AUCS / TR9805

Constraint Handling in Common LISP

S. White & D. Sleeman

Department of Computing Science
King's College
University of Aberdeen
Aberdeen AB24 3UE
Scotland, U.K.

Email: {swhite,dsleeman}@csd.abdn.ac.uk

December 1998

Abstract

We demonstrate how constraint programming can be achieved in Common LISP, and share our experience of the Common LISP Constraints Package, SCREAMER. We found the package to be a very useful basis for constraint programming in LISP, but were surprised to see that it provides very little support for combining constraints with many typical LISP data structures. We have addressed these shortcomings by providing an additional library of functions, called SCREAMER+. This extension develops the constraint handling package of SCREAMER in three major directions. Firstly, it provides facilities for expressing and manipulating constraints on LISP lists (including lists interpreted as sets). Secondly, it extends the capabilities for combining constraints with higher order[1] functions in LISP (including logical predicates such as **some**, and **every**). Lastly, it includes functions for dealing with constraints on object-oriented representations in the form of CLOS objects.

Keywords: Constraints, LISP, SCREAMER, CLOS.

1. A higher order function in LISP is a function which accepts another function as an argument.

1 Introduction

SCREAMER (Siskind & McAllester, 1993; Siskind & McAllester, 1994) is a freely available extension of Common LISP (Steele, 1990) that provides for *nondeterministic* and *constraint-based programming*. It was developed at the MIT AI Laboratory and the University of Pennsylvania, and is portable across most modern Common LISP implementations. It provides a very useful basis for constraint programming in LISP, but we found it lacking in some important aspects. For example, although SCREAMER appears to provide ample facilities for efficient numeric constraint handling, it is surprising to see that it provides very little support for many usual LISP operations, particularly the manipulation of lists. We have addressed these and other shortcomings by providing an additional library of functions, called SCREAMER+, to extend the original functionality of SCREAMER. This library contains many important new functions, including those for expressing and manipulating constraints on LISP lists (including lists interpreted as sets), some additional higher order constraint functions (such as a constraint-based version of **`mapcar`**), and also some functions for dealing with constraints on CLOS[1] objects.

In the following section, we sketch the features of SCREAMER, and comment on our experiences of using it. We explain how SCREAMER integrates nondeterminism into LISP and summarise the facilities provided (and not provided) by the constraints package. Section 3 addresses perceived shortcomings of SCREAMER by describing our extension, SCREAMER+. Section 4 describes two example applications which use some of the functions introduced by SCREAMER+. Finally, section 5 summarises our achievements and provides some brief concluding remarks.

2 The Features of SCREAMER

2.1 Nondeterminism using SCREAMER

To add nondeterminism to LISP, SCREAMER defines a *choice point* operator and a *fail* operator. Choice points are most easily introduced with the macro **`either`**, and failures are generated with the function **`fail`**. Functions and other LISP expressions which define choice points but do not attempt to retrieve values from them are called *nondeterministic contexts*. A small number of new functions and macros are also supplied by SCREAMER for retrieving values from nondeterministic contexts, notably **`one-value`** and **`all-values`**. Let us now illustrate these ideas with the following simple example:

```
> (one-value (either 'black 'blue))
BLACK
```

1. CLOS is the *Common LISP Object System*, a package for object-oriented programming in LISP which was adopted by the ANSI committee X3J13 in 1988 as part of the Common LISP standard.

The special form **`either`** has set up a choice point, providing a nondeterministic context for retrieving a single value with the macro **`one-value`**. If the same non-deterministic context is supplied to the macro **`all-values`**, then the list **`'(BLACK BLUE)`** is returned instead of the single symbol **`'BLACK`**. Consider also the following examples, which explore the possible values at *two* choice points:

```
> (one-value (list (either 'tall 'short) (either 'fat 'thin)))
(TALL FAT)
> (all-values (list (either 'tall 'short) (either 'fat 'thin)))
((TALL FAT) (TALL THIN) (SHORT FAT) (SHORT THIN))
```

`Either` can be called with any number of expressions as arguments, and initially returns the value of the first supplied expression to the surrounding nondeterministic context. If a failure later causes a backtrack to this choice point, the next expression will be chosen and returned. If SCREAMER backtracks to a choice point which has no further values, then backtracking continues to the previous choice point. If no choice points remain, an error is generated.

`Either` also forms the basis of several other more specialised nondeterministic forms. For example, consider defining a function **`an-integer-between`**, which nondeterministically returns integers between two supplied integers *low* and *high*:

```
(defun an-integer-between (low high)
  (when (or (not (integerp low)) (> low high)) (fail))
  (either low (an-integer-between (1+ low) high))
  )
```

The first line of this definition traps unexpected parameter values and halts recursion; the second line nondeterministically returns either the integer *low* itself, or, should that fail, an integer between *1+low* and *high*. A similar function can be defined to nondeterministically return the members of a list. SCREAMER already provides many of the commonly used nondeterministic functions, as well as some macros for protecting LISP variables from any local side-effects caused by backtracking.

Nondeterministic programming using SCREAMER is exemplified by a solution[2] to the N-Queens problem, as given in Figure 1.

2.2 Constraint Handling using SCREAMER

The SCREAMER constraints package provides LISP functions which enable a programmer to *create constraint variables*, (often referred to simply as *variables*), *assert constraints* on those variables, and *search for assignments of values to variables* according to the asserted constraints.

An unbound constraint variable, *x*, can be created with a call to the function **`make-variable`**; for example: **`(setq x (make-variable))`**. Assertions are carried out using *constraint primitives*, which are generally constraint-based counterparts of some Common LISP function. So, for example, **`integerpv`** is the SCREAMER constraint primitive corresponding to the LISP function **`integerp`**, which determines whether its argument is an integer. It is a convention that constraint primitives end in the letter v,

2. The code is adapted from (Siskind and McAllester, 1993).

```
(defun attacks-p (qi qj distance)
  (or (= qi qj)
      (= (abs (- qi qj)) distance)))

(defun check-queens (queen queens &optional (distance 1))
  (unless (null queens)
    (if (attacks-p queen (first queens) distance) (fail))
    (check-queens queen (rest queens) (1+ distance))))

(defun n-queens (n &optional queens)
  (if (= (length queens) n)
      queens
    (let ((queen (an-integer-between 1 n)))
      (check-queens queen queens)
      (n-queens n (cons queen queens)))))

(defun queens (n)
  (dolist (sol (one-value (n-queens n)))
    (dotimes (c n)
      (format t " ~a" (if (= (1- sol) c) "Q" "+")))
    (terpri)
    )
  )
```

Figure 1: A Solution to the N-Queens Problem using nondeterministic programming in SCREAMER

because each of them creates and returns a constraint variable to hold the result of evaluating the expression. To assert truths about constraint variables, one uses the primitive **`assert!`**, which takes an expression as its argument and binds the constraint variable associated with that expression to true. Consider the expression **`(assert! (integerpv x))`**, which constrains *x* to be an integer. Likewise, the expression **`(assert! (andv (>=v x 0) (<=v x 10)))`** sets the inclusive range of *x* to be between 0 and 10. Once the problem has been set up in this way, solutions can be found with the nondeterministic function **`solution`**. This function explores the search space of domain values, returning each solution found to its surrounding nondeterministic context. Two arguments must be supplied to **`solution`**: firstly a data structure (typically a list) containing those constraint variables requiring assignments, and secondly, a function for ordering the variables at each choice point. The simplest of these is a static-ordering which linearly forces the constraint variables to assume each value in its domain in turn, until a solution is found. So, to find all possible values of x in the above example, one uses **`(all-values (solution x (static-ordering #'linear-force)))`**.

2.3 Discussion

We were impressed with the way that SCREAMER integrates nondeterminism and constraint-handling into Common LISP. It appears to be very good for solving sets of logical and/or numeric constraints; indeed, it is superior to many other constraint solvers in its ability to deal with sets of non-linear constraints. We were surprised, however, at

the lack of facilities for symbolic constraint handling, such as the expression of constraints on lists. Two of the major features of Common LISP are its list handling and its provision of higher-order functions (functions which take functions as arguments), such as **`mapcar`**. We therefore felt that in order to maximise the utility of constraints in LISP, we should extend SCREAMER to embrace those qualities. In addition, we felt the ability to impose constraints on Common LISP objects would be useful for more sophisticated data/knowledge modelling problems. The functions of our extension to SCREAMER, SCREAMER+, have been designed to provide the required functionality for these three main directions.

To assess the scope of SCREAMER and its extension, SCREAMER+, refer to tables 1 and 2. Table 1 summarises the constraint primitives provided by SCREAMER, and table 2 summarises the additional primitives provided by SCREAMER+. The primitives of SCREAMER are documented in (Siskind, 1991); those of SCREAMER+ are described in the following section of this document.

