MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1343 December 1991

Lifting Transformations

David Allen McAllester and Jeffery Mark Siskind

Abstract

Lifting is a well known technique in resolution theorem proving, logic programming, and term rewriting. In this paper we formulate lifting as an efficiency-motivated program transformation applicable to a wide variety of nondeterministic procedures. This formulation allows the immediate lifting of complex procedures, such as the Davis-Putnam algorithm, which are otherwise difficult to lift. We treat both classical lifting, which is based on unification, and various closely related program transformations which we also call lifting transformations. These nonclassical lifting transformations are closely related to constraint techniques in logic programming, resolution, and term rewriting. Formulating these techniques as transformations on nondeterministic programs expands the range of procedures to which the techniques can be easily applied.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the work described in this paper was provided in part by Misubishi Electric Research Laboratories, Inc. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

1 Introduction

Lifting is a well known technique in resolution theorem proving, logic programming, and term rewriting [Robinson, 1965], [Knuth and Bendix, 1969]. In this paper we formulate lifting as an efficiency-motivated program transformation that can be applied to a wide variety of nondeterministic programs. This lifting transformation immediately gives lifted versions of a variety complex ground procedures. Section 6 shows how the lifting transformation provides an immediate lifting of the Davis-Putnam procedure for automated theorem proving [Davis and Putnam, 1960]. The early success of resolution over the Davis-Putnam procedure is generally attributed to the fact that, unlike the Davis-Putnam procedure, resolution is easily lifted. The general lifting transformation can be applied to other complex ground procedures such as so-called "nonlinear planners" for solving AI planning problems [McAllester and Rosenblitt, 1991].

To formulate lifting as a general transformation on nondeterministic programs we must first formulate nondeterministic programs. The basic observation that nondeterministic programs provide a good representation of search problems has been made by a variety of authors [McCarthy, 1967], [Floyd, 1967]. The use of nondeterminism to represent search is a motivation for automatic backtracking in Prolog. It seems that nondeterminism (and automatic backtracking) are largely orthogonal to other features of a programming language — one can add nondeterminism to Lisp [McCarthy, 1967], or flowcharts [Floyd, 1967], and one could even remove nondeterminism from logic programming. Nondeterminism and automatic backtracking can be easily implemented in Scheme or SML using first class continuations [Haynes, 1987]. All of the techniques discussed in this paper have been implemented in a nondeterministic version of Common Lisp [Siskind and McAllester, 1992b]. Any nondeterministic programming language provides a formal model of nondeterministic computation that can be substituted for the notion of a nondeterministic Turing machine in the definition of nondeterministic complexity classes such as NP. In this paper we give an operational semantics for a simple nondeterministic functional language similar to pure Lisp but extended with primitives for nondeterminism.

Section 4 formally defines lifting as an efficiency-motivated transformation on nondeterministic programs. Section 4 gives three lifting lemmas for this general transformation. The first lemma states that lifting is sound — any value derived from lifted computation is "legitimate". The second lemma states that lifting is complete, i.e., any possible ground computation is an "instance" of some lifted computation. The third lemma states that lifting preserves systematicity. Systematicity is a desirable property of search programs — a search is systematic if it never generates the same solution twice. Although lifting lemmas similar to

the first two lemmas of section 4 are common in the resolution and term rewriting literature, the formulation of lifting lemmas for a general purpose program transformation appears to be new.

Classical lifting is viewed here as one example of a large family of efficiency-motivated lifting transformations. Classical lifting generates equational constraints whose satisfiability can be tested with classical unification. Nonclassical (or constraint) lifting generates other forms of constraints, such as constraints on variables that range over integers or floating point numbers. The lifting lemmas of section 4 apply to a wide variety of mechanical lifting transformations.

Our nonclassical lifting transformations are closely related to the use of constraints in logic programming, resolution theorem proving, and term rewriting, e.g., [Jaffar and Lassez, 1987], [Colmerauer, 1986], [van Hentenryck, 1989], [Bürckert, 1990], [Martin and Nipkow, 1990], [Peterson, 1990]. All of these systems can be loosely characterized as variants on "constraint programming". Each system maintains sets of constraints and uses some form of inference procedure to either test the satisfiability of constraints or to infer "new" constraints from those given. Constraint programming is formulated here as a general efficiency-motivated lifting transformation on nondeterministic programs. As with classical lifting, the formulation of constraint programming as a general program transformation immediately gives constraint versions for a wide variety of nondeterministic procedures and algorithms.

