\}$. Given a substitution θ and a logical term t, the application of θ on t is denoted by $t\,\theta$. For example, substitution $\{X/d\}$ when applied on the term $a(X)$ will derive the term $a(d)$. Composition of two substitutions $\sigma_1 = \{X_1/t_1, ..., X_N/t_N\}$ and $\sigma_2 = \{Y_1/w_1, ..., Y_M/w_M\}$, denoted by $\sigma_1 \bullet \sigma_2$ is defined as $\{X_I/(t_I\sigma_2 \sqcap w_K)\ |_{X_{I(1\leq I\leq N)}=Y_{K(1\leq K\leq M)}}, X_J/t_J\sigma_2\ |_{X_J\notin\{Y_1,...,Y_M\}}, Y_L/w_L\ |_{Y_L\notin\{X_1,...,X_N\}}\}$ where $\sqcap$ denotes glb(greatest lower bound) of two values. Two substitutions $\{X_1/t_1, ..., X_N/t_N\}$ and $\{Y_1/w_1, ..., Y_M/w_M\}$ are *strongly-disjoint* if $X_{I(1\leq I\leq N)} \notin \{Y_1, ..., Y_M\}$ and $Y_{J(1\leq J\leq M)} \notin \{X_1, ..., X_N\}$ and $t_{I(1\leq I\leq N)} \notin \{Y_1, ..., Y_M\}$ and $w_J(1 \leq J \leq M) \notin \{X_1, ..., X_N\}$. The composition of two strongly-disjoint substitutions is the same as the union of the substitutions: due to the single occurrence of variables no extra bindings are added during composition. For example, composition of $\{X/4, Y/5\}$ and $\{Z/6\}$ derives $\{X/4, Y/5, Z/6\}$. Two substitutions $\{X_1/t_1, ..., X_N/t_N\}$ and $\{Y_1/w_1, ..., Y_M/w_M\}$ are *loosely-disjoint* if $t_{I(1\leq I\leq N)} \notin \{Y_1, ..., Y_M\}$ and $w_{J(1\leq J\leq M)} \notin \{X_1, ..., X_N\}$, and there exists at least

one $X_I(1 \leq I \leq N)$ which is equal to $Y_J(1 \leq J \leq M)$. The composition of loosely-disjoint substitutions replaces the bindings for common variables by the greatest lower bound of the two bindings. For example, composition of $\{X/4,\ Y/4\}$ and $\{X/c,\ Z/5\}$ derives $\{X/\perp,\ Y/4,\ Z/5\}$. This composition is not simple union since the bindings X/c has been replaced by $X/\perp$. Two substitutions are are non-disjoint if there exists at least one $t_{I(1\leq I\leq N)} \in \{Y_1, ..., Y_M\}$ and $w_{J(1\leq J\leq M)} \in \{X_1, ..., X_N\}$. For non-disjoint substitutions, extra bindings are added due to transitivity. For example, composition of $\{X/Y,\ Z/4\}$ and $\{Y/5\}$ derives $\{X/5,\ Y/5,\ Z/4\}$. Bindings caused by transitivity cause sequentiality in data parallel computation since two bindings may be stored in separate address spaces.

2.3. *Associative computation*

Associative computation has two major components: search-by-content and data-parallel computation. The use of search-by-content facilitates the use of heterogeneous address spaces since no explicit indexing scheme is needed to address a data element. Consequently, data is distributed across different address spaces, and different processors exploit data-parallelism while working simultaneously on different data elements using the same abstract computation.

A generic architecture on which an associative computation model can be implemented is an ordered collection of *processing cells*: each cell is a quadruple $<C_i, R_i, S_i, M_i>$ where C_i denotes a processing element (PE), R_i denotes a set of local registers, S_i denotes local storage, and M_i denotes a mask-bit. An associative search of a field for a specific value sets up a bag of Boolean results which are used to filter an abstract computation on different data elements of a bag. An instruction is broadcast to each cell simultaneously to initiate the same abstract computation[1] on different data elements. A SIMD architecture satisfies this criteria. However, the major difference between SIMD and other generic architectures is that the SIMD architecture supports fine grain synchronization for every data-parallel computation. Our model does not impose the restriction of low level synchronization. However, relaxation of this restriction causes increased overhead to resynchronize the computation in non-SIMD architectures.

2.4. *Abstract data representation*

A bag is a collection of data items with multiple possible occurrences of a value [13]. Two bags are associated if the individual elements in the bags are index-wise paired. The advantage of pairing is that the knowledge of an element in one bag is sufficient to derive the value of the corresponding element (having same index) in the other bag. The author defines two types of data structures: a D-bag to hold data elements and an F-bag to select D-bags and perform computations on the selected data elements.

[1] an abstract computation may have multiple definitions with different a sequence of commands

A *D-bag*, denoted by $\mathcal{D}$, is defined as an ordered bag which includes null value $\perp$ and top value $\top$. $\perp$ denotes the absence of any value, and $\top$ denotes an undefined value. $\top$ can be instantiated with any value. For example, $\{2, \perp, 3\}$ is a D-bag. However, $\{2, \perp, 3\} \neq \{\perp, 2, 3\}$ since D-bags are ordered. The null element $\perp \preceq$ every element in the D-bag. A D-bag is denoted as ϕ^D if all the elements in the D-bag are equal to $\perp$. A new element in a D-bag is inserted at the end of the D-bag. The insertion in D-bag is denoted by $\uplus$. For example, $<4, \perp, 6> \uplus 7$ derives a D-bag $<4, \perp, 6, 7>$. A D-bag is mapped on the generic architecture such that by accessing the same index on every S_i — the local storage on the architecture — all the elements of a D-bag are accessed from different cells simultaneously.

An F-bag is a D-bag with Boolean values *true* and *false*. The Boolean values *true* and *false* are treated synonymously with the values "1" and "0" respectively, and *False* (or "0") $\prec$ *true* (or "1"). An F-bag of *1s* is denoted by $\mathcal{F}^1$, and an F-bag of *0s* is denoted by $\mathcal{F}^0$.

A D-bag of M-tuples is stored as an *association* of M D-bags index-wise aligned with each other such that accessing the I_{th} element of one bag also gives access to the I_{th} element of the other D-bags. The aligned D-bags are denoted as $\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus \ldots \bigoplus \mathcal{D}_M$.