Type Restrictions:	**`numberpv realpv integerpv booleanpv memberv`**
Boolean:	**`andv orv notv`**
Numeric:	**`<v <=v >v >=v =v /=v +v -v *v /v minv maxv`**
Expression:	**`equalv`**
Function:	**`funcallv applyv`**

Table 1: The Constraint Primitives of SCREAMER

We believe that SCREAMER is difficult for seasoned LISP programmers to grasp initially, because its programs tend to be a mixture of nondeterministic and/or constraint-based programming, and conventional LISP functions. At first, we found it easier to write nondeterministic programs than constraint-based ones. However, we believe that the efficient search procedure[3] of the constraints package combined with its superior ability to deal with infinite domains makes it a more attractive approach for most applications. For this reason, we have only extended the constraints package of SCREAMER; the underlying nondeterminism of SCREAMER remains unchanged.

Another advantage of the constraints package is that it lends itself to an interactive session, because it is easy to inspect intermediate results. For example, this is what happens when creating a constraint variable:

```
> (setq x (make-variable))
[72]
```

3. The constraints package of SCREAMER includes five kinds of inference process to improve its search performance over naive backtracking. The inference processes are: binding propagation, Boolean constraint propagation (BCP), generalised forward checking (GFC), bounds propagation on numeric variables, and unification.

Type Restrictions[a]:	**listpv conspv symbolpv stringpv typepv**
Boolean:	**impliesv**
Expression:	**ifv make-equal**
Lists[b]:	**carv cdrv consv firstv secondv thirdv fourthv nthv subseqv lengthv appendv make-listv all-differentv**
Sets and Bags:	**set-equalv subsetpv intersectionv unionv bag-equalv**
Arrays:	**make-arrayv arefv**
Objects:	**make-instancev classpv slot-valuev class-ofv class-namev slot-exists-pv reconcile**
Higher Order Fns:	**funcallinv mapcarv maplistv everyv somev noteveryv notanyv at-leastv at-mostv exactlyv constraint-fn**
Stream Output:	**formatv**

a.We also define corresponding constraint variable generators: **a-listv**, **a-consv**, **a-symbolv**, **a-stringv**, and **a-typed-varv**.
b. Surprisingly, a function **listv** is unnecessary because the LISP function **list** already works with constraint variables as arguments.

Table 2: The Additional Constraint Primitives of SCREAMER+

The structure associated with a new constraint variable has been returned, and tagged with a unique number for that session, in this case 72. Look what happens when we assert it to be an integer:

```
> (assert! (integerpv x))
NIL                              ; assert! always returns nil
> x                              ; Check the value of x
[72 integer]
```

Now when we inspect the value of the variable, SCREAMER reminds us that it is constrained to be an integer. When we constrain the integer values that it can take, this information is also made visible:

```
> (assert! (andv (>=v x 0) (<=v x 10)))
NIL
> x
[72 integer 0:10 enumerated-domain:(0 1 2 3 4 5 6 7 8 9 10)]
```

The entry 0:10 in the returned value indicates the lower and upper bounds of the range of values which the variable can take, and the *enumerated domain* represents the *actual* values which it can take within the given range. Feedback such as this is not only reassuring when things are going well, it also provides invaluable debugging information when things are not working as expected.

Not all constraints have an immediate effect on the domain values of variables, however. Sometimes, the condition associated with the constraint is wrapped up into a small body of executable code (implemented as a lambda function) and stored as a so-called *noticer*. Noticers are used later, for example, when further assertions are made on the variable, but often not until the associated variable becomes bound as part of the search process. As an example of this, consider a further assertion on the example we have already developed:

```
> (assert! (notv (=v x 2)))
NIL
> x                                ; Check the value of x
[72 integer 0:10 enumerated-domain:(0 1 2 3 4 5 6 7 8 9 10)]
```

The assertion appears not to have reduced the domain of the variable. But the search for solutions reveals that the constraint has not been forgotten:

```
> (all-values (solution x (static-ordering #'linear-force)))
(0 1 3 4 5 6 7 8 9 10)
```

Nevertheless, we feel that since the inspection of intermediate values is such an advantage, the propagation of constraints to domain values should occur as early as possible[4]. This may also enable some reasoning about the value of the solution even before the search process starts. In SCREAMER+, therefore, we have attempted to propagate values as early as practicable.

Unfortunately, SCREAMER lacks the ability to *retract* constraints, so that some previous problem solving state can be resumed. This is important because the assertion of a constraint which ultimately fails can irreparably damage the values of constraint variables[5]. We found that this problem can be circumvented either by taking copies[6] of constraint variable structures *before* dangerous assertions are made, or checking first that the assertion does not fail by using the SCREAMER macro **`possibly?`**.

We found that in some cases, SCREAMER had not carried some of the finer detail of LISP through to its constraint-based counterparts. Most importantly, the SCREAMER function **`equalv`** had not been defined in terms of the LISP function **`equal`**, but in terms of the function **`eq`** instead! This led to some unexpected results; for example, **`(equalv "foo" "foo")`** returned **`nil`**. This probably reflects the fact that SCREAMER was originally designed for solving constraint problems with *atomic* (e.g. numbers, Boolean values), rather than *symbolic* solutions (such as lists). In our copy of SCREAMER, we have changed the definition of **`equalv`** so that it does mirror the LISP definition of **`equal`**.

---

4. As long as the propagation does not impinge severely on efficiency.
5. This does not apply to nondeterministic failure within the scope of a call to **`solution`**.
6. Constraint variables are stored in LISP structures, so a temporary copy of its state can be made by the LISP function **`copy-structure`**

Similarly, we noticed that **memberv** does not accept a keyword *test* argument, like the Common LISP function **member**. (Recall that **member** tests for the equality of some value against each of the members of some list using a test function, which can be supplied as an argument, but is **eq** by default). Thus, it is not possible in SCREAMER to constrain a variable to be one of, say, three candidate lists. In our current version of SCREAMER+, we have changed the definition of **memberv** so that it always uses the test **equal** instead of **eq**. This is sufficient to allow us to work with compound structures such as lists.

As a final observation, we found that the SCREAMER function (**funcallv** $f\ x_1 \ldots x_n$) makes an *idempotency assumption* of its function argument. That is, rather than call the supplied function just once as soon as the other arguments became bound, the function was called many times! Often, this is not too detrimental, impinging only on efficiency and not on correctness. On the occasions when the function *f* has side-effects, however, the difference can be crucial. The problem is simple to illustrate with the following example:

```
> (setq a (make-variable))
[1]
> (funcallv #'print a)        ; Print a when it becomes bound
[2]
> (assert! (equalv a 'hello))
HELLO
HELLO
HELLO
NIL
```

As soon as the constraint variable became bound, the message “HELLO” was output not once, but three times! Note that this particular problem has been addressed by the function **formatv** in SCREAMER+, but the idempotency assumption of **funcallv** remains.

3 The Extension to SCREAMER

3.1 Type Restrictions

---

listpv, conspv, symbolpv, stringpv [Macros]

Synopsis: (**listpv** *x*) | (**conspv** *x*) | (**symbolpv** *x*) | (**stringpv** *x*)

Description: The macros **listpv**, **conspv**, **symbolpv**, and **stringpv** each return a variable, *z,* constrained to indicate with a boolean value whether *x* is of the given type. For example, the variable returned by **listpv** indicates whether its argument is a list as soon as the argument itself becomes bound. Each of these macros is a specialisation of the **typepv** function.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[432]
;;; Constrain z to be the listp of x
> (setq z (listpv x))
[433]
;;; Bind x to a non-list value
> (make-equal x "hello")
"hello"
;;; Inspect the value of z
> z
NIL

;;; Constrain a to be one of the given values
> (setq a (a-member-ofv '(1 nil t (hello))))
[124 enumerated-domain:(1 NIL T (HELLO))]
;;; Assert a to be a list
> (assert! (listpv a))
NIL
;;; The possible values are now only those which are lists
> a
[124 nonnumber enumerated-domain:(NIL (HELLO))]
```

---

typepv [Function]

Synopsis: (**typepv** *x type*)

Description: This function returns a variable, *z*, constrained to indicate with a boolean value whether *x* is of the given type. The supplied *type* must be one of those acceptable to the Common LISP *typep* function. The output variable *z* does not become bound until both *x* and *type* are bound. However, if *z* is bound to true, *type* is bound, and *x* has an enumerated domain, then all values are eliminated from the domain which are not of the supplied type.

Examples

```
> (setq x (make-variable))
[434]
> (setq ty (make-variable))
[435]
> (setq z (typepv x ty))
[436]
> (make-equal x "jolly")
"jolly"
> (make-equal ty 'string)
STRING
> z
T
```

`a-listv`, `a-consv`, `a-symbolv`, `a-stringv` [Macros]

Synopsis: (**`a-listv`**) | (**`a-consv`**) | (**`a-symbolv`**) | (**`a-stringv`**)

Description: The macros **`a-listv`**, **`a-consv`**, **`a-symbolv`** and **`a-stringv`** are commonly used specialisations of the function **`a-typed-varv`**, which are used to constrain a variable to be either a list, cons, symbol, or string, respectively. See also **`a-typed-varv`**.

Examples