The formulation of lifting and constraint programming as efficiency-motivated program transformations emphasizes a certain perspective on the relationship between logic and computer science. Logic is often viewed as a source of techniques for ensuring the "correctness" or "safety" of computer programs. But the program transformation view of lifting and constraint programming shows how logical inference, in the form of constraint processing, plays an important role in improving the *efficiency* of a wide variety of computations. There are many computational problems whose solution requires search. Inference improves the efficiency of search by pruning large portions of the search space. The formulation of lifting and constraint programming as program transformations allows inference-based pruning to be applied to a wide variety of nondeterministic programs.

2 A Simple Functional Language

This section precisely defines the syntax and semantics of a simple functional programming language. Neither the language nor its semantics are particularly interesting or novel. The operational semantics is given as a rewrite relation analogous to operational semantics that have been given for a variety of languages. A

good discussion of operational semantics and the relation between operational and denotational semantics can be found in [Plotkin, 1977] or [Milner, 1977]. Although the operational semantics presented here is not novel, it is necessary for a rigorous discussion of lifting transformations.

The language defined here serves as a formal model of deterministic computation. The language defined in the next section serves as a formal model of nondeterministic computation. Our models of deterministic and nondeterministic computation could be used in the place of Turing machines to define standard complexity classes such as P and NP. Compared to modern typed λ-calculi our languages may seem unsophisticated. However, they serve as an adequate foundation for formulating efficiency-motivated constraint transformations on nondeterministic computation.

The language defined here is an untyped call by value λ-calculus. The language includes a handful of primitives for manipulating list structure. These primitives play an important role in the classical lifting transformation. We also provide an explicit Y operator to simplify the discussion of recursion.

Definition: Quoted symbols, such as `'foo`, and the (unquoted) symbols `cons`, `car`, `cdr`, `equal`, and `Y` are called *primitive constants.*

Definition: The symbol `if` is called a *special form.*

Definition: An *expression* is one of the following.

- A variable.
- A primitive constant.
- A special form.
- A λ-expression of the form (`LAMBDA` $(x_1 \ \ldots \ x_n)$ $s[x_1 \ldots x_n]$) where each x_i is a variable and $s[x_1 \ldots x_n]$ is an expression possibly involving the variables $x_1, \ldots, x_n$.
- A combination $(e_1 \ e_2 \ \ldots \ e_n)$ where each e_i is an expression.

An expression which does not contain free variables will be called *closed.* We allow λ-expressions of no arguments, i.e., expressions of the form (`LAMBDA ()` b), as a special case of λ-expressions. λ-expressions of no arguments are sometimes called *thunks.*

We now define the notion of a "value".

Definition: A *first order value* is either a quoted symbol or an application of the form (`cons` s t) where s and t are values.

Definition: A *value* is either a first order value or a closed λ-expression.

Note that the application `(car (cons 'a 'b))` is not a value, although (as we shall see) it computes to the value `'a`. Similarly, `(cons (car (cons 'a 'b)) 'c)` computes to the value `(cons 'a 'c)`. We define a separate class of first order values because the lifting transformations discussed in this paper require that the logic variables introduced by lifting be first order — they range only over first order values.

We formalize computation as a process of rewriting expressions. We write $s \rightarrow t$ to indicate that the program s is converted to the program t by a single rewrite step. Intuitively, each rewrite step corresponds to a single step in a computational process. The rewrite relation $\rightarrow$ is defined by the set of rewrite rules given below.

- If s is a first order value then `(EQUAL` s s`)` $\rightarrow$ `'T`.
- If s and t are different first order values then `(EQUAL` s t`)` $\rightarrow$ `'F`.
- If s and t are first order values then `(CAR (CONS` s t`))` $\rightarrow s$.
- If s and t are first order values then `(CDR (CONS` s t`))` $\rightarrow t$.
- If $s \rightarrow s'$ then `(if` s t u`)` $\rightarrow$ `(if` s' t u`)`.
- `(if 'T` t u`)` $\rightarrow t$.
- `(if 'F` t u`)` $\rightarrow u$.
- If all of $v_1, \ldots, v_n$ are values and f is the λ-expression `(LAMBDA (`x_1 ... x_n`)` $B[x_1 \ldots x_n]$), then (f v_1, ... v_n) $\rightarrow B[v_1 \ldots v_n]$.
- `(Y (LAMBDA (`f`)` $b[f]$`))` $\rightarrow b$[`(Y (LAMBDA (`f`)` $b[f]$`))`]
- If $f \rightarrow g$ then (f e_1 ... e_n) $\rightarrow$ (g e_1 ... e_n).
- If f is a value other than a special form and e_i is the first expression among e_1, ..., e_n that is not a value then if $e_i \rightarrow e_i'$ we have (f e_1, ..., e_i, ..., e_n) $\rightarrow$ (f e_1, ..., e_i', ..., e_n).