The author also defines the notion of D-inclusion, D-equality, D-union, and D-intersection of D-subbags. These notions are different from the set-theoretic counterparts due to the presence of order in D-bags and F-bags, inclusion of $\perp$ and $\top$ in D-bags, and pairwise comparison of the corresponding elements. A D-bag $\mathcal{D}_1 = \{d_{11}, \ldots, d_{1N}\}$ is included in another D-bag $\mathcal{D}_2 = \{d_{21}, \ldots, d_{2N}\}$ if $\forall I_{(1 \leq I \leq N)}$ $d_{1I} \preceq d_{2I}$. For the sake of clarity, the author refers to the inclusion of a D-bag by D-inclusion, and denotes D-inclusion by $\sqsubseteq$. For example, $\{4, \perp, 5, 6\} \sqsubseteq \{4, 3, 5, 6\}$ since $\perp \prec 3$. D-union of two D-subbags $\{d_{11}, \ldots, d_{1N}\}$ and $\{d_{21}, \ldots, d_{2N}\}$ derives a new D-bag $\{d_{31}, \ldots, d_{3N}\}$ such that $\forall I_{(1 \leq I \leq N)}$ $d_{3I} = d_{1I}$ if $d_{2I} \preceq d_{1I}$, and $d_{3I} = d_{2I}$ if $d_{1I} \preceq d_{2I}$. The author denotes D-union by $\bigsqcup$. For example, $\{\perp, b, c\} \bigsqcup \{a, b, \perp\}$ derives $\{a, b, c\}$. D-intersection, denoted by $\bigsqcap$, of two D-subbags $\{d_{11}, \ldots, d_{1N}\}$ and $\{d_{21}, \ldots, d_{2N}\}$ derives a new D-bag such that $\forall I_{(1 \leq I \leq N)}$ $d_{3I} = d_{1I}$ if $d_{1I} \preceq d_{2I}$, and $d_{3I} = d_{2I}$ if $d_{2I} \preceq d_{1I}$. For example, $\{2, 3, \perp\} \bigsqcap \{\perp, 3, 4\}$ derives the D-subbag $\{\perp, 3, \perp\}$. D-equality, denoted by $\doteq$, of two D-bags $\mathcal{D}_1$ and $\mathcal{D}_2$ derives an F-bag $\mathcal{F}$ such that $\forall I_{1 \leq I \leq N}$ d_{3I} $(\in \mathcal{F}) = 0$ if $d_{1I} \neq d_{2I}$ or $d_{1I} = \perp$ or $d_{2I} = \perp$; $d_{3I} = 1$ if $d_{1I} = d_{2I}$ and $d_{1I} \neq \perp$ and $d_{2I} \neq \perp$. For example, $\{a, b, \perp\} \doteq \{a, c, \perp\}$ derives the F-bag $\{1, 0, 0\}$.

The application of an F-bag on a D-bag is denoted by $\mathcal{D} \bigotimes \mathcal{F}$. Under the assumption *false* $\prec$ *true*, D-union of F-bags is implemented by a logical-OR of the corresponding logical bit-vectors, and D-intersection of F-bags is implemented by a logical-AND of the corresponding logical bit-vectors.

3. An Algebra for Associative Computation

Our model is based upon eighteen rules of associative computation. There are five

types of laws of associative computation: the *laws of data association, the laws of associative search, the laws of selection, the laws of data-parallel computations* and *the laws of associative update*. In this section, we describe an algebra for associative computation and its application in heterogeneous associative computing.

3.1. *Laws of data association*

This subsection describes three rules for the association of data. Rule (1) describes the formation of an association, and projection of information from an association of D-bags. Rule (2) describes the isomorphism of nested associations, and Rule (3) describes the isomorphism under permutation of an association. The implication of isomorphism of association is that the information in an association is not altered by nesting or by permutation.

Rule 1A: Construction states that a D-bag of M-tuples is given by the association of M D-bags such that corresponding elements are aligned by index. For example, $\{a, 2, 3, 4\} \bigoplus \{b, 5, 6, 7\}$ is equivalent to $\{<a, b>, <2, 5>, <3, 6>, <4, 7>\}$.

Rule 1B: Projection a tuple can be retrieved from an association $\mathcal{D}_1, \ldots, \mathcal{D}_N$ in constant time by data parallel projection of the corresponding element from each D-bag. For example, projection of the second element from the association $\{a, 2, 3, 4\} \bigoplus \{b, 5, 6, 7\}$ derives tuple $<2, 5>$.

Rule 2: Isomorphism in nesting states that nested associations are isomorphic For example, $\{1, 2\} \bigoplus (\{3, 4\} \bigoplus \{5, 6\})$ is isomorphic to $(\{1, 2\} \bigoplus \{3, 4\}) \bigoplus \{5, 6\}$, and both are isomorphic to $\{<1, 3, 5>, <2, 4, 6>\}$.

Rule 3: Isomorphism in permutation states that permutations on an association is isomorphic: a pair $(x, y) \in \mathcal{D}_1 \bigoplus \mathcal{D}_2$ (such that $x \in \mathcal{D}_1$ and $y \in \mathcal{D}_2$) has a bijective mapping to $(y, x) \in \mathcal{D}_2 \bigoplus \mathcal{D}_1$.

3.2. *Laws of associative search*

This subsection describes two rules for associative search. Rule (4) describes the mapping of a D-bag to an F-bag based upon an associative search of an element, and Rule (5) describes the derivation of information from D-bags. Rule (4) is necessary for data-parallel goal reduction, and Rule (5) is necessary for deriving the bindings of variables in a goal after unification.

Rule 4: Membership states that associative search of a data element d in a D-bag D_1 (of the form $<d_1, \ldots, d_N>$ derives an F-bag $\mathcal{F}$ such that if $d_j = d$ then the corresponding element in $\mathcal{F}$ is "1" otherwise "0". For example, associative search of an element 4 in the D-bag $\{3, 5, 4, 7, 4, 9\}$ derives an F-bag $\{0, 0, 1, 0, 1, 0\}$.

Rule 5: Search states that by associatively searching in one field, the associated data elements in the other field can be extracted. For example, associative search

for tuple $\{4, \top, \top\}$ in the D-bag $\{<4,5,6>, <3, 7, 9>, <4, 9, 10>\}$ derives an F-bag $\{1, 0, 1\}$ which, when applied on the D-bag, derives a D-subbag $\{<4, 5, 6>, \bot, <4,9,10>\}$.

3.3. *Laws of selection*

There are six rules for selection of data elements from an association using F-bags. Computation on F-bags reduces the computation overhead when selecting D-subbags, and transmission of F-bags across heterogeneous address spaces reduces the overhead of moving large D-subbags. Rule (6) is used to select the data elements from tuples; Rule (7) reduces the overhead of creation of D-subbags. Rule (8) – Rule (11) reduce the data movement during computations on the associations if the corresponding D-bags are distributed on heterogeneous address spaces; Rules (12) and (13) have been used to derive the properties of many low level computations.

Rule 6: Selection states that the association of an F-bag with a D-bag selects the data elements whenever the corresponding element in the F-bag is *1*. For example, $\{3, 5, 6\} \otimes \{0, 1, 0\}$ derives $\{\bot, 5, \bot\}$.

Rule 7: Monotonicity in Selection states a D-subbag $\mathcal{D} \otimes \mathcal{F}_1$ includes the D-subbag $\mathcal{D} \otimes \mathcal{F}_2$ if $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$. For example, $\{5, 6, 7\} \otimes \{1, 0, 1\}$ derives the D-bag $\{5, \bot, 6\}$. While $\{5, 6, 7\} \otimes \{1, 0, 0\}$ derives the D-bag $\{5, \bot, \bot\}$.