```
> (setq z (a-listv))
[100]
> (make-equal z 'not-a-list nil)
Warning: (make-equal Z 'NOT-A-LIST) failed
  Z = [100]; 'NOT-A-LIST = NOT-A-LIST
NIL
```

`a-typed-varv` [Function]

Synopsis: (**`a-typed-varv`** *type*)

Description: This function returns a variable, *z*, constrained to be of the given LISP type. An attempt to bind *z* to a value which is incompatible with the supplied type will fail.

Examples

```
> (setq z (a-typed-varv 'string))
[97]
> (make-equal z '(1 2) 'failed)
Warning: (make-equal Z '(1 2)) failed
  Z = [97]; '(1 2) = (1 2)
FAILED

> (setq a (a-typed-varv 'string))
[98]
> (assert! (memberv a '(1 two "three")))
NIL
> a
"three"
```

3.2 Boolean Values

---

impliesv [Function]

Synopsis: (**impliesv** *x y*)

Description: This function is a simple extension to SCREAMER which returns a boolean variable, *z*, constrained to indicate whether *x* implies *y*. Typically, the implication is asserted, and *y* becomes bound to true as soon as *x* becomes true.

```
;;; Create two boolean constraint variables, x and y
> (setq x (a-booleanv))
[360 Boolean]
> (setq y (a-booleanv))
[361 Boolean]
;;; Assert that x implies y
> (assert! (impliesv x y))
NIL
;;; Set x to be true
> (make-equal x t)
T
;;; Check that y is bound to true
> y
T

;;; The truth table for implies is easy to generate:
> (setq p (a-booleanv))
[3558 Boolean]
> (setq q (a-booleanv))
[3559 Boolean]
> (setq r (a-booleanv))
[3560 Boolean]
> (assert! (equalv r (impliesv p q)))
NIL
> (all-values (solution (list p '=> q 'is r) (static-ordering
#'linear-force)))
((T => T IS T) (T => NIL IS NIL) (NIL => T IS T) (NIL => NIL IS T))
```

3.3 Expressions

---

ifv [Macro]

Synopsis: (**ifv** *condition expression1 expression2*)

Description: This macro returns a variable, *z*, constrained to be the value of *expression1* or *expression2*. If the boolean constraint variable *condition* becomes bound to **t**, then *z* becomes bound to *expression1*; otherwise *z* becomes bound to *expression2* as soon as *condition* becomes bound to **nil**. An important property of **ifv**

is that neither *expression1* nor *expression2* are actually evaluated until the *condition* becomes bound. This means that the macro can be used in recursive constraint definitions, such as that given below.

Examples

```
;;; Create our own recursive definition of memberv
> (defun my-memberv (m ll)
    (when ll
      (ifv (equalv m (carv ll))
           t
           (my-memberv m (cdrv ll))
           )
      )
    )
MY-MEMBERV
> (setq x (make-variable))
[11731]
> (setq z (my-memberv 1 x))
[11741]
> (make-equal x '(3 2 1))
(3 2 1)
> z
T
```

`make-equal` [Macro]

Synopsis: (**`make-equal`** *x y* [*failure-expression*])

Description: This macro attempts to constrain *x* to be **`equal`** to *y*. Although the arguments are actually symmetric, *x* is usually a constraint variable, and *y* some new (constrained) value, so that the macro amounts to the equivalent of an assignment in imperative language programming. The macro returns the updated value of *x* if the assertion succeeded, which is now constrained to be **`equal`** to *y*. If the assertion fails, then the *failure-expression* is evaluated and returned. Also, unlike a conventional SCREAMER **`(assert! (equalv x y))`**, a failed **`make-equal`** assertion leaves *x* with the same value after the assertion as it had before the assertion was attempted.

Examples

```
;;; First, an example of make-equal succeeding...
> (setq x (make-variable))
[4]
> (make-equal x '(1 2 3))
(1 2 3)
> x
(1 2 3)

;;; Now an example of the assertion failing...
> (setq a (an-integer-betweenv 1 5))
[10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]
> (make-equal a 9 nil)
```

```
Warning: (make-equal A 9) failed
  A = [10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]; 9 = 9
NIL
> a
[10214 integer 1:5 enumerated-domain:(1 2 3 4 5)]
>
```

3.4 Lists and Sequences

`firstv`, `secondv`, `thirdv`, `fourthv` [Macros]

Synopsis: (**`firstv`** *x*) | (**`secondv`** *x*) | (**`thirdv`** *x*) | (**`fourthv`** *x*)

Description: The functions **`firstv`**, **`secondv`**, **`thirdv`**, and **`fourthv`** each return a variable, *z,* constrained to be either the first, second, third, or fourth element of a list, respectively. If *x* is already bound to a list value at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *x* becomes bound to a value which is not a list, so that the core function **`first`**, **`second`**, **`third`**, or **`fourth`** cannot be properly executed, then a failure is generated. Note that **`firstv`** is identical to **`carv`**, except that **`firstv`** is implemented as a macro and **`carv`** is implemented as a function.

Examples

```
;;; If the argument is bound the answer is returned directly
> (firstv '(a b c))
A

;;; Create an unbound constraint variable, x
> (setq x (make-variable))
[813]
;;; Constrain z to be the first element of x
> (setq z (firstv x))
[814]
;;; Bind x to a specific value
> (make-equal x '(a b c))
(A B C)
;;; Inspect z
> z
A
```

---

nthv [Function]

Synopsis: (**nthv** *n x*)

Description: The function **nthv** returns a variable, *z,* constrained to be the n^{th} element of the list *x*. Both *n* and *x* may be either a bound value or an unbound constraint variable at the time of function invocation. The argument *n,* when bound, is an integer which can range from 0, which indexes the first element of the list, to (length *x*) - 1, which indexes the last element. As soon as *x* becomes bound, the value of *z* is computed. Similarly, if *z* and *n* both become bound before *x*, then *x* is constrained (but not bound) to be a list whose n^{th} element is *z*. If the bindings are such that the index *n* is out of range for the list *x*, or *x* is not a list at all, a failure is generated.

Examples

```
;;; This example extracts the nth element of a list
> (setq x (make-variable))
[823]
> (setq z (nthv 2 x))
[824]
> (make-equal x '(a b c d e f))
(A B C D E F)
> z
C

;;; This example sets the nth element of a list
> (setq a (make-listv 4))
([396] [395] [394] [393])
> (assert! (equalv (nthv 1 a) 'foo))
NIL
> a
([413] FOO [394] [393])
> (secondv a)
FOO
```

---

subseqv [Function]

Synopsis: (**subseqv** *x start* [*stop*])

Description: The function **subseqv** returns a variable, *z*, constrained to be the subsequence of *x* running from the inclusive index *start* up to the exclusive index *stop*. (The index of the first value in a sequence is zero.) If *stop* is not supplied, the subsequence *z* runs from *start* to the end of *x*. The variable *z* becomes bound as soon as its arguments become bound. Also, if *z* should become bound to a list value, for example before the elements of *x* are all bound, then the elements of *z* are unified with their respective elements of *x*.

Examples

```
;;; When the args are bound, subseqv is the same as subseq
> (subseqv '(1 2 3 4 5 6) 2 4)
(3 4)

;;; Create a constraint variable for the index
> (setq n (make-variable))
[199]
;;; Constrain a to be a subsequence of '(1 2 3 4 5 6)
> (setq a (subseqv '(1 2 3 4 5 6) n))
[201]
;;; Bind n
> (make-equal n 3)
3
;;; Check the value of a
> a
(4 5 6)

;;; Create a list of 6 elements
> (setq x (make-listv 6))
([203] [204] [205] [206] [207] [208])
;;; Create another list of 2 elements
> (setq z (make-listv 2))
([210] [211])
;;; Say that the elements of z are both integers
> (assert! (integerpv (firstv z))
)
NIL
> (assert! (integerpv (secondv z)))
NIL
> z
([210 integer] [211 integer])
;;; Assert that z are the same as the middle two of x
> (assert! (equalv z (subseqv x 2 4)))
NIL
;;; Check that the elements of z have been unified with x
> x
([203] [204] [205 integer] [206 integer] [207] [208])
>
```

`lengthv` [Function]

Synopsis: (**`lengthv`** *x* [*is-list*])

Description: The function **`lengthv`** returns a variable, *z*, constrained to be the length of the list *x*. If *x* is already bound at the time of function invocation, then the length of *x* is returned directly. Otherwise *z* becomes bound as soon as *x* becomes bound. Although **`lengthv`** will work with other types of sequences, such as strings and one-dimenional arrays, its propagation properties have been optimised for use with lists. If *z* should become bound before *x* and the optional argument *is-list* is not supplied (or is supplied with a non-**`nil`** value), then *x* is assumed to be a list and becomes bound to a list structure of the length given by *z*. If *z* becomes bound before *x*, and *is-list* is sup-

plied with the value **nil**, then no assumptions are made about the sequence type of *x*, but no propagation takes place from *z* to *x* either. Also, if *x* has an enumerated domain of possible values, then the lengths of these sequences are propagated through to *z*.

Examples