Notice that it is possible for expressions to rewrite to normal forms which are not values. For example, the expression `(car 'a)` can not be further rewritten but it not a value. Normal forms which are not values are considered to be errors or failures.

In the examples throughout the remainder of this paper we use definitions of the following form.

```
(define (foo x1 ... xn)
    body)
```

If the definition is not recursive, e.g., if `foo` does not appear in the body of the definition, then the definition signifies that `foo` is to be taken as an abbreviation for (`LAMBDA` (x_1 ... x_n) *body*). If the definition is recursive then `foo` is to be taken as an abbreviation for

(`Y` (`LAMBDA` (`foo`) (`LAMBDA` (x_1 ... x_n) *body*))).

We give the following definition of `append` as an example.

```
(define (append x y)
  (if (equal x 'nil)
      y
      (cons (car x) (append (cdr x) y))))
```

On can readily verify that if x is a well formed list then (`append` x y) will rewrite to the standard value of the append function. If x is not a well formed list then (`append` x y) will rewrite to a normal form that is not a value — we would say that the computation terminates in an error.

3 Nondeterminism

In this section we extend the language defined in the previous section with the new special form `either` and the new primitive constant `a-value`.[1] The operational semantics of these new primitives can be defined with the following rewrite rules.

- (`either` s t) $\rightarrow$ (if 'T s t).
- (`either` s t) $\rightarrow$ (if 'F s t).
- (`a-value`) $\rightarrow$ s where s is any first order value.

The particular form of the rewrite rules for `either` is required for the "systematicity" lifting lemma given in section 4. Intuitively, the form of these rules ensure

[1] By making `either` a special form we avoid the evaluation of both of its arguments as would otherwise be required by rewrite rules of the previous section.

that one can distinguish the two possible computation paths for an expression of the form (either s s). If we allowed the computation step (either s s) $\rightarrow$ s then we could not distinguish selecting the first argument form selecting the second argument. Systematicity can be defined as follows.

We give the following simple example of a nondeterministic procedure. The procedure takes a list of proposition symbols and returns a truth assignment on those symbols. The truth assignment is represented by a list of pairs.

```
(define (a-truth-assignment-on propositions)
  (if (equal propositions 'nil)
      'nil
      (cons (cons (car propositions) (either 'T 'F))
            (a-truth-assignment-on (cdr propositions)))))
```

Although the special form either is sufficient for many purposes, the primitive a-value plays a crucial role in the classical lifting transformation. The following is a simple example of the use of the primitive a-value.

```
(define (a-substitution-on variables)
  (if (equal variables 'nil)
      'nil
      (cons (cons (car variables) (a-value))
            (a-substitution-on (cdr variables)))))
```

It is convenient to introduce the primitive fail. The expression (fail) is not considered to be an error, nor is it a value. If one of the computations of s ends in the expression (fail) then that computation is effectively "pruned" — it is not an error but it does not contribute any value. The technical distinction between errors and failures is not needed for the analyses given in this paper and no formal definition of this distinction is given here. The following is a nondeterministic procedure for finding a truth assignment satisfying a given formula.

```
(define (satisfying-assignment formula)
  (let ((assignment (a-truth-assignment-on (variables-of formula))))
    (if (satisfies?  assignment formula)
        assignment
        (fail))))
```

We leave it to the reader to define the procedure satisfies?. We now introduce some additional terminology.

Nondeterministic programs define search trees. More specifically, the set of possible executions of an expression s form a tree whose root is s itself and where branching occurs whenever there is more than one way of continuing the computation from an intermediate expression. Each leaf in this tree is a normal form of s. It is possible to build efficient interpreters (or compilers) that systematically search the tree of possible executions with a minimum of overhead. This search process for a nondeterministic language can be easily implemented in languages such as Scheme or SML with first class continuations. More efficient implementations can (arguably) be achieved by performing a continuation passing transformation and then stack allocating failure continuations. This latter approach is taken in the Common Lisp implementation of the ideas presented in this paper [Siskind and McAllester, 1992b].