Rule 8: Distributivity of association states that the D-subbag derived by the application of an F-bag on an association of D-bags is the same as the association of D-subbags derived by the application of the F-bag on the individual D-bags. For example, $(\{4, 5, 6\} \otimes \{1, 0, 1\}) \oplus (\{a, b, c\} \otimes \{1, 0, 1\}) \Rightarrow \{4, \bot, 6\} \oplus \{a, \bot, c\} \Rightarrow \{<4, a>, \bot, <6, c>\}$; and $(\{4, 5, 6\} \oplus \{a, b, c\}) \otimes \{1, 0, 1\}) \Rightarrow \{<4, a>, <5, b>, <6, c>\} \otimes \{1, 0, 1\} \Rightarrow \{<4, a>, \bot, <6, c>\}$.

Rule 9: Distributivity of D-union states that the data elements selected by the D-union of two F-bags is the same as result of selecting the data elements by applying individual F-bags on the D-bags and then performing D-union on the resulting D-subbags. For example, $\{2, 3, 4\} \otimes (\{1, 0, 0\} \bigsqcup \{0, 1, 0\}) \Rightarrow \{2, 3, 4\} \otimes \{1, 1, 0\} \Rightarrow \{2, 3, \bot\}$; and $(\{2, 3, 4\} \otimes \{1, 0, 0\}) \bigsqcup (\{2, 3, 4\} \otimes \{0, 1, 0\}) \Rightarrow \{2, \bot, \bot\} \bigsqcup \{\bot, 3, \bot\} \Rightarrow \{2, 3, \bot\}$.

Rule 10: Distributivity of D-intersection states that the data elements of a D-bag selected by D-intersection of two different F-bags is the same as the result of selecting the data elements by applying the individual F-bags on the D-bags and then performing the D-intersection on the resulting D-subbags. For example, $\{2, 3, 4\} \otimes (\{1, 1, 0\} \sqcap \{0, 1, 1\}) \Rightarrow \{2, 3, 4\} \otimes \{0, 1, 0\} \Rightarrow \{\bot, 3, \bot\}$; and $(\{2, 3, 4\} \otimes \{1, 1, 0\}) \sqcap (\{2, 3, 4\} \otimes \{0, 1, 1\}) \Rightarrow \{2, 3, \bot\} \sqcap \{\bot, 3, 4\} \Rightarrow \{\bot, 3, \bot\}$.

Rule 11: Distributivity of D-equality states that the F-bag derived by the D-equality of two aligned D-bags $\mathcal{D}_{11} \oplus \ldots \oplus \mathcal{D}_{1M}$ and $\mathcal{D}_{21} \oplus \ldots \mathcal{D}_{2M}$ is equivalent

to $(\mathcal{D}_{11} \doteq \mathcal{D}_{21}) \sqcap \ldots \sqcap (\mathcal{D}_{1M} \doteq \mathcal{D}_{2M})$. For example, $\{\perp, <2, 3>, <5, 6>\} \doteq \{\perp, <2, 3>, <6, 6>\} \equiv (\{\perp, 2, 5\} \doteq \{\perp, 2, 6\}) \sqcap (\{\perp^2, 3, 6\} \doteq \{\perp, 3, 6\}) \Rightarrow \{0, 1, 0\} \sqcap \{0, 1, 1\} \Rightarrow 0, 1, 0$.

Rule 12: Identity states that the Cartesian product of a bag $\mathcal{D}$ with *true* is equivalent to the set obtained by applying $\mathcal{F}^1$ to a D-bag, and is equivalent to $\mathcal{D}$ itself. For example, $\{2, 3, 4\} \times \{true\} \equiv \{2, 3, 4\} \bigotimes \{1, 1, 1\} \Rightarrow$ D-bag $\{2, 3, 4\}$.

Rule 13A: Zero states that the Cartesian product $\mathcal{D} \times \{\text{false}\}$ is equivalent to $\mathcal{D} \bigotimes \mathcal{F}^0 \Rightarrow \phi^D$.

Rule 13B: Zero states that the presence of a $\perp$ in a tuple makes the tuple as $\perp$. For example, a tuple $<\perp, 4, 5>$ is treated as $\perp$.

3.4. *Laws of data-parallel computation*

This subsection describes two rules for data-parallel computation:

Rule 14: Vector-vector $\Rightarrow$ vector states that the data-parallel computation on an association of D-bags is equivalent to performing the same abstract computation on every element of the associated fields. Any computation on $\perp$ derives $\perp$. For example, $\{2, 3, \perp\} *^D \{3, 4, \perp\} \Rightarrow \{6, 12, \perp\}$.

Rule 15: Scalar-vector $\Rightarrow$ vector states that the data-parallel computation of a scalar value *Val* with a D-bag is equivalent to taking Cartesian product of the singleton set $\{Val\}$ with $\mathcal{F}^1$, and performing data-parallel computation on the association $(\{Val\} \times \mathcal{F}^1) \doteq \mathcal{D}_1$. For example, $4 * \{2, 3, 4\} \equiv \{4, 4, 4\} *^D \{2, 3, 4\} \Rightarrow \{8, 12, 16\}$.

Rule 16: Vector $\Rightarrow$ vector states that application of a function on a D-bag derives another D-bag in a data-parallel manner. For example, sqaure($\{4, 5, 9\}$ derives a D-bag $\{16, 25, 81\}$.

3.5. *Laws of associative update*

There are two rules for associative update: Rule (17) describes associative insertion of a tuple in an association, and Rule (18) describes data-parallel release of tuples from an association.

Rule 17: Associative insert states that if a tuple of the form $<d_1, \ldots, d_N>$ is inserted in an association of D-bags $\mathcal{D}_1 \bigoplus \ldots \bigoplus \mathcal{D}_N$, then the association is updated to $(\mathcal{D}_1^U \bigoplus \ldots \bigoplus \mathcal{D}_N^U)$ where $\mathcal{D}_I^U$ denotes the updated bag. Each $\mathcal{D}_I^U$ is equal to $\mathcal{D}_I \uplus \{d_I\}$.

Rule 18: Associative delete states that by associatively searching in one field, the associated data elements in the other field are released in a constant number of

[2] $\perp$ is equivalent to $<\perp, \perp>$ see rule 13b

computations. For example, associative search for a tuple $\{4, \top, \top\}$ from a D-bag $\{<4, 5, 6>, <3, 7, 9>, <4, 9, 10>\}$ derives an F-bag $\{1, 0, 1\}$; complement of the F-bag $\{1, 0, 1\}$ derives an F-bag $\{0, 1, 0\}$. Application of the F-bag $\{0, 1, 0\}$ on the association derives the D-subbag $\{\bot, <3, 7, 9>, \bot\}$: all the tuples containing 4 in the first field are deleted.

3.6. *Extending associative rules for logic programming*

In this section, the algebra of associative computing has been extended to include both constants and variables; and to exploit associative computation for the variable binding, the goal reduction, the identification of unifiable clauses, and the selection of unifiable clauses. A D-bag of substitutions is referred as D-substitution, and binding of a variable to a D-bag is referred as D-binding. An element in a *D-substitution* is of the form $\bot$, $\top$, X_I/t_I (t_I is a singleton), $\{X_{I1}/t_{I1}, ..., X_{IN}/t_{IN}\}$ ($N \geq 1$) where t_{IJ} (or t_I) is either a constant symbol, a variable, or a D-bag of constants, a $\bot$, or a $\top$. For a singleton binding N is equal to 1.