```
> (setq a (a-listv))
[104]
> (setq b (lengthv a))
[107]
> (make-equal b 4)
4
> a
([112] [111] [110] [109])


;;; Create a constraint variable called x
> (setq x (make-variable))
[185]
;;; Constrain len to be the length of x, where x is not a list
> (setq len (lengthv x nil))
[187 integer]
;;; Enumerate the possibilities for x
> (assert! (memberv x '("one" "two" "three" "four")))
NIL
;;; Check the domain of x
> x
[185 nonnumber enumerated-domain:("one" "two" "three" "four")]
;;; Check the domain of len
> len
[187 integer 3:5 enumerated-domain:(3 5 4)]
;;; Bind x
> (make-equal x "three")
"three"
;;; Check that len has also been bound
> len
5
```

---

consv [Function]

Synopsis: (**consv** *x y*)

Description: The function **consv** returns a variable, *z*, constrained to be the **cons** of *x* and *y*. If *y* is bound at the time of function invocation then *z* is immediately bound to the **cons** of *x* and *y* (regardless of whether *x* is bound); otherwise *z* becomes bound to the **cons** of *x* and *y* as soon as *y* becomes bound[7]. If *z* should become bound before *y*, then *x* and *y* become bound to the **car** and **cdr** of *z*, respectively.

7. We should also like to bind *z* if *x* is bound and *y* is unbound, so that `(car z)` would return *x*. The problem is that if the tail of *z* becomes bound to a constraint variable because *y* is not bound, then LISP stores *z* as a dotted list pair whose `cdr` is an unbound constraint variable (rather than a list). This dotted list pair cannot later be unified with a list when *y* eventually becomes bound.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[1]
;;; Constrain z to be the cons of the symbol ‘g and x
> (setq z (consv 'g x))
[2]
;;; Bind x to a value
> (make-equal x '(1 2 3))
(1 2 3)
;;; Inspect the value of z
> z
(G 1 2 3)


;;; Create two constraint variables to be cons’ed
> (setq x (make-variable))
[7]
> (setq y (make-variable))
[8]
;;; Constrain z to be the cons of x and y
> (setq z (consv x y))
[9]
;;; Bind the value of z
> (make-equal z '(a b c))
(A B C)
;;; Check that the values of x and y have been derived
> x
A
> y
(B C)
```

---

carv [Function]

Synopsis: (**carv** *x*)

Description: The function **carv** returns a variable, *z,* constrained to be the *car* (i.e. first element) of a *cons*. (A cons is either a normal list or a dotted list.) If *x* is already bound to a cons value at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *z* becomes bound before *x*, then *x* is constrained to be a cons in which *z* is the first element. If *x* becomes bound to a value which is not a cons, so that the Common LISP function *car* cannot be properly executed, then a fail is generated.

Examples

```
;;; Create a variable to contain a list
> (setq x (make-variable))
[13]
;;; Constrain the variable z to be the car of x
> (setq z (carv x))
```

```
[14]
;;; Bind x to a value
> (make-equal x '(fee fi fo fum))
(FEE FI FO FUM)
;;; Inspect the value of z
> z
FEE


;;; Create a variable to contain a dotted pair
> (setq x (make-variable))
[153]
;;; Constrain z to be the first element of the pair
> (setq z (carv x))
[154]
;;; Create a binding for x
> (make-equal x (cons 'one 'two))
(ONE . TWO)
;;; Inspect the value of z
> z
ONE
```

See also the examples for **cdrv**.

---

cdrv [Function]

Synopsis: (**cdrv** *x*)

Description: The function **cdrv** returns a variable, *z*, constrained to be the *cdr* (i.e. the tail) of a *cons* (normally a list). If *x* is already bound to a cons at the time of invocation, then the *value* of the constraint variable *z* is returned, rather than the variable itself. Otherwise *x* must be a constraint variable, and *z* becomes bound as soon as *x* becomes bound. If *z* becomes bound before *x*, then *x* is constrained to be a cons which has *z* as its *cdr*. If *x* becomes bound to a value which is not a *cons*, so that the Common LISP function *cdr* cannot be properly executed, then a fail is generated.

Examples

```
;;; Create a constraint variable to contain a list
> (setq x (make-variable))
[133]
;;; Constrain z to be the tail of the list
> (setq z (cdrv x))
[134]
;;; Create a binding for x
> (make-equal x '(1 2 3 4))
(1 2 3 4)
Inspect the value of z
> z
(2 3 4)

> (setq x (make-variable))
[114]
> (setq head (carv x))
```

```
[115]
> (setq tail (cdrv x))
[116]
> (make-equal head 'g)
G
> (make-equal tail '(h i))
(H I)
> x
(G H I)
```

---

appendv [Function]

Synopsis: (**appendv** x_1 x_2 [... x_n])

Description: The function **appendv** returns a variable, z, constrained to be the **append** of all its arguments, unless all the arguments are bound at the time of function invocation, when the value of the append is returned directly. The function applies tail recursion to any arguments x_3 ... x_n supplied. Usually, however, **appendv** will be called with only two arguments, so that z is constrained to be the **append** of x_1 and x_2. In such circumstances, when any two of x_1, x_2 and z become bound, the third value is deduced. If z only should become bound, then x_1 and x_2 are each constrained to be one of the sublists taken from the front or back of z, respectively. For example, if the append, z, of two lists x and y is known to be **'(a b c)**, then x must be one of **'()**, **'(a)**, **'(a b)**, and **'(a b c)**, and y must be one of **'(a b c)**, **'(b c)**, **'(c)**, and **'()**.

Examples

```
;;; Create a constraint variable
> (setq x (make-variable))
[165]
;;; Constrain z to be the append of x and '(3 4)
> (setq z (appendv x '(3 4)))
[166]
;;; Bind x to the list '(1 2)
> (make-equal x '(1 2))
(1 2)
;;; Inspect z
> z
(1 2 3 4)
```

```
;;; Create two constraint variables
> (setq x (make-variable))
[167]
> (setq y (make-variable))
[168]
;;; Assert that appending x to y gives the result '(a b c)
> (assert! (equalv (appendv x y) '(a b c)))
NIL
;;; Inspect x and y - note the enumerated domains
> x
[167 nonnumber enumerated-domain:(NIL (A) (A B) (A B C))]
> y
```

```
[168 nonnumber enumerated-domain:((A B C) (B C) (C) NIL)]
;;; Bind x to one of its possible values
> (make-equal x '(a))
(A)
;;; Inspect y, which has now automatically been bound
> y
(B C)
```

---

make-listv [Function]

Synopsis: (**make-listv** *n*)

Description: The function **make-listv** returns a variable which is constrained to be a list of the length given by *n*. More precisely, the function actually generates a list of length *n* with a different constraint variable taking the place of every element. Note that since a constraint variable is generated for every place in the list, there is a memory overhead in generating large lists using this function. For long lists it is preferable to use **a-listv** which does not actually construct a list, but instead checks that the variable is a list once it is instantiated.

Examples

```
> (setq n (make-variable))
[28]
> (setq x (make-listv n))
[29]
> (make-equal n 4)
4
> x
([33] [32] [31] [30])
> (assert! (equalv (nthv 1 x) 'foo))
NIL
> x
([36] FOO [31] [30])
>
```

---

all-differentv [Function]

Synopsis: (**all-differentv** x_1 x_2 ... x_n)

Description: This function returns a boolean variable, z, constrained to indicate whether all the supplied arguments have distinct values. This is in effect a symbolic equivalent of the existing numeric SCREAMER constraint predicate '**/=v**', which can be used to constrain its numeric arguments to contain distinct values.

Thus, whilst **(assert! (/=v** x_1 x_2 x_3**))** is equivalent to:
(andv (numberpv x_1**) (numberpv** x_2**) (numberpv** x_3**)**
(notv (=v x_1 x_2**)) (notv (=v** x_2 x_3**)) (notv (=v** x_1 x_3**))),**

the symbolic counterpart **(assert! (all-differentv** x_1 x_2 x_3**))** is equivalent to **(andv (notv (equalv**x_1 x_2**)) (notv (equalv**x_2 x_3**)) (notv (equalv**x_1 x_3**)))**.