> **Definition:** An *execution* of an expression s is a finite reduction sequence of the form $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$ where s_0 is s.

> **Definition:** A *possible value* of an expression s is any value that is the final expression in an execution of s.

Intuitively, a search process is systematic if it never examines the same solution twice.

> **Definition:** An expression s is called *systematic* if no possible value of s is the final value of more than one execution of s.

Most simple generate and test programs are systematic provided they return the generated data structure as a value. Systematicity can be viewed as an "anti-confluence" condition — in systematic search divergent nondeterministic computation paths should not reconverge.

4 Lifted Computation

Lifted computation operates on "logic variables" and constraints on logic variables. We start by defining a general notion of a constraint.

> **Definition:** A *value substitution* is a mapping σ from variables to first order values.

> **Definition:** A *type constraint* is a pair of a variable x and an expression s. A type constraint is written as $x : s$.

Definition: A value substitution σ *satisfies* a type constraint $x : s$ if the value $\sigma[x]$ is a possible value of the expression $\sigma[s]$. A value substitution σ satisfies a set of constraints Σ if σ satisfies each member of Σ.

For example, the constraint x : `(either 'T 'F)` states that x is a Boolean variable — for any substitution σ satisfying this constraint $\sigma[x]$ must be either `'T` or `'F`. As another example consider a constraint of the form x : `(cons` y z`)` where x, y, and z are variables. This constraint can be viewed as an equation of the form $x =$ `(cons` y z`)`.

We formalize lifted computation as a rewrite relation. In the lifted rewrite relation the states of the computation are formalized as triples $<s, \Sigma, \mathcal{V}>$ where s is an expression, Σ is a set of constraints (discussed below), and $\mathcal{V}$ is an infinite list of "unused" variables. The relation symbol $\hookrightarrow$ is defined by a set of rewrite rules. The first rule states that $\hookrightarrow$ is an extension of the nondeterministic relation $\rightarrow$.

- If $s \rightarrow t$ then $<s, \Sigma, \mathcal{V}> \hookrightarrow <t, \Sigma, \mathcal{V}>$.

Lifted computation is based on a new special form, `make-variable` which takes one argument, a "type", and returns a new variable of that type. The following rewrite rule gives the operational semantics of `make-variable`. The expression `(make-variable` t`)` rewrites to a previously unused variable and imposes a constraint on the type of that variable.

- $<$`(make-variable` t`)`$, \Sigma, \mathcal{V}> \hookrightarrow <x, \Sigma \cup \{x : t\}, \mathcal{V} - \{x\}>$ where x is the first variable in the sequence $\mathcal{V}$.

The classical lifting transformation involves expressions of the form `(make-variable (a-value))`. Such expressions generate type constraints of the form x : `(a-value)`.

We now give the remaining rewrite rules for lifted computation. The following rewrite rules specify the "locus of control". These rules are compatible with the corresponding rules for the ground relation $\rightarrow$ but they handle the case where the internal computation results in new variables and constraints.

- If $<s, \Sigma, \mathcal{V}> \hookrightarrow <s', \Sigma', \mathcal{V}'>$ then $<$`(if` s t u`)`$, \Sigma, \mathcal{V}> \hookrightarrow <$`(if` s' t u`)`$, \Sigma', \mathcal{V}'>$.
- If $<f, \Sigma, \mathcal{V}> \hookrightarrow <g, \Sigma', \mathcal{V}'>$ then $<(f\ e_1\ \ldots\ e_n), \Sigma, \mathcal{V}> \hookrightarrow <(g\ e_1\ \ldots\ e_n), \Sigma', \mathcal{V}'>$.
- If f is a value other than a special form and e_i is the first expression among $e_1, \ldots, e_n$ that is not a value then if $<e_i, \Sigma, \mathcal{V}> \rightarrow <e_i', \Sigma', \mathcal{V}'>$ then

$$<(f\ e_1\ \ldots\ e_i\ \ldots\ e_n), \Sigma, \mathcal{V}> \hookrightarrow <(f\ e_1\ \ldots\ e_i'\ \ldots\ e_n), \Sigma', \mathcal{V}'>.$$

Finally we give three rules that are the essence of our general concept of lifted computation.