Rule 19: Logical D-binding states that the associative match of a variable X with a D-bag derives an F-bag $\mathcal{F}^1$ and a D-binding $X/\{d_1, ... d_N\}$ which is equivalent to the D-substitution $\{X/d_1, ..., X/d_N\}$

Rule 20: D-application states that the D-application of a D-bag of substitutions $\Sigma^D = \{\sigma_1, ... \sigma_N\}$ on a D-bag $\mathcal{D} = \{d_1, ..., d_N\}$ is equivalent to $\{d_1\sigma_1, ..., d_2\sigma_N\}$ where each $d_I\sigma_I$ is a standard application with the difference that the application of $\bot$ derives $\bot$, and application of $\top$ does not alter the value. D-application will be denoted by $\circ$ for convenience. For example, $\{X, Y, a, c\} \circ \{X/1, W/4, \top, \bot\}$ derives $\{1, Y, a, \bot\}$.

Rule 21: Logical D-search states that the associative search of a given constant in a D-bag (containing constants and variables) $\{d_1, ..., d_N\}$ derives an F-bag $\{f_1, ... f_N\}$ and a D-substitution $\{\sigma_1^D, ... \sigma_N^D\}$ in Table 1.

Table 1. Definition of logical D-search

f_I	σ_I^D	Condition
1	$\top$	if d_I is not a variable and ($d_I \sqcap$ the given constant) $\neq \bot$
1	$\{d_I/$the constant value$\}$	if d_I is a variable
0	$\bot$	otherwise

For example, associative search of a value c with the D-bag $\{X, Y, c, b\}$ derives an F-bag $\{1, 1, 1, 0\}$ and a D-substitution $\{X/d, Y/d, \top, \bot\}$.

Rule 22: Logical D-equality states that the logical d-equality, denoted by $\cong^D$, of two associated D-bags $\mathcal{D}_1$ and $\mathcal{D}_2$ (both containing constants and variables) derives a D-substitution $\Sigma^D = \{\sigma_1^D, \ldots, \sigma_N^D\}$, an F-bag $\mathcal{F} = \{f_1, \ldots, f_N\}$, and a D-bag $\mathcal{D}_3$ such that $\mathcal{F}$ is equal to $(\mathcal{D}_1 \circ \Sigma) \doteq (\mathcal{D}_2 \circ \Sigma)$, and $\mathcal{D}_3$ is equal to $(\mathcal{D}_1 \circ \Sigma) \bigotimes \mathcal{F}$ or $(\mathcal{D}_2 \circ \Sigma) \bigotimes \mathcal{F}$. The D-bags $(\mathcal{D}_1 \circ \Sigma) \bigotimes \mathcal{F}$ and $(\mathcal{D}_2 \circ \Sigma) \bigotimes \mathcal{F}$ are equal. Each element $d_{3I(1 \le I \le N)}$ is given by $d_{1I} \simeq d_{2I}$ where $\simeq$ denotes logical equality of two data elements. Each $d_{3I} = d_{1I} \simeq d_{2I}$ is defined in Table 2.

Table 2. Definitions for logical D-equality

f_I	σ_I^D	d_{3I}	Condition
1	$\top$	$(d_{1I} \sqcap d_{2I})$	if $(d_{1I} \sqcap d_{2I}) \neq \perp$ and d_{1I}, d_{2I} are constants
1	$\{d_{1I}/d_{2I}\}$	d_{2I}	if $d_{1I} \not\simeq d_{2I}$ and d_{1I} is a variable
1	$\{d_{2I}/d_{1I}\}$	$d_{3I} = d_{1I}$	if $d_{1I} \not\simeq d_{2I}$ and d_{2I} is a variable
1	$\{d_{1I}/d_{2I}\}$ or $\{d_{2I}/d_{1I}\}$	d_{2I} or d_{1I}	if $d_{1I} \simeq d_{2I}$
0	$\sigma_I^D = \perp$	$\perp$	otherwise

For example, $\{a, b, Z\} \cong^D \{c, b, W\}$ derives an F-bag $\{0, 1, 1\}$, a D-substitution $\{\perp, \top, W/Z\}$, and a D-bag as $\{\perp, b, Z\}$. Also, $\{a, b, Z\} \circ \{\perp, \top, W/Z\} \Rightarrow \{\perp, b, Z\}$. Alternately, $\{c, b, W\} \circ \{\perp, \top, W/Z\} \Rightarrow \{\perp, b, Z\}$.

3.7. *Data-parallel composition*

The data parallel composition of two D-substitutions ${\Sigma^D}_1$ (of the form $\{\sigma_{11}^D, \ldots, \sigma_{1I}^D, \ldots, \sigma_{1N}^D\}$) and ${\Sigma^D}_2$ (of the form $\{\sigma_{21}^D, \ldots, \sigma_{2I}^D, \ldots, \sigma_{2N}^D\}$), denoted by ${\Sigma^D}_1 \bullet^D {\Sigma^D}_2$, derives a D-bag $\{\sigma_{11}^D \bullet \sigma_{21}^D, \ldots, \sigma_{1I}^D \bullet \sigma_{2I}^D, \ldots, \sigma_{1N}^D \bullet \sigma_{2N}^D\}$. The composition of individual elements is derived by $\bullet$. For convenience, the variables in σ_{1I}^D are denoted by X_{IJ}, the bindings in σ_{1I}^D are denoted by X_{IJ}/t_{IJ}, the variables in σ_{2I}^D are denoted by Y_{IJ}, and bindings in σ_{2I}^D are denoted by Y_{IK}/w_{IK}. In the definition, the notation $X_I/(t_I \cong w_K)$ generates a D-substitution of the form δ_I such that $\delta_I = X_I/(t_I \sqcap w_I)$ if t_I and w_I are not variables; $\delta_I = \{X_I/t_I, t_I/w_I\}$ if t_I is a variable; $\delta_I = \{X_I/w_I, w_I/t_I\}$ if w_I is a variable; and $\delta_I = X_I/\perp$ otherwise. Each $\sigma_{1I}^D \bullet \sigma_{2I}^D$ is:

The definitions (i) to (v) derive composition of singleton substitution; and the definitions (vi) to (viii) derive the composition of the set of substitutions. Definition (i) states that application of $\perp$ derives $\perp$. Definitions (ii) and (iii) state that the application of $\top$ will not affect the binding. Definition (iv) states that in the presence of multiple occurrence variable and the absence of aliased variable the logical D-equality of the D-bindings derives the substitution. Definition (v,

states that in the presence of aliased variables, two variables are bound to the same value; Definition (vi) combines two strongly-disjoint substitutions. The disjoint substitutions avoid sequentiality caused by transitivity and data transfer overhead of membership tests $X_{I(1\leq I\leq N)} \in \{ Y_1, ..., Y_M\}$, and $Y_{J(1\leq J\leq M)} \in \{X_1, ..., X_M\}$ and $t_{I(1\leq I\leq N)} \in \{Y_1, ..., Y_M\}$ and $w_{J(1\leq I\leq M)} \in \{X_1, ..., X_N\}$. Definition (vii) combines two loosely-disjoint substitutions, and avoids the sequentiality and data transfer overhead caused by transitivity but suffers from data transfer overhead for the membership tests $X_{I(1\leq I\leq N)} \in \{Y_1, ..., Y_M\}$, and $Y_{J(1\leq J\leq M)} \in \{X_1, ..., X_M\}$. Definition (viii) suffers from both the overheads. For example, D-composition of two D-substitutions $\{\perp, \top, X_3/a, \{X_{41}/ c, X_{42}/d\}, \{X_{51}/P, X_{52}/ c\}\}$ and $\{Y_1/a, Y_2/b, X_3/b, Y_{41}/f,$ P/d$\}$ derives a D-substitution $\{\perp, Y_2/b, X_3/\perp, \{X_{41}/c, X_{42}/ d, Y_{41}/f\}, \{X_{51}/d, P/d, X_{52}/c\}\}$. Note that the last substitutions in both the D-substitutions introduce a new binding X_{51}/d which causes sequentiality due to transitivity.