Examples

```
;;; Clearly all distinct
> (all-differentv 'a 'b 'c)
T

;;; Not yet clear whether the values are distinct
> (all-differentv (a-member-ofv '(a b)) 'b 'c)
[10224 Boolean]

;;; Clearly not all distinct
> (all-differentv 'a 'b 'a)
NIL
```

3.5 Sets and Bags

---

set-equalv [Function]

Synopsis: (**set-equalv** *x y*)

Description: The function **set-equalv** returns a boolean variable, *z*, which is constrained to indicate whether its two list arguments are equal if they are interpreted as sets. In other words, as soon as both *x* and *y* become bound, *z* becomes bound to true if *x* and *y* have the same members; otherwise *z* becomes bound to false.

Examples

```
> (setq x '(a b c))
(A B C)
> (setq y (make-variable))
[283]
> (setq same-set (set-equalv x y))
[284 Boolean]
> (make-equal y '(b b a c a a))
(B B A C A A)
> same-set
T
;;; In this example the second set has 4 distinct members
> (set-equalv '(a b c) '(b b a c a d))
NIL
```

`intersectionv` [Function]

Synopsis: (**`intersectionv`** *x y*)

Description: The function **`intersectionv`** returns a variable, *z*, constrained to be the intersection of the two sets *x* and *y*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If either *x* or *y* has duplicate entries, the redundant entries may or may not appear in the result[8]. Otherwise, if *z* becomes bound before *x* or *y*, then the latter are constrained to contain each of the elements of *z*.

Examples

```
;;; Create a variable x
> (setq x (make-variable))
[11]
;;; Constrain i to be the intersection of x and (a a b c c c)
> (setq i (intersectionv x '(a a b c c c)))
[12]
;;; Bind x
> (make-equal x '(a b d e f))
(A B D E F)
;;; Check that the intersection is now also bound
> i
(B A)

;;; Create a variable x
> (setq x (make-variable))
[107]
;;; Constrain i to be the intersection of (a b c) and x
> (setq i (intersectionv '(a b c) x))
[108]
;;; Bind i
> (make-equal i '(a b))
(A B)
;;; Try to bind x in an inconsistent way
> (make-equal x '(c b a) 'failed)
Warning: (make-equal X '(C B A)) failed
  X = [107]; '(C B A) = (C B A)
FAILED
```

8. **`intersectionv`** uses the Common LISP function **`intersection`**, which leaves the duplication issue open to the LISP implementation.

unionv [Function]

Synopsis: (**unionv** *x y*)

Description: The function **unionv** returns a variable, *z*, constrained to be the union of the two sets *x* and *y*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If either *x* or *y* has duplicate entries, the redundant entries may or may not appear in the result[9]. Otherwise, if *z* becomes bound before *x* or *y*, then the elements of the latter are constrained to be members of *z*.

```
;;; Create a constraint variable
> (setq x (make-variable))
[145]
;;; Constrain u to be the union of x and (a a b c c c)
> (setq u (unionv x '(a a b c c c)))
[146]
;;; Bind x
> (make-equal x '(a b d e f))
(A B D E F)
;;; Check the binding of u
> u
(F E D A A B C C C)

;;; Create a constraint variable
> (setq x (make-variable))
[149]
;;; Constrain u to tbe the union of x and (a b c)
> (setq u (unionv x '(a b c)))
[150]
;;; Bind u first to be (a b c d)
> (make-equal u '(a b c d))
(A B C D)
;;; Try binding x in an inconsistent way
> (make-equal x '(d e f) 'failed)
Warning: (make-equal X '(D E F)) failed
  X = [149]; '(D E F) = (D E F)
FAILED
```

9. **unionv** uses the Common LISP function **union**, which leaves the duplication issue open to the LISP implementation.

---

bag-equalv [Function]

Synopsis: (**bag-equalv** *x y*)

Description: The function **bag-equalv** returns a boolean variable, *z*, which is constrained to indicate whether its two list arguments are equal when interpreted as bags. In other words, as soon as both *x* and *y* become bound, *z* becomes bound to true if *x* and *y* not only have the same members, but also the same number of each of those members; otherwise *z* becomes bound to false.

Examples

```
> (setq x '(a a b c c c))
(A A B C C C)
> (setq y (make-variable))
[300]
> (setq same-bag (bag-equalv x y))
[301 Boolean]
> (make-equal y '(c a c b c a))
(C A C B C A)
> same-bag
T
;;; In this example, the second list has an extra c
;;; So the lists are set-equalv, but not bag-equalv
> (bag-equalv '(a b c) '(a b c c))
NIL
```

---

subsetpv [Function]

Synopsis: (**subsetpv** *x y*)

Description: The function **subsetpv** returns a boolean variable which is constrained to indicate whether $x \subseteq y$ if the two lists *x* and *y* are interpreted as sets. Note that the related function **proper-subsetv** can easily be defined as (**andv** (**subsetpv** *x y*) (**not** (**set-equalv** *x y*))).

```
;;; State the necessary ingredients for some recipe
> (setq ingredients '(eggs flour milk salt sugar))
(EGGS FLOUR MILK SALT SUGAR)
;;; We don't know what ingredients are avaialble yet
> (setq available (make-variable))
[366]
;;; It is only possible to prepare the recipe if the necessary
;;; ingredients is a subset of the available ingredients
> (setq can-prepare (subsetpv ingredients available))
[378 Boolean]
;;; State which ingredients are available
> (make-equal available '(salt pepper milk juice eggs sugar rai-
sins flour))
(SALT PEPPER MILK JUICE EGGS SUGAR RAISINS FLOUR)
```

```
;;; Check that can-prepare has become bound
> can-prepare
T
```

3.6 Arrays

---

make-arrayv [Function]

Synopsis: (**make-arrayv** *d*)

Description: The function **make-arrayv** returns a variable which is constrained to be an array with dimensions given by *d*. More precisely, the function actually generates an array with dimensions *d* in which every element is automatically assigned to be a new, unique, unbound constraint variable. Note that since a constraint variable is generated for every place in the array, there is a memory overhead in generating large arrays using this function. The argument *d* can be either a list of integers or a constraint variable which later becomes bound to a list of integers.

Examples

```
;;; d is a constraint variable which holds the array dimensions
> (setq d (make-variable))
[262]
;;; a is constrained to be an array of dimensions d
> (setq a (make-arrayv d))
[263]
;;; Bind d, so that a is created as a 2 by 3 array
> (assert! (equalv d '(2 3)))
NIL
;;; Check the binding of a
> a
#2A(([266] [265] [264]) ([269] [268] [267]))
```

---

arefv [Function]

Synopsis: (**arefv** *array* &rest *indices*)

Description: The function **arefv** returns a variable which is constrained to be the element of *array* at the position given by *indices*. The arguments to **arefv** can be either bound values or constraint variables. This is a constraint-based version of the Common LISP function **aref**.

Examples

```
> (setq a (make-array '(2 3) :initial-contents '((p q r) (s t
u))))
#2A((P Q R) (S T U))
```

```
> (setq x (make-variable))
[274]
> (setq y (make-variable))
[275]
> (setq val (arefv a x y))
[276]
> (make-equal x 0)
0
> (make-equal y 1)
1
> val
Q
```

3.7 Objects

We have used three basic principles relating to the combination of SCREAMER constraints and Common LISP objects. Firstly, SCREAMER constraint variables can represent either (predefined) *classes*[10], object *instances*, or *slot values* within an object instance. They should not normally contain slot definitions. Secondly, the constraint functions listed here do not define any new classes. This mirrors the approach to class definition in Common LISP, in which all classes must be defined in advance with the **`defclass`** construct. (You cannot **`funcall #'defclass`** anyway, because it is defined as a Common LISP macro.) Thirdly, the only constraint function which creates any new object instances is **`make-instancev`**. Note, however, that very often, programs do not need to use **`make-instancev`**, because it is possible to use the conventional form **`make-instance`**, instead. In such cases, constraint variables may hold the slot values of objects, but not the objects themselves.

It is important to understand an anomaly which arises when dealing with CLOS objects and constraint variables together. Firstly, recall that CLOS objects may have slots which are unbound. That is, the slot contains no value at all, and an attempt to retrieve the value will normally result in an error. Likewise, constraint variables themselves may be unbound. In the case of constraint variables, this means that a structure has been created for maintaining the domain properties of the variable, but the variable has not yet been assigned a definite value. By combining constraint variables and CLOS objects, it is therefore possible to create a bound slot, containing an unbound constraint variable. That is, the Common LISP predicate (**`slot-boundp`** *obj slot-name*) would return **`t`**, whereas the SCREAMER expression (**`bound?`** (**`slot-value`** *obj slot-name*)) would return **`nil`**. One possible approach to this problem is to ensure that the slot of any new class receives by default an **`:initform`** argument at creation time which fills the slot with an unbound constraint variable. The definition of a class would then resemble the following:

```
(defclass my-class ()
  ((slot-1 :accessor slot-1 :initform (make-variable)
    ...
   (slot-n :accessor slot-n :initform (make-variable))))
```

Then, when I create an instance of the class, the slots already contain constraint variables:

10. as would be returned by the Common LISP function **`class-of`**

```
> (setq x (make-instance 'my-class))
#<MY-CLASS @ #x82d01a>
> (describe x)
#<MY-CLASS @ #x82d01a> is an instance of #<STANDARD-CLASS MY-
CLASS>:
 The following slots have :INSTANCE allocation:
  SLOT-N   [234]
  SLOT-1   [235]
> (slot-boundp x 'slot-1)
T
> (type-of (slot-value x 'slot-1))
SCREAMER::VARIABLE
> (bound? (slot-value x 'slot-1))
NIL
```

In the rest of this section, we document the SCREAMER+ functions which enable the fusion of constraint variables and CLOS objects.