- If c is a primitive constant, $s_1 \ldots s_n$ are either values or variables, and at least one of $s_1, \ldots, s_n$ is a variable, then $<(c\ s_1 \ldots s_k), \Sigma, \mathcal{V}> \hookrightarrow <x, \Sigma \cup \{x : (c\ s_1 \ldots s_k)\}, \mathcal{V} - \{x\}>$ where x is the first variable in $\mathcal{V}$.
- If all of $v_1, \ldots, v_n$ are either first order values, variables, or λ-expressions, and f is the λ-expression (`LAMBDA` $(x_1 \ldots x_n)\ B[x_1 \ldots x_n]$), then

$$<(f\ v_1, \ldots\ v_n), \Sigma, \mathcal{V}> \hookrightarrow <B[v_1 \ldots v_n], \Sigma, \mathcal{V}>.$$

- If x is a variable then $<$(`if` $x\ t\ u$), $\Sigma, \mathcal{V}> \hookrightarrow <t, \Sigma \cup \{x : \text{'T}\}, \mathcal{V}>$.
- If x is a variable then $<$(`if` $x\ t\ u$), $\Sigma, \mathcal{V}> \hookrightarrow <u, \Sigma \cup \{x : \text{'F}\}, \mathcal{V}>$.

Lifted computation will terminate if it encounters an application of the form $(x\ e_1 \ldots e_n)$ where x is a variable. If all variables are first order then any corresponding ground computation must also terminate at this point. Higher order logic variables could be handled by adding a lifted rewrite rule which allowed expressions of the form $(x\ e_1 \ldots e_n)$ to be rewritten to a variable y subject to the constraint $y : (x\ e_1 \ldots e_n)$. In this paper, however, we restrict our attention to first order logic variables. We now have three lifting lemmas.

> **Definition:** A *properly lifted expression* is a closed expression s such that, for any expression of the form (`make-variable` t) that appears in s, every possible value of t is first order.

> **Definition:** An expression s will be called *ground* if it is closed and it does not contain the constant `make-variable`.

> **Definition:** We define the *ground version* of an expression s, denoted $\mathcal{G}[s]$, to be the result of replacing each expression of the form (`make-variable` t) in s by the expression t.

The reader can verify that if s is properly lifted then $\mathcal{G}[s]$ is ground.

> **Definition:** A *lifted execution* of an expression s is a finite reduction sequence of the form $<s_0, \Sigma_0, \mathcal{V}_0> \hookrightarrow <s_1, \Sigma_1, \mathcal{V}_1> \hookrightarrow \ldots \hookrightarrow <s_k, \Sigma_k, \mathcal{V}_k>$ where s_0 is s, Σ_0 is the empty set of constraints, and $\mathcal{V}_0$ is the (fixed) initial infinite sequence of variables.

Definition: A *lifted value* of an expression s is a pair $<v, \Sigma>$ where v is either a value, a variable, or a λ-expression, and where $<v, \Sigma>$ appears as the first two components in the final triple of a lifted computation of s.

Definition: A pair $<v, \Sigma>$ is said to *cover* a value v' if there exists a value substitution σ satisfying Σ such that $\sigma[v]$ is v'.

Soundness Lifting Lemma: If s is a properly lifted expression then any value covered by a lifted value of s is a possible value of $\mathcal{G}[s]$.

Completeness Lifting Lemma: If s is a properly lifted expression then any a possible value of $\mathcal{G}[s]$ is covered by some lifted value of s.

Systematicity Lifting Lemma: For any properly lifted expression s, if $\mathcal{G}[s]$ is systematic then no value of $\mathcal{G}[s]$ is covered by two different lifted computations of s, i.e., no two lifted computations of s terminate in lifted values that both cover the same value of $\mathcal{G}[s]$.

The systematicity lifting lemma requires the particular form of the rewrite rules for `either` given in section 3.

5 Classical Lifting and Unification

In this section we introduce a "classical" lifting transformation which maps a ground expression s to a properly lifted expression $\mathcal{L}[s]$. The constraints generated by a lifted execution of $\mathcal{L}[s]$ can be viewed as a set of equations and disequations between expressions.[2] Unification can be used to find a most general unifier of the equation constraints and one can check that this unifier does not violate any disequation constraint. In short, unification provides a fast consistency test for the constraint sets generated by classical lifting.