Table 3. Definition of data-parallel composition

No.	$\sigma^D_{1I} \bullet \sigma^D_{2I}$	Condition
(i)	$\perp$	if $\sigma^D_{1I} = \perp$ or $\sigma^D_{2I} = \perp$
(ii)	σ^D_{1I}	if $\sigma^D_{2I} = \top$
(iii)	σ^D_{2I}	if $\sigma^D_{1I} = \top$
(iv)	$X_I/(t_I \cong w_I)$	if $X_I = Y_I$ and $t_I \not\cong w_I$.
(v)	$\{X_I/t_I, Y_I/t_I, t_I/w_I\}$	if $t_I \simeq w_I$
(vi)	$\sigma^D_{1I} \bigcup \sigma^D_{2I}$	if σ^D_{1I} and σ^D_{2I} are strongly disjoint
(vii)	$\{X_{IJ}/(t_{IJ} \cong w_{IK}) \mid_{X_{IJ(1\leq J\leq N)}=Y_{IK(1\leq K\leq M)}},$ $X_{IJ}/t_{IJ} \mid_{X_{IJ}\notin\{Y_{I1},...,Y_{IM}\}},$ $Y_{IK}/w_{IK} \mid_{Y_{IK}\notin\{X_{I1},...,X_{IN}\}}\}$	if σ_{1I}, σ_{2I} are loosely-disjoint
(viii)	$\{X_{IJ}/(t_{IJ}\sigma_{2I} \sqcap w_{IK}) \mid_{X_{IJ(1\leq J\leq N)}=Y_{IK(1\leq K\leq M)}},$ $X_{IJ}/t_{IJ}\sigma_{2K} \mid_{X_{IJ}\notin\{Y_{I1},...,Y_{IM}\}},$ $Y_{IK}/w_{IK} \mid_{Y_{IK}\notin\{X_{I1},...,X_{IN}\}}\}$	otherwise

4. Associative Model of Logic Programming

In this section, the author describes the program representation, and describes a framework for heterogeneous associative logic programming.

The associative computation model maps a logic program as a pair of associations of the form $<\mathcal{L} \bigoplus \mathcal{N} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus ... \bigoplus \mathcal{A}_N, \mathcal{L} \bigoplus \mathcal{C}>$. First element of the

pair represents a D-bag of clause-head tuples, and the second element of the pair represents the clause-body tuples. $\mathcal{L}$ is the D-bag of labels connecting clause-heads to a sequence of the compiled abstract instructions of the corresponding clause-bodies; $\mathcal{N}$ the D-bag of associated priority of the clause-heads to be selected for resolution; $\mathcal{P}$ is the D-bag of procedure-names; $\mathcal{A}_{I(1\leq I\leq N)}$ is the D-bag of the I_{th} argument in the set of clause-heads; $\mathcal{C}$ is a D-bag of the sequence of compiled instructions corresponding to the set of clause-bodies such that each element $c_i \in \mathcal{C}$ is a sequence of instructions corresponding to one clause-body.

The binding environment has two components: a data-parallel binding space and the shared space (popularly called heap). The binding environment in the model is represented as an association of D-bags and F-bags. D-bindings are associated with a D-bag of time stamps for efficient environment recovery (based upon associative search on a time stamp) upon backtracking and with index of the local address space for retrieval of data.

4.1. *Associative resolution*

A goal is represented as a tuple of data elements. Given a conjunctive goal of the form :- $A_1 \bigwedge \ldots \bigwedge A_N$ $(N \geq 0)$, a subgoal A_I is selected randomly, and each individual argument is matched with the corresponding D-bag using rules (19) – (21). At the end of each match, the resulting F-bag $\mathcal{F}_I$ is D-intersected with $\sqcap_{J=0}^{J=I-1}\mathcal{F}_J$ (where $\mathcal{F}_0$ is $\mathcal{F}^1$) — the cumulative result of the previous data parallel matches. The D-substitution after every match is derived by Σ_I^D. If $\sqcap_{J=0}^{J=I}\mathcal{F}_J = \mathcal{F}^0$ then further data-parallel reduction is terminated and backtracking occurs. At the end of successful associative goal reduction, the cumulative D-substitution, denoted by Θ^D, is given by $\Sigma_1^D \bullet^D \ldots \bullet^D \Sigma_N^D$. During the D-composition of D-substitutions, associative computation is exploited in the absence of aliased variables in definitions (i), (ii), (iii), (iv), and (vi). The presence of aliased variables, in conjunction with multiple occurrence variables, causes sequentiality in definitions (v), (vii), and (viii) of D-composition during associative goal reduction. To reduce the runtime overhead caused by sequential execution and data transfer, composition involving aliased variables and multiple-occurrence variables in clause-heads is deferred until a specific clause is selected for the resolution.

The D-bag of unifiable clauses is selected by applying $<\mathcal{L} \bigoplus \mathcal{N} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus \ldots \bigoplus \mathcal{A}_N> \bigotimes \sqcap_{I=0}^{I=N} \mathcal{F}_I$. The associative search strategy for the resolution selects next unifiable clause and execution of the compiled code in the order: simple facts followed by complex facts and complex rules. The rationale is to avoid the nonterminating branches, and to improve the execution efficiency to derive solutions. This strategy is directly supported by search-by-content: the label connecting the clause-head and clause-body is identified after associative goal reduction, and control is transferred to the code marked with the label. Example 1 illustrates the power of associative search strategy.

Example 1:

$p(X, Y)$:- $q(X, Z)$, $r(Z, Y)$.
$p(a, b)$.
$q(X, Y)$:- $q(Y, X)$.
$q(b, c)$.
$r(c, 4)$.
$r(c, 5)$.

In the above example, a simple fact in procedure $p/2$ is selected first to get an early solution. For an alternate solution of $p/2$, a complex clause is selected. However, nonterminating branch in $q/2$ is avoided; execution of $q/2$ derives the value of variable Z as c which when applied in subgoal $r/2$ returns a D-bag $\{4, 5\}$.