`make-instancev` [Function]

Synopsis: **`make-instancev`** *class-name* x_1 x_2 ... x_n

Description: This function returns a variable, *z*, constrained to be an instance of the supplied type. An object instance is created as soon as the class-name becomes bound. If *z* should become bound before the *class-name*, then the class-name is derived, an instance is created from the arguments x_1 ... x_n and the slots of this instance are unified with those of the object *z*.

The function also displays an interesting phenomenon, in that the *identity* of the object is not known until it is actually created, and this can happen much later than at function invocation time, depending on the bindings of the variables. Thus, you cannot **`(assert! (equalv (make-instancev 'junk) (make-instancev 'junk)))`** because each of the newly created object instances has a different identity, and the equality constraint will always fail. However, you *can* assert **`(slots-equalv (make-instancev 'junk) (make-instancev 'junk))`** since **`slots-equalv`** does not check the identity of the objects, but only the equality of the slot values.

Examples

```
> (defclass junk () ((top :initarg :top) (bottom :initarg :bot-
tom)))
#<STANDARD-CLASS JUNK #xF0A3BC>
> (setq a (make-variable))
[7]
> (setq b (make-variable))
[8]
> (setq c (make-variable))
[9]
> (setq d (make-instancev a :top b :bottom c))
[10]
> d
[10]
```

```
> (make-equal a 'junk)
JUNK
> d
#<JUNK #xF24880>
> (describe (value-of d))
#<JUNK #xF243F4> is a JUNK:
TOP:                          [8]
BOTTOM:                       [9]
```

`slot-valuev` [Function]

Synopsis: (**`slot-valuev`** *x y*)

Description: The function **`slot-valuev`** returns a variable, *z*, constrained to contain the value of the slot *y* of either the object *x* or a constraint variable which will later become bound to such an object. If the object contains no such slot, then a *slot-missing* error is generated, as would occur with **`slot-value`** in conventional Common LISP. If the slot value is already bound at the time of function invocation, then that value is returned instead of a constraint variable. Otherwise the variable becomes bound as soon as both the object *x* and the constraint variable contained by the slot *y* become bound. Note that if the value of slot *y* in object *x* is a constraint variable, then using **`(setf (slot-value x y) value)`** would overwrite the constraint variable, and bypass its associated noticers (which perform value propagation). The macro **`setf`** should therefore be avoided when working with slots containing constraint variables. Instead, the function **`slot-valuev`** (or any accessor functions defined) should be used for both reading and writing to the slot, as shown by the example below.

Examples

```
;;; Define a very simple CLOS class
> (defclass person ()
    ((name :accessor name :initarg :name))
    )
#<STANDARD-CLASS PERSON>
;;; Create an instance of the class with the 'name
;;; slot set to a constraint variable
> (setq anon (make-instance 'person :name (make-variable) ))
#<PERSON @ #x544362>
;;; Inspect the object. Note the slot value of name
> (describe anon)
#<PERSON @ #x544362> is an instance of #<STANDARD-CLASS PERSON>:
 The following slots have :INSTANCE allocation:
  NAME   [92]
;;; Constrain name-of-anon to be slot-value 'name of object anon
> (setq name-of-anon (slot-valuev anon 'name))
[92]
;;; Set the slot value
> (make-equal name-of-anon "Fred Bloggs")
"Fred Bloggs"
;;; Inspect the object again. Note the slot value has been set
> (describe anon)
#<PERSON @ #x544362> is an instance of #<STANDARD-CLASS PERSON>:
```

```
 The following slots have :INSTANCE allocation:
  NAME   "Fred Bloggs"
;;; Read the value back from the slot
> (slot-valuev anon 'name)
"Fred Bloggs"
;;; Another way of reading the value
> (name anon)
"Fred Bloggs"
```

---

classpv [Function]

Synopsis: (**classpv** *x y*)

Description: The function **classpv** returns a variable, *z*, constrained to contain a boolean value which indicates whether the object *x* is an instance of the class *y*. If *x* is not an object, or if *y* is bound to a non-existent class, then *z* becomes bound to **nil**. The value of *z* becomes bound as soon as both *x* and *y* become bound.

Also, if *z* and *y* are both bound and *x* has an enumerated domain, then the domain values of *x* are reduced such that *z* = **(classp x y)** is true. For example, if *y* is the classname **'junk**, and *z* becomes bound to **nil** (meaning that *x* is not allowed to be an instance of **junk**), then all instances of the class **junk** will be removed from the domain of *x* (see example below).

Examples

```
;;; Define a simple class
> (defclass junk() ())
#<STANDARD-CLASS JUNK #xF08B9C>
;;; Create a new variable
> (setq x (make-variable))
[1]
;;; Constrain z to indicate whether it is of class 'junk
> (setq z (classpv x 'junk))
[4 Boolean]
;;; Bind x
> (make-equal x (make-instance 'junk))
#<JUNK #xF1EB90>
;;; Inspect z
> z
T

> (setq x (a-member-ofv (list 1 'hello (make-instance 'junk) nil
"hello")))
[17 enumerated-domain:(1 HELLO #<JUNK #xF13440> NIL "hello")]
> (assert! (notv (classpv x 'junk)))
NIL
> x
[17 enumerated-domain:(1 HELLO NIL "hello")]
```

`class-ofv` [Function]

Synopsis: (**`class-ofv`** *x*)

Description: This function returns a constraint variable, *z*, constrained to be the class of the object *x*. The output variable *z* becomes bound as soon as *x* is bound. If *x* has an enumerated domain, then the classes of those values are propagated to the enumerated domain of *z*.

Examples

```
> (defclass junk () ())
#<STANDARD-CLASS JUNK #xF14018>
> (setq x (make-variable))
[31]
> (setq z (class-ofv x))
[32]
> (make-equal x (make-instance 'junk))
#<JUNK #xF16B98>
> z
#<STANDARD-CLASS JUNK #xF14018>
>
```

`class-namev` [Function]

Synopsis: (**`class-namev`** *x*)

Description: This function returns a variable, *z*, constrained to be the class name of the class *x*. As soon as *x* becomes bound, the LISP function *class-name* is applied, and *z* becomes bound to the result. Also, if *x* has an enumerated domain, then the class names of those values are propagated to *z*. Note that *x* must be a class, and not an object. See **`class-ofv`** for deriving the class of a variable constrained to be an object.

Examples

```
> (defclass foo () ())
#<STANDARD-CLASS FOO #xF1F9E4>
> (defclass bar () ())
#<STANDARD-CLASS BAR #xF17528>
> (setq x (make-variable))
[22]
> (setq name (class-namev (class-ofv x)))
[24]
> (assert! (memberv x (list (make-instance 'foo) (make-instance
'bar))))
NIL
> x
[22 nonnumber enumerated-domain:(#<FOO #xF14ABC> #<BAR
#xF14BE0>)]
```

```
> name
[24 nonnumber enumerated-domain:(FOO BAR)]
>
```

`slot-exists-pv` [Function]

Synopsis: (**`slot-exists-pv`** *x y*)

Description: This function returns a variable, *z*, constrained to be a boolean value which indicates whether the slot *y* exists in the object *x*. The output variable *z* becomes bound as soon as both *x* and *y* become bound. If *x* has an enumerated domain and *y* is bound, then the possible values of *x* are checked for the existence of slot *y,* in an attempt to bind *z* as early as possible. Also, if *z* and *y* are both bound, and *x* has an enumerated domain, then this domain (a set of object instances) may be reduced according to the value of *z* and whether each of the instances contains the slot y. This enables some powerful constraint reasoning; for example, "Constrain object *p* not to have a slot called *name*".