Definition: For any expression s we define $\mathcal{L}(s)$, called the classical lifting of s, to be the expression that results from replacing each ocurrence `(a-value)` in s by `(make-variable (a-value))` and each expression of the form `(equal` s t`)` by `(if (equal` s t`) 'T 'F)`.

[2] A disequation is the negation of an equation. Disequations have not played an important role in logic programming but they are required under a program transformation view of lifting.

The classical lifting transformation has the property that for any ground expression s, the expression $\mathcal{L}[s]$ is properly lifted and the ground expression $\mathcal{G}[\mathcal{L}[s]]$ has the same set of possible values as s. By replacing each expression of the form `(equal` s t`)` by `(if (equal` s t`) 'T 'F)` we ensure that whenever lifted computation generates a constraint of the form x : `(equal` s t`)` it also generates either x : `'T` or x : `'F`. This ensures that the set of constraints generated by lifted computation can be converted to a set of equations and disequations. For example, a constraint of the form x : `(car` y`)` can be converted to an equation of the form $y =$ `(cons` x z`)` where z is a new variable.

6 Lifting the Davis-Putnam Procedure

The transformation view of lifting allows the immediate lifting of a variety of nondeterministic programs that are otherwise difficult to lift. The example given here is the Davis-Putnam procedure for automated theorem proving. Many authors have attributed the early success of resolution techniques over the Davis-Putnam procedure to the fact that ground resolution can be easily lifted while Davis-Putnam can not. However, given a formulation of lifting as a general program transformation one has an immediate lifting Davis-Putnam. We can define a version of the Davis-Putnam procedure as follows.

> **Davis-Putnam Procedure:** To determine the consistency of a given set Δ of clauses do the following.
>
> 1. Let Γ be a finite set of ground instances of clauses in Δ.
> 2. If Γ is unsatisfiable then output `'unsatisfiable` else fail.

The given set of clauses Δ is unsatisfiable if and only if there exists some execution of the above procedure which outputs `'unsatisfiable`. The above procedure can be implemented as a nondeterministic ground λ-expression of one argument. Step 1 is (of course) nondeterministic with an infinite number of possible outcomes. Step 1 can be implemented using the following procedure to construct a ground instance of a clause.

```
(define (a-ground-instance-of exp)
  (let ((subst (a-substitution-on (variables-in exp))))
    (apply-substitution subst exp)))
```

We leave it to the reader to construct a representation of variables and to construct the procedures `variables-in` and `apply-substitution`. The procedure `a-substitution-on` is defined in section 3 and uses the primitive `a-value`.

Step 2 must be implemented deterministically — step 2 must show that *all* truth assignments fail to satisfy Γ. In step 2 we can use the most efficient known algorithm for determining if a set of ground clauses is satisfiable.

Once the Davis-Putnam procedure has been implemented as a nondeterministic ground procedure the classical lifting transformation can be applied directly. In the lifted version, step 2 constructs a set of copies of clauses in Δ where each copy is built from fresh logic variables. The correctness of this transformation on the Davis-Putnam procedure is immediately implied by the general lifting lemmas of the previous section. The systematicity lemma does not apply since the ground Davis-Putnam procedure is not systematic. Another example of a lifted procedure where lifting would be difficult without the general transformation can be found in [McAllester and Rosenblitt, 1991]. This example involves a procedure for solving AI planning problems.

7 Other Lifting Transformations

Other useful efficiency-motivated lifting transformations can be constructed by introducing other nondeterministic primitives into the language. Our first example is a Boolean lifting transformation. We can introduce the primitive `(a-boolean-value)` and the primitives `and`, `or`, `implies`, and `not`. These primitives can be given operational semantics with ground rewrite rules such as the following.

- `(a-boolean-value)` $\rightarrow$ `'T`
- `(a-boolean-value)` $\rightarrow$ `'F`
- `(and 'T 'T)` $\rightarrow$ `'T`
- `(and 'T 'F)` $\rightarrow$ `'F`
- $\vdots$

We can now define a Boolean lifting transformation.

> **Definition:** For any expression s we define the *Boolean lifting* of s, denoted $\mathcal{B}[s]$, to be the result of replacing each ocurrence of `(a-boolean-value)` by `(make-variable (a-boolean-value))`.