The cumulative D-substitution Θ^D is of the form $\{\theta_1^D, ..., \theta_N^D\}$ where θ_I^D is a partial unifier for each potentially unifiable clause-head. The corresponding substitution for aliased variables, denoted by θ_I^S is derived after selecting a potentially unifiable clause. The most general unifier θ_I is derived by $\theta_I^D \bullet \theta_I^S$. θ_I is $\perp$ if either θ_I^D or θ_I^S is $\perp$. It can easily be verified that if an element in $\sqcap_{I=0}^{I=N} F_I$ is $\emptyset$ then the corresponding θ_I^D is $\perp$. If a goal A_J is used for resolution with a clause of the form A :- B_1, ..., B_Q ($Q \geq 0$) then, after the resolution, the resolvent is :- (A_1, ..., A_{J-1}, B_1, ..., B_Q, A_{J+1}, ..., A_N) θ_I. If $\theta_I = \perp$ or there is a binding of the form $X_I/\perp$ then resolution fails; an alternate θ_K ($K \neq I$) is selected according to the associative search strategy. The proof for soundness is very similar proof of soundness for SLD resolution [12], and has been omitted from this paper.

5. A Framework of an Abstract Machine

In this section, the author describes an associative abstract instruction set based on the above model. A detailed scheme for the implementation of this framework on SIMD architecture is given in [2]. Readers can come up with their own implementation model for other architectures based on this framework.

5.1. *A generic abstract instruction set*

The abstract instruction set has five types of instructions: *the set of instructions for data-parallel goal reduction, the set of instructions for data selection, the set of instructions for data movement, the set of instructions for control flow,* and *the set of instructions for data-parallel computation.* The set of data movement instructions is used to transfer data between registers, transfer data between the D-bags in a data parallel binding environment, transfer data between heap and registers, and transfer data between heterogeneous address spaces. The set of control instructions is used to test for $\mathcal{F}^0$, to backtrack, to revert control flow from backtracking to forward flow, and to save the environment on a control stack. Logical parallel instructions are used to select the unifiable clauses by deriving the D-intersection of the corresponding F-bags, to compute the D-union of aliased sets, and to handle the

negation for facts. Arithmetic computation instructions are of three types: *scalar-scalar* → *scalar*, *vector-vector* → *vector*, and *scalar-vector* → *vector*. A description of the set of abstract instructions is given in Figure 1.

data_parallel_arg_match $\mathcal{A}_I$, $\mathcal{U}_I$	matches a goal argument with the corresponding D-bag $\mathcal{A}_I$ (see Rules 19 and 20), derives the F-bag $\mathcal{F}_I$ which is D-intersected with $\mathcal{U}_{I-1}$=$\sqcap_{J=0}^{J=I-1}\mathcal{F}_J$) to derive the new value of $\mathcal{U}_I = \sqcap_{J=0}^{J=I}\mathcal{F}_J$).
data_parallel_unify $\mathcal{A}_I$, $\mathcal{A}_J$, $\mathcal{U}_I$	perform data-parallel equality test between two D-bags $\mathcal{A}_I$ and $\mathcal{A}_J$ to handle multiple-occurrence goal variables (see Rule 21). The resulting F-bag is d-intersected with $\mathcal{U}_{I-1} = \sqcap_{J=0}^{J=I-1}\mathcal{F}_J$) to derive the new value of $\mathcal{U}_I = \sqcap_{J=0}^{J=I}\mathcal{F}_J$).
unify $\mathcal{A}_1$, $\mathcal{A}_2$	performs conventional unification of two logical terms $\mathcal{A}_1$ and $\mathcal{A}_2$, and is used for aliased variables.
d_intersect_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$, $\mathcal{F}_K$	computes and stores $\mathcal{F}_I \sqcap \mathcal{F}_J$ into $\mathcal{F}_K$.
d_union_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$, $\mathcal{F}_K$	computes and stores $\mathcal{F}_I \sqcup \mathcal{F}_J$ into $\mathcal{F}_K$.
d_complement_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$	computes and stores $\neg\ \mathcal{F}_I$ into $\mathcal{F}_J$.
save_bag_id $\mathcal{A}_I$, $\mathcal{D}_I$	stores the reference of a D-substitution (of the variable in the argument $\mathcal{A}_I$) $\mathcal{D}_I$ in the heap. $\mathcal{D}_I$ itself is stored in the data-parallel environment.
save_subbag_id $\mathcal{A}_I$, $\mathcal{D}_I$, $\mathcal{F}_I$	saves the references of the D-substitution of a variable in the argument A_I. The substitution is a D-subbag $\mathcal{D}_I \otimes \mathcal{F}_I$. Both $\mathcal{D}_I$ and $\mathcal{F}_I$ are stored in the data parallel environment.
load_next_binding $\mathcal{A}_I$	loads a value from the D-binding $\mathcal{D}_I \otimes \mathcal{F}_I$ of the variable argument $\mathcal{A}_I$ in a local register for further processing. After selecting an element, the corresponding position in $\mathcal{F}_I$ is replaced by 0.
load_binding $\mathcal{A}_I$	selects a binding for the variable in argument $\mathcal{A}_I$.

Fig. 1. A generic framework of an instruction set

copy_reference $\mathcal{A}_1$, $\mathcal{A}_2$	copies the reference of the binding of the variable in argument $\mathcal{A}_1$ of a goal to the variable in argument $\mathcal{A}_2$ of a subgoal. The advantage of copying the reference is that the large data transfer and structuring overhead of D-subbags across heterogeneous address spaces is reduced.
*backtrack_if_*$\mathcal{F}^0$ $\mathcal{F}_I$	backtracks and restores previous environment if $\mathcal{F}_I$ is $\mathcal{F}^0$.
$Label_K$: *repeat_until_*$\mathcal{F}^0$ $\mathcal{A}_I$, $Label_K$	enforces forward control if the D-binding (for the variable $\mathcal{A}_I$) $\mathcal{D}_I \otimes \mathcal{F}_I \neq \phi^D$, and saves the label $Label_K$; and backtracks if $\mathcal{D}_I \otimes \mathcal{F}_I = \phi^D$. The instruction simulates failure driven iteration, and is used to revert the control flow.
select_next_simple_fact $\mathcal{F}_I$, $Label_I$	selects randomly a unifiable simple fact from $<\mathcal{L} \oplus \mathcal{N} \oplus \mathcal{P} \oplus \mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_N, \mathcal{L} \oplus \mathcal{C}> \otimes \mathcal{F}_I$. After the selection, the corresponding position in $\mathcal{F}_I$ is substituted by *0*. For $\mathcal{F}_I = \mathcal{F}^0$ control is passed to code sequence starting from label $Label_I$.
select_next_clause $\mathcal{F}_I$	selects randomly a unifiable complex clause from $<\mathcal{L} \oplus \mathcal{N} \oplus \mathcal{P} \oplus \mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_N. \mathcal{L} \oplus \mathcal{C}> \otimes \mathcal{F}_I$. After the selection, the corresponding position in $\mathcal{F}_I$ is marked *0*. For $\mathcal{F}_I = \mathcal{F}^0$ backtracking occurs.
call $Label_K$	saves the current environment on the control stack, and passes control to label $Label_K$ corresponding to first instruction of a procedure.
return	restores the previous environment, and *returns* from the called procedure.