Examples

```
;;; Set up two classes, person and house, with different slots
> (defclass person () (name))
#<STANDARD-CLASS PERSON #xF22AB8>
> (defclass house () (number))
#<STANDARD-CLASS HOUSE #xF11968>
;;; Say that x is either a house or a person
> (setq x (a-member-ofv (list (make-instance 'house)
                              (make-instance 'person))))
[38 nonnumber enumerated-domain:(#<HOUSE #xF22D10> #<PERSON
#xF2331C>)]
;;; SCREAMER+ can't know for sure whether the slot 'name
;;; exists in x, so an unbound Boolean is returned
> (slot-exists-pv x 'name)
[39 Boolean]
;;; But x certainly doesn't contain the slot 'town
> (slot-exists-pv x 'town)
NIL
;;; Assert that x should not contain the slot 'name
> (assert! (notv (slot-exists-pv x 'name)))
NIL
;;; This leaves only one possible value for x...
> x
#<HOUSE #xF2277C>
>
```

`reconcile` [Function]

Synopsis: (**`reconcile`** *x y*)

Description: This function unifies the constraint variables *x* and *y* by trying to make them **`equal`**. For example, if *x* is constrained to be an integer between 0 and 10, and *y* is constrained to be an integer between 5 and 15, then after (**`reconcile`** *x y*), both *x* and *y* will be constrained to be an integer between 5 and 10. If *x* and *y* contain (or are) objects, the function ensures that all their respective *slot values* are **`equal`**, rather than the variables themselves. If the slots themselves contain objects, then **`reconcile`** is called recursively on these respective slot values.

In the example below, notice that after **`reconcile`** has been invoked, the two objects still retain their separate object *identities* (as evidenced by the identifiers #x825b72 and #x825f6a), but that the slot values have been made **`equal`**. This required a bidirectional passing of data – both from **`chalk`** to **`cheese`** and from **`cheese`** to **`chalk`**.

Examples

```
> (defclass thing ()
   ((colour :initarg :colour)
    (odour :initarg :odour)
    (name :initarg :name)))
#<STANDARD-CLASS THING>
> (setq chalk (make-instance 'thing :colour 'white))
#<THING @ #x7a83fa>
> (setq cheese (make-instance 'thing :odour 'strong))
#<THING @ #x7a883a>
> (reconcile chalk cheese)
T
> (describe chalk)
#<THING @ #x825b72> is an instance of #<STANDARD-CLASS THING>:
 The following slots have :INSTANCE allocation:
  NAME     [10117]
  ODOUR    STRONG
  COLOUR   WHITE
> (describe cheese)
#<THING @ #x825f6a> is an instance of #<STANDARD-CLASS THING>:
 The following slots have :INSTANCE allocation:
  NAME     [10117]
  ODOUR    STRONG
  COLOUR   WHITE
```

3.8 Higher Order Functions

`mapcarv` [Function]

Synopsis: (**`mapcarv`** $f\ x_1\ x_2\ \ldots\ x_n$)

Description: The function **`mapcarv`** returns a variable, z, constrained to be the **`mapcar`** of f applied to the arguments $x_1\ \ldots\ x_n$. Note that since the elements of the lists $x_1\ \ldots\ x_n$ may be constraint variables, rather than bound values, the function f should be a SCREAMER (or SCREAMER+) constraint function and not a "conventional" Common LISP function. This means, for example, that to constrain a variable to be the list of sums of respective elements of the two lists p and q, you should use **`(mapcarv #'+v p q)`**, and *not* **`(mapcarv #'+ p q)`**. See also the description of **`constraint-fn`**, which, given a conventional LISP function, returns a modified version of the same function which can also deal with constraint variables as arguments.

Note that in order to maximise propagation, we have deviated slightly from the Common LISP definition of **`mapcar`** in that we expect all the lists $x_1\ \ldots\ x_n$ to be of the same length. This allows us to propagate the length of any of the lists $x_1\ \ldots\ x_n$ through to the length of z as soon as they become bound. (When the list arguments to **`mapcar`** are of different lengths, some of the arguments are not used in the result. For example, **`(mapcar #'+ '(1 2 3) '(10 20))`** returns **`'(11 22)`**.)

Examples

```
> (setq a (a-listv))
[183]
> (setq b (a-listv))
[186]
> (setq sums (mapcarv #'+v a b))
[195]
> (make-equal a '(1 2 3))
(1 2 3)
> sums
([207 number] [209 number] [211 number])
> b
([205 number] [204 number] [203 number])
> a
(1 2 3)
> (make-equal b '(10 20 30))
(10 20 30)
> sums
(11 22 33)
```

---

maplistv [Function]

Synopsis: (**maplistv** $f\ x_1\ x_2\ \ldots\ x_n$)

Description: The function **maplistv** returns a variable, z, constrained to be the **maplist** of f applied to the arguments $x_1\ \ldots\ x_n$ (**maplist** applies the function f to successive **cdr**s of the arguments $x_1\ \ldots\ x_2$). As with **mapcarv**, the function f should be a SCREAMER (or SCREAMER+) constraint function and not a "conventional" Common LISP function (see **constraint-fn**).

Note that in order to maximise propagation, we have deviated slightly from the Common LISP definition of **maplist** in that all the lists $x_1\ \ldots\ x_n$ supplied to **maplistv** must be of the same length.

Examples

```
;;; First a reminder of the Common LISP function...
> (maplist #'(lambda(x) (cons 'head x)) '(1 2 3 4))
((HEAD 1 2 3 4) (HEAD 2 3 4) (HEAD 3 4) (HEAD 4))


;;; Now create a constraint variable
> (setq a (make-variable))
[232]
;;; And constrain b to the the maplist of a with given lambda fn.
> (setq b (maplistv (constraint-fn #'(lambda(x) (cons 'head x)))
a))
[233]
;;; Now bind a
> (make-equal a '(1 2 3 4))
(1 2 3 4)
;;; Check the value of b
> b
((HEAD 1 2 3 4) (HEAD 2 3 4) (HEAD 3 4) (HEAD 4))
```

---

everyv, somev, noteveryv, notanyv [Function]

Synopsis: (**everyv** $f\ x$) | (**somev** $f\ x$) | (**noteveryv** $f\ x$) | (**notanyv** $f\ x$)

Description: The functions **everyv**, **somev**, **noteveryv**, and **notanyv** are the constraint-based counterparts of the Common LISP functions **every**, **some**, **notevery**, and **notany**. Thus, for example, the function **everyv** returns a variable, z, constrained to indicate whether the predicate f is true of every element of x. In each case, the predicate f must already be bound to a constraint function (see **constraint-fn**) at the time of function invocation.

Examples

```
;;; Create a list of length 3
> (setq a (make-listv 3))
([189] [188] [187])
;;; Assert that it contains only integers
> (assert! (everyv #'integerpv a))
NIL
;;; Observe the effect of the assertion
> a
([189 integer] [188 integer] [187 integer])

;;; Create a list of length 3
> (setq a (make-listv 3))
([176] [175] [174])
;;; b indicates whether there is a non-integer in the list
> (setq b (noteveryv #'integerpv a))
[182 Boolean]
;;; Bind the first element of a to a non-integer
> (assert! (equalv (firstv a) 'p))
NIL
;;; Inspect a
> a
(P [175] [174])
;;; Check whether b has been bound
> b
T
```

`at-leastv`, **`at-mostv`** [Macros]

Synopsis: (**`at-leastv`** $n\ f\ x_1\ x_2\ \ldots\ x_m$) | (**`at-mostv`** $n\ f\ x_1\ x_2\ \ldots\ x_m$)

Description: The functions **`at-leastv`** and **`at-mostv`** each return a boolean variable, z, constrained to indicate whether the m-argument predicate f is true of the arguments $x_1 \ldots x_m$ *at least* or *at most* n times, respectively. Note that as with the other higher-order functions described in this section, f must be a constraint-based function such as **`integerpv`** or a function returned by **`constraint-fn`**.

Note also that the effect of the related notion *strictly-less-than* can be achieved with **`(notv (at-leastv`** $n\ f\ x_1\ x_2\ \ldots\ x_m$ **`))`**. Likewise, the effect of the notion *strictly-greater-than* can be achieved with **`(notv (at-mostv`** $n\ f\ x_1\ x_2\ \ldots\ x_m$ **`))`**.

Examples

```
> (at-leastv 2 #'integerpv (list (make-variable)))
NIL
> (at-leastv 2 #'integerpv (list 1 (make-variable)))
[594 Boolean]
> (at-leastv 2 #'integerpv (list 1 (make-variable) 'p 6))
T
> (at-leastv 2 (constraint-fn #'evenp) (list 3 4 5 6))
T
```

---

exactlyv [Macro]

Synopsis: (**exactlyv** $n\ f\ x_1\ x_2\ \ldots\ x_m$)

Description: The **exactlyv** form returns a boolean variable, z, constrained to indicate whether the m-argument predicate f is true of the respective elements of the lists $x_1 \ldots x_m$ exactly n times. Note that f must be a constraint-based function, such as **integerpv**, **equalv**, or the returned value of a call to **constraint-fn**.