Lifted computation is defined with the rewrite rules given in the section 4. The three lifting lemmas of section 4 hold for the Boolean lifting transformation as well as classical lifting.

A lifted computation of a Boolean-lifted expression can generate constraints of the form x : `(and` y z`)`. Any value assignment satisfying this constraint must assign either `'T` or `'F` to each of the variable x, y, and z, and the truth value assigned to x must be the logical and of the values assigned to y and z. Given a careful implementation of constraint satisfaction techniques for equational (unification) constraints, such as x : `(cons` y z`)`, and Boolean constraints, such as x : `(and` y z`)`, the Boolean lifting transformation mechanically converts the simple generate and test procedure for Boolean satisfiability procedure given in section 3 into one of the most efficient known procedures for Boolean satisfiability.

As another example of an efficiency motivated lifting transformation we consider the primitives `a-floating-point-number`, `+`, `*`, `<`, and `=`. These primitives can be given operational semantics using ground rewrite rules such as the following.

- `(a-floating-point-number)` $\rightarrow f$ where f is any IEEE standard floating point number.
- `(+` f_1 f_2`)` $\rightarrow f_3$ where f_3 is the IEEE standard sum of the floating point numbers f_1 and f_2.

  $\vdots$
- `(<` f_1 f_2`)` $\rightarrow$ `'T` if f_1 is less than f_2.

  $\vdots$

We have choosen floating point numbers rather than reals or rationals to emphasize that efficiency-motivated lifting transformation can be applied to a variety of "practical" data types and operations.

> **Definition:** For any expression s we define the *floating point lifting* of s, denoted $\mathcal{F}[s]$, to be the result of replacing each ocurrence of `(a-floating-point-number)` by `(make-variable (a-floating-point-number))`.

As an example consider the following simple nondeterministic procedure.

```
(let ((x (a-floating-point-number))
      (y (a-floating-point-number)))
  (if (and (< (+ (* x x) (* y y)) 1.0)
           (< .9 (* x y)))
      (cons x y)
      (fail)))
```

Although this expression has a well defined set of possible values, one would not normally attempt to find a value by simple backtrack search on the choice of floating point numbers. However, the floating point lifting of this expression has only one lifted value which consists of a variable representing the cons expression (`cons` x y) and a collection of constraints on the floating point numbers x and y.

In the case of classical lifting the lifted computations can be pruned by eliminating any lifted computation that generates an unsatisfiable set of constraints. Satisfiability can be tested with the unification procedure. In the case of Boolean and floating-point lifting, however, it seems that a more efficient strategy is to use fast but incomplete pruning of the lifted computations. Under incomplete pruning of the lifted computations there is no guarantee that the constraint set resulting from an unpruned lifted computation is satisfiable. After a lifted value is found one may still have to search for a solution of the resulting constraint set. In the case of floating point lifting one may have to search for solutions to a system of nonlinear equations involving numerical variables. Our Common Lisp implementation of floating point lifting uses interval techniques to find solutions to systems of nonlinear equations [Hansen, 1968], [Hansen and Walster, 1991]. In the interval method one associates each numerical variable with a numerical upper and lower bound. The two bounds define an interval of possible values for each variable. New bounds can be inferred from existing bounds and constraints. For example, given the constraint x : (`+` y `3.0`), and an upper bound of 2.0 for y, we can infer an upper bound of 5.0 for x. We call this inference process *bounds propagation*. Bounds propagation can often determine that a set of constraints is unsatisfiable relative to given bounds on variables. In the interval method for solving a set of constraints one iteratively selects the numerical variable with the largest allowed interval (as determined by the bounds on that variable). After selecting a numerical variable one computes the midpoint of the interval associated with that variable and then nondeterministically either increases the lower bound to the midpoint or decreases the upper bound to the midpoint. Any solution to the constraint set must be consistent with one of these two choices. Bounds propagation is run in each case and in many cases bounds propagation prunes one of the two possible selections. This nondeterministic splitting of intervals is repeated until a unique floating point value is found for each variable.[3] Floating point lifting, combined with the interval method of solving numerical constraints, can efficiently find ground values of the above "generate and test" floating point expression.

It is also possible to define a constraint satisfaction problem (CSP) lifting transformation. This lifting transformation is appropriate for solving classical constraint satisfaction problems such as the eight queens puzzle, or line labeling problems in vision [Kirousis and Papadimitriou, 1988], [van Hentenryck, 1989], [McAllester,

[3]The interval method described in [Hansen, 1968] and [Hansen and Walster, 1991] apply to constraints on real numbers rather than to floating point representations.