Fig. 1. (*Continued*)

5.2. *Applying instruction set to solve a query*

For associative goal reduction, a sequence of instructions *data_parallel_unify* or *data_parallel_arg_match*, *d_intersect_F-bag*, and *backtrack_if_*$\mathcal{F}^0$ are called repeatedly until all the goal-arguments are processed or the F-bag marking the unifiable clauses is equal to $\mathcal{F}^0$. After deriving the D-bag of unifiable clauses, the control is transferred to the compiled code of the corresponding procedure using the label entries in the D-bag of labels corresponding to the unifiable clauses.

For handling simple facts, *select_next_simple_fact* derives a unifiable simple fact to be processed. In the absence of any unifiable simple fact, the control is transferred to process the set of unifiable complex clauses.

For handling the set of unifiable complex clauses, *complement_F-bag* and *d_intersect_F-bag* are used to remove the D-bag of the simple unifiable facts from the D-bag of unifiable clauses; *select_next_clause* is used to select the next unifiable complex clause; *unify* is used to match aliased variables; and *copy_reference* is used to transfer the reference of a D-bindings from a goal to a subgoal.

Producer consumer relationships are divided into two categories:

1. A producer is bound to a D-bag, and is followed by a data-parallel computation (see Rules 14, 15, and 16) on consumer occurrences. The examples for data-parallel computations are an arithmetic operation, a subset selection operation, or a comparison operation.

2. A producer is bound to a D-bag and the consumer is in a goal which consumes one scalar value at a time. Repeated backtracking is used to fetch and process a new value until the D-bag is ϕ^D. This repeated backtracking is achieved by a pair of abstract instructions *repeat_until_*$\mathcal{F}^0$ and *load_next_binding*: the instruction *repeat_until_*$\mathcal{F}^0$ is used to store its label in the previous environment, and passes the control to the instruction *load_next_binding* which selects next value. Upon backtracking, the label is retrieved and the control is passed back to the instruction *repeat_until_*$\mathcal{F}^0$.

In addition, there are multiple data-parallel arithmetic operations, data-parallel arithmetic comparison operations, and data-parallel logical operations.

6. An illustrative example

In this section, the authors explains commonly used abstract instructions through a compiled program. Example 2 illustrates the compilation of a program with simple facts, complex facts, and a complex clause with shared variables. The compiled code has been generated using a compiler, and is executable with minor syntactic changes which were introduced for better readability.

Example 2:

The program has three procedures: $p/2$, $q/2$, and $r/2$. Procedure $p/2$ illustrates compilation of simple facts and complex facts: procedure $q/2$ illustrates compilation of simple facts: procedure $r/2$ illustrates compilation of simple facts mixed with a complex clause. The right hand side of the complex clause in $r/2$ exhibits producer-consumer relationship: the variable Y has producer occurrence in subgoal $p/2$ and consumer occurrence in subgoal $q/2$.

```
p(1, 2). p(2, 3). p(3, 4).
p(X, X). % Complex fact due to aliasing
q(2, 1).    q(3, 2).    q(4, 3).    q(5, 4). % Simple facts
r(2, 2).
r(X, Y) :- p(X, Y), q(2, Y). % Complex clause
```

6.1. *The compiled code*

The I_{th} argument of a goal is denoted by $\mathcal{A}_I^G$; the I_{th} argument in the D-bag of clause-heads is denoted by $\mathcal{A}_I$; $\sqcap_{J=0}^{J=I-1}\mathcal{F}_J$ is denoted by $\mathcal{F}_0^U$; the F-bag of simple facts is denoted by $\mathcal{F}_0^F$; the F-bag of unifiable simple facts is denoted by $\mathcal{F}_0^{UF}$; the F-bag of unifiable complex clauses is denoted by $\mathcal{F}_0^{UC}$; an F-bag to store temporary result is denoted by $\mathcal{F}_1^T$. The set of instructions between the label *p/2* and the label L_{10} correspond to the data parallel goal reduction and the processing of the simple facts in the procedure *p/2*; the set of instructions between the label L_{10} and the label *q/2* correspond to the processing of the complex facts of the procedure *p/2*; the set of instructions between the label *q/2* and the label *r/2* correspond to the data parallel goal reduction and the processing of the simple facts in the procedure *q/2*; the set of instructions between the label *r/2* and the label *CC_r1* correspond to the data parallel goal reduction and the processing of the simple facts in the procedure *r/2*; and the set of instructions following the label *CC_r1* correspond to the processing of the complex clause in the procedure *r/2*.

6.1.1. *Compiled code for data parallel goal reduction*

In this subsection, the sequence of instructions for data parallel goal reduction has been explained for the procedure *p/2*. The instructions for data parallel goal reduction in procedures *q/2* and *r/2* are similar. The instruction *data_parallel_arg_match* — the first instruction after the label *p/2* — matches the first argument of the goal *p/2* and stores the result in an F-bag $\mathcal{F}_0^U$. The instruction *backtrack_if_*$\mathcal{F}^0$ backtracks if the F-bag $\mathcal{F}_0^U$ is equal to $\mathcal{F}^0$. The third instruction matches the second argument of the goal and stores the cumulative result in an F-bag $\mathcal{F}_0^U$. The fourth instruction backtracks if the F-bag $\mathcal{F}_0^U$ is equal to $\mathcal{F}^0$. The instruction *d_intersect_F_bag* identifies the set of unifiable simple facts. The instructions *save_subbag_id* save the D-bindings for the first and second goal arguments respectively.

6.1.2. *Compiled code to process facts*

In this subsection, the sequence of instructions to process simple facts and complex facts in the procedure *p/2* has been described. The instructions for processing simple facts in *q/2* and *r/2* are similar. The instruction *complement_F_bag* marks the set of complex clauses using an F-bag $\mathcal{F}_1^T$. The following instruction marks the set of unifiable complex clauses in an F-bag $\mathcal{F}_0^{UC}$. The instruction *select_next_simple_fact* selects an unifiable simple fact if the F-bag $\mathcal{F}_0^{UFc}$ is not null. Otherwise, the control is transferred to L_{10} to process a complex clause. The instruction *select_next_clause* selects next complex clause for processing, and the corresponding bit in the F-bag $\mathcal{F}_0^{UC}$ is reset. The instruction *unify* unifies aliased variable X in

the complex fact of *p/2*.

6.1.3. *Compiled code to process a complex clause*

In this subsection, the compiled code (following label *CC_r1*) for the complex clause of the procedure *r/2* has been explained. The instructions *copy_reference* copies the reference of the bindings of the variables X and Y in the first and second arguments of the subgoal *p/2* respectively. The instruction *call* invokes the procedure *p/2*. The instruction *load_data* following the label L_{39} loads constant *2* in the first argument of the subgoal *q/2*, and the following instruction copies the reference of the binding of the variable Y in the second argument of the subgoal *q/2*. The instruction *repeat_until_*$\mathcal{F}^0$ sets up a failure driven loop to process the following sequence of instructions for every binding of the shared variable Y. The instruction *load_next_bindings* takes the next scalar binding from the D-binding of the variable Y, and the following instruction invokes the procedure *q/2* for every scalar binding of the variable Y. After all the bindings of the variable Y are consumed, the instruction *repeat_until_*$\mathcal{F}^0$ backtracks.