Note that **exactlyv** is defined such that:

(**exactlyv** $n f x_1 \ldots x_m$) $\Leftrightarrow$ (**andv** (**at-leastv** $n f x_1 \ldots x_m$) (**at-mostv** $n f x_1 \ldots x_m$))

Examples

```
> (exactlyv 3 #'equalv '(1 1 2 3 5 8) '(0 1 2 3 4 5))
T

> (exactlyv 1 #'integerpv '(1 a 2 b))
NIL
```

---

constraint-fn [Function]

Synopsis: (**constraint-fn** f)

Description: This function takes the conventional LISP function f, and returns f', a modified version of the same function which accepts constraint variables as arguments as well as bound LISP values. The function reflects our approach to SCREAMER+, since most of the functions we provide are simply constraint-based counterparts to frequently used LISP functions. In contrast to the other functions of the SCREAMER+ library, however, the functions returned by **constraint-fn** are not optimised for their propagation properties. This means, for example, that **carv** is preferable to **(constraint-fn #'car)**. On the other hand, **constraint-fn** can be used with *any* function known to LISP, providing a good basis for general symbolic constraint-based reasoning.

Examples

```
;;; First, define a simple function which returns a string
> (defun hello (x) (format nil "Hello ~a" x))
HELLO
;;; Test the function
> (hello 'fred)
"Hello FRED"
;;; Get the constraint-based version of HELLO
> (setq hellov (constraint-fn #'hello))
#<closure 0 #xF216E8>
;;; Create a variable for testing
```

```
> (setq who (make-variable))
[216]
;;; Constrain hello-string to be the HELLO of who
> (setq hello-string (funcall hellov who))
[217]
;;; Bind who
> (make-equal who 'mary)
MARY
;;; Check the binding of hello-string
> hello-string
"Hello MARY"
```

`funcallinv` [Function]

Synopsis: (**`funcallinv`** f f^{-1} x_1 x_2 ... x_n)

Description: This is an extended version of **`funcallv`** (supplied with SCREAMER) which allows the programmer to supply an inverse mapping function f^{-1} as well as the "forward" mapping function f. As with **`funcallv`**, if the arguments x_1 ... x_n become bound, the returned constraint variable, z, becomes bound to the result of applying the function f to those arguments. If, however, z becomes bound first, then f^{-1} is applied to it to derive the list of arguments x_1 ... x_n. Also, if the function f applies only to a single argument x_1 (rather than multiple arguments x_1 ... x_n), and this is an unbound constraint variable with an enumerated domain, then the possible values of z are computed and recorded as its enumerated domain.

Examples

```
> (setq a (make-variable))
[16]
> (setq b (make-variable))
[17]
> (assert! (equalv b (funcallinv #'reverse #'reverse a)))
NIL
> (make-equal b '(one two three))
(ONE TWO THREE)
> a
(THREE TWO ONE)
>

;;; Demonstrate the propagation of enumerated domains for single
;;; argument functions
> (setq a (an-integer-betweenv 1 3))
[116 integer 1:3 enumerated-domain:(1 2 3)]
;;; b is constrained to be one more than a
> (funcallinv #'1+ #'1- a)
[120 integer 2:4 enumerated-domain:(2 3 4)]
>
```

3.9 Miscellaneous

`formatv` [Function]

Synopsis: (**`formatv`** *stream fstring* x_1 x_2 ... x_n)

Description: This function returns a variable, *z,* constrained to be the result of applying the LISP function **`format`** to the arguments *stream*, *fstring* and x_1 ... x_n. As with the Common LISP function, the formatting string *fstring* determines how the rest of the arguments x_1 ... x_n will be used and represented in the result. The stream argument determines what happens to the resulting output. If it is a stream value, then output is sent directly to that stream. If it is set to true (in Common LISP '**`t`**'), however, the output is sent to standard output, and if it is set to **`nil`**, the output is returned as a string. The output variable *z* does not become bound until all the arguments are also bound. In the case of stream output, this can have the effect of creating "format daemons", which wait until all their arguments are bound, and then comment upon it by writing to the stream (see example below). In addition, a mechanism is also provided for switching such a format daemon off. If the returned constraint variable becomes bound to **`nil`** *before* all the arguments to **`formatv`** become bound, then the output will not be generated.

Examples

```
;;; Example of a format daemon
;;; First, set up a couple of constraint variables
> (setq x (make-variable))
[32]
> (setq y (make-variable))
[33]
;;; Then set up the format daemon itself
> (setq z (formatv t "~%*** ~a ~a ***~%" x y))
[34]
;;; Then bind x and y
> (make-equal x 'hello)
HELLO
> (make-equal y 'world)
*** HELLO WORLD ***
WORLD
>

;;; Illustrate switching off format daemons
;;; First, set up constraint variables as before
> (setq x (make-variable))
[88]
> (setq y (make-variable))
[89]
;;; Set up the daemon
> (setq z (formatv t "*** ~a ~a ***~%" x y))
[90]
;;; Bind x
> (make-equal x 'hello)
```

```
HELLO
;;; Switch off the daemon
> (make-equal z nil)
NIL
;;; Now bind y. Note that the HELLO WORLD message is not printed.
> (make-equal y 'world)
WORLD
>

;;; Format daemons also work while searching for solutions:
> (setq a (a-member-ofv '(1 2 3)))
[69 integer 1:3 enumerated-domain:(1 2 3)]
> (formatv t "A is ~d~%" a)
[70]
> (all-values (solution a (static-ordering #'linear-force)))
A is 1
A is 2
A is 3
(1 2 3)
>
```

`*enumeration-limit*` [Variable]

Synopsis: **`*enumeration-limit*`**

Description: This variable is used to limit the scale of value propagation in SCREAMER+ when it might otherwise become too greedy for memory. The problem is that when considering enumerated domains of constraint variables, there are cases in which the domain size of the "output" constraint variable is larger than the "input" constraint variable. This means that as propagation progresses, domain sizes may increase uncontrollably if unchecked. We have therefore used the variable **`*enumeration-limit*`** to control the sizes of domains. If value propagation results in any constraint variable having a domain size of greater than, **`*enumeration-limit*`**, then the propagation is frozen. It may later proceed from the same point, however, if changes elsewhere result in the domain sizes of the output variables remaining within the stated limit. The limit can be changed by the user, but is currently set at 100.

Examples

```
> *enumeration-limit*
100
```

4 Example Programs

4.1 The Mastermind Game

This example was introduced as a non-trivial programming problem by Sterling and Shapiro (Sterling & Shapiro, 1986), and has also been the topic of other articles (e.g., Van Hentenryck, 1989; Merelo, 1996). The problem concerns the game of master-

mind, in which the object is for the code-breaker to crack the opponent's secret code in as few 'moves' as possible. In our version, the code consists of an ordering of 4 differently coloured pegs, in which the pegs are known to be coloured either red, green, blue, yellow, white, or black. Each 'move' consists of the code-breaker guessing the code and the opponent truthfully providing two pieces of information: firstly, how many of the guessed pegs are correctly coloured, and secondly, how many of the guessed pegs are both correctly coloured *and* positioned. (In practice these two pieces of information are provided in the reverse order, since if all the pegs are both correctly coloured and positioned, then the code has been cracked.) The problem described here is to write a program which cracks the code by each time making a guess which is consistent with its current knowledge of the code. That is, it should always try to guess the right answer, rather than, say, 'sacrificing' a guess which it knows *not* to be correct in an attempt to find out about the colours and/or positioning of the pegs more quickly (an optimal strategy is of this type).

A SCREAMER+ program to play mastermind in this way is given in Figure 2. The program uses the functions **`all-differentv`**, **`exactlyv`**, **`lengthv`**, and **`intersectionv`** from the SCREAMER+ library. Notice that if the problem becomes insoluble because the information provided by the human operator is inconsistent, then the program aborts immediately. Notice also that the program tests at each iteration to see if there is only one candidate solution left. If this is the case, the program reports the solution, and then halts.

An example session with the program is given below (the user's inputs have been underlined) :

```
My guess is (RED GREEN BLUE YELLOW)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
2
My guess is (RED BLUE WHITE BLACK)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
3
My guess is (RED YELLOW BLACK WHITE)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
3
My guess is (WHITE BLUE BLACK YELLOW)
How many are the right colour and correctly positioned ?
1
How many are the right colour ?
4
The code is (WHITE YELLOW BLUE BLACK)
```

```
(defun mastermind ()
  (if (catch 'fail
        (let* (
               (colours '(red green blue yellow white black))
               (peg1 (a-member-ofv colours))
               (peg2 (a-member-ofv colours))
               (peg3 (a-member-ofv colours))
               (peg4 (a-member-ofv colours))
               (sol (list peg1 peg2 peg3 peg4))
               guess total-correct colour-correct
               )
          (assert! (all-differentv peg1 peg2 peg3 peg4))

          (loop
            (setq guess (one-value (solution sol (static-ordering #'linear-force))))
            (if (ith-value 1 (solution sol (static-ordering #'linear-force)) nil)
                (format t "My guess is ~s~%" guess)
              (progn
                (format t "The code is ~s~%" guess)
                (return t)
                )
              )
            (format t "How many are the right colour and correctly positioned ? ~%