1990]. CSP lifting is described in the setting of the Common Lisp implementation in [Siskind and McAllester, 1992a]. A variety of examples of constraint transformed programs that run under our Common Lisp implementation is also given in [Siskind and McAllester, 1992a].

8 Discussion

Nondeterministic programs provide a natural formal representation of a wide variety of search problems. Lifting is formulated here as an efficiency motivated transformation on nondeterministic programs. Lifted computation constructs a set of declarative constraints from a given nondeterministic program. In fact, it seems appropriate to view lifting as a general method of translating a search problem expressed as a nondeterministic program into a constraint satisfaction problem expressed as a set of declarative constraints. In many cases one can find general purpose methods of solving declarative constraints that are vastly more efficient than simple backtracking in the original procedural nondeterministic program. This is true of most classical, floating point, Boolean, or CSP lifting.

To summarize, lifted computation translates procedural search programs into systems of declarative constraints. Unlike procedural programs, systems of constraints admit inference techniques. Inference is a powerful tool in pruning search. Lifting improves efficiency by allowing inference techniques to be applied to search problems originally expressed procedurally.

If inference is to be used as a tool for improving efficiency it is imperative that inference be done quickly. In the case of classical lifting, unification is a fast and semantically complete procedure for deriving all equations (bindings) that follow from a given set of constraints. In other lifting transformations, semantically complete inference requires exponential time in the worst case. Fortunately there are fast but incomplete inference techniques which seem quite effective in improving search efficiency [van Hentenryck, 1989] [McAllester, 1990].

References

[Bürckert, 1990] Hans-Jürgen Bürckert. A resolution principle for clauses with constraints. In *CADE-10, LNAI 449*, pages 178–192. Springer-Verlag, 1990.

[Colmerauer, 1986] A. Colmerauer. *Logic Programming and Its Applications*, chapter Theoretical Model of PrologII, pages 181 – 200. Ablex Series in Artificial Intelligence. Ablex Publishing Corporation, 1986.

[Davis and Putnam, 1960] M Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3), July 1960.

[Floyd, 1967] Robert Floyd. Nondeterministic algorithms. *JACM*, 14(4):636–644, October 1967.

[Hansen and Walster, 1991] E. R. Hansen and G. W. Walster. Nonlinear equations and optimization. *Control and Games of Comput. Math. Appl.*, 1991. To appear in the second special issue on Global Optimization.

[Hansen, 1968] E. R. Hansen. On the solution of linear algebraic equations using interval arithmetic. *Mathematical Computation*, 22:153–165, 1968.

[Haynes, 1987] Christopher T. Haynes. Logic continuations. *JLP*, 4:157–176, 1987.

[Jaffar and Lassez, 1987] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of POPL-87*, pages 111–119, 1987.

[Kirousis and Papadimitriou, 1988] L. M. Kirousis and C. H. Papadimitriou. The complexity of recognizing polyhedral scenes. *Journal of Computer and Systems Science*, 37(1):14–38, 1988.

[Knuth and Bendix, 1969] Donald E. Knuth and Peter B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Word Problems in Universal Algebras, pages 263–297. Pergamon Press, Oxford, England, 1969.

[Martin and Nipkow, 1990] Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. In *CADE-10, LNAI 449*, pages 365–380. Springer-Verlag, 1990.

[McAllester and Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *AAAI-91*, pages 634–639. Morgan Kaufmann Publishers, July 1991.

[McAllester, 1990] David McAllester. Truth maintenance. In *Proceedings AAAI90*, pages 1109–1116. Morgan Kaufmann Publishers, 1990.

[McCarthy, 1967] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Systems*. North-Holland, 1967.

[Milner, 1977] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–23, 1977.

[Peterson, 1990] Gerald E. Peterson. Complete sets of reductions with constraints. In *CADE-10, LNAI 449*, pages 381–395. Springer-Verlag, 1990.

[Plotkin, 1977] Gordon Plotkin. Lcf considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1), January 1965.

[Siskind and McAllester, 1992a] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. Submitted to LFP92, 1992.

[Siskind and McAllester, 1992b] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. Submitted to LFP92, 1992.

[van Hentenryck, 1989] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.