% data-parallel goal reduction for procedure *p/2*.

p/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	d_intersect_F-bag	$\mathcal{F}_0$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_1^G$,	$\mathcal{A}_1$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_2^G$,	$\mathcal{A}_2$,	$\mathcal{F}_0^{UF}$

% Handling simple facts for procedure *p/2*

complement_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_1^T$	
d_intersect_F-bag	$\mathcal{F}_1^T$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$
select_next_simple_fact	$\mathcal{F}_0^{UF}$,	L_{10}	

% Complex fact processing for procedure *p/2*

L_{10}:	*select_next_clause*	$\mathcal{F}_0^{UC}$	
CF_p/2:	*unify*	$\mathcal{A}_1$,	$\mathcal{A}_2$
	return		

% Instructions for data parallel goal reduction in the procedure *q/2*.

q/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	d_intersect_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_1^G$,	$\mathcal{A}_1$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_2^G$,	$\mathcal{A}_2$,	$\mathcal{F}_0^{UF}$

Fig. 2. A compiled code

% Instructions to process simple facts in the procedure *q/2*.

	complement_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_1^T$	
	d_intersect_F-bag	$\mathcal{F}_1^T$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$
	select_next_simple_fact	$\mathcal{F}_0^{UF}$,	L_{23}	
L_{23}:	*select_next_clause*	$\mathcal{F}_0^{UC}$		

% Instructions for data parallel goal reduction in the procedure *r/2*.

r/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	d_intersect_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_1^G$,	$\mathcal{A}_1$,	$\mathcal{F}_0^{UF}$
	save_subbag_id	$\mathcal{A}_2^G$,	$\mathcal{A}_2$,	$\mathcal{F}_0^{UF}$

% Instructions to process simple facts in the procedure *r/2*

	complement_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_1^T$	
	d_intersect_F-bag	$\mathcal{F}_1^T$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$
	select_next_simple_fact	$\mathcal{F}_0^{UF}$,	L_{34}	
L_{34}:	*select_next_clause*	$\mathcal{F}_0^{UC}$		

% Instructions to handle complex rule

CC_r1:	*copy_reference*	$\mathcal{A}_1$,	$\mathcal{A}_1$
	copy_reference	$\mathcal{A}_2$,	$\mathcal{A}_2$
	call	*DP_p/2*	
L_{39}:	*load_data*	$\mathcal{A}_1$,	*2*
	copy_reference	$\mathcal{A}_2$,	$\mathcal{A}_2$
L_{42}:	*repeat_until_*$\mathcal{F}^0$	$\mathcal{A}_2$	L_{42}
	load_next_binding	$\mathcal{A}_2$	
	call	*DP_q/2*	
	return		

Fig. 2. (*Continued*)

6.2. *Performance evaluation*

An emulator has been implemented for the generic abstract instruction set in a homogeneous address space. The emulator is portable to any architecture which supports data-parallel version of C. The results demonstrate that the number of operations needed for associative lookup is independent of the number of simple facts. Thirty operations are needed to match a simple fact with two arguments. The number of operations is linearly dependent upon the number of arguments in a query. For each extra argument, nine extra operations are needed.

For a 10 ns clock supported by current technology, and three clock cycles (load-execute-store cycle), the associative look up speed is 1.2 million × number of facts

for a set of facts with two arguments. In the presence of data-parallel scientific computations intertwined with associative lookup, the peek execution speed is limited by the associative look-up speed which will be one hundred and twenty MCPS (million computations per second) for one thousand facts.

When two subgoals of a rule share variables, and the second subgoal is not a data-parallel computation, the data elements in the vector bindings for the shared variables are processed one at a time. This scenario is the worst case for execution, and the execution speed reduces to four hundred thousand logical inferences per second (LIPS) for one shared variable. This slow down is caused primarily due to the overhead of storing the control thread during the forward control flow, the register set up, and the retrieval of the control thread during backtracking.

7. Conclusion

In this paper, the author describes a framework of heterogeneous associative logic programming for heterogeneous address spaces and a generic abstract instruction set which can be easily altered to exploit the advantages of a particular architecture. The model is scalable across different architectures, and is also compatible with low level synchronization if needed. An advantage of associative goal reduction is that the data can be distributed on heterogeneous collection of architectures, and each machine may execute the same abstract computation on different data sets. The use of bit-vectors reduces the data transfer overhead when the data is distributed across different machines.

References

[1] A. K. Bansal, and J. L. Potter, *An Associative Model to Minimize Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases*, The International Journal of Engineering Applications of Artificial Intelligence, Permagon Press, **Volume 5, Number 3**, (1992) 247–262.

[2] A. K. Bansal, *An Associative Model to Integrate Knowledge Retrieval and data-parallel Computation*, International Journal on Artificial Intelligence Tools, **Volume 3, Number 1**, World Scientific, (1994) 97–125.

[3] A. K. Bansal, *Towards a Formal Computational Model for Associative Logic Programming*, Proceedings of the International Workshop on data-parallel Implementation of Declarative Languages, Geneoa, Italy, (1994) 11–20.

[4] A. K. Bansal, L. Prasad, and M. Ghandikota, *A Formal Associative Model of Logic Programming and its Abstract Instruction Set*, Proceedings of the International Conference of Tools with Artificial Intelligence, (1994) 145–151.

[5] C. Dwork, P. Kanellakis, and J. Mitchell, *On the Sequential Nature of Unification*, The Journal of Logic Programming, Springer-verlag, (1984) 35-50.

[6] J. A. Feldman, and D. Rovner, *An Algol Based Associative Language*, *Communications of the ACM*, **Volume 12, Number 8**, (1969) 439–449.

[7] D. Gries, *The Science of Programming*, Monograph, Springer Verlag, New York, 1987.

[8] K. Hwang, and F. A. Briggs, *Computer Architecture and Parallel Processing*, Mcgraw Hill Book Company, New york, USA, (1984).

[9] Kacsuk, P., and Bale, A., *DAP Prolog: A Set Oriented Approach to Prolog*, The Computer Journal, **Volume 30, Number 5**, (1987) 393-403.

[10] K. Knobe, J. D. Lukas, G. Steele, *Massively Data Parallel Optimization*, The 2nd Symposium of Massively Parallel Computation, Fairfax, Virginia, (1988) 551–558.

[11] R. Kowalski, *Logic for Problem Solving*, Elsevier-North Holland, (1979).

[12] J. Lloyd, *Foundations of Logic Programming*, Springer Verlag, New York, (1984).

[13] Z. Manna, and R. Waldinger, *The Logical Basis for Computer Programming*, **Volume 1**: Deductive Reasoning, Addison Wesley, (1985).

[14] Message Passing Interface Forum, *MPI: A message-passing interface standard*, Computer Science Dept. Technical Report CS-94-230, University of Tennessee, (1994).

[15] J. L. Potter, *Associative Computing*, Plenum Publishers. New York, (1992).

[16] V. S. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice and Experience, **2**, (1990) 315-339.

[17] A. Takeuchi, and K. Furukawa. *Parallel Logic Programming Languages*, Lecture Notes In Computer Science, **Vol. 225**, Springer Verlag, New York, (1986) 242–254.

[18] D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical Report 309. SRI International. (1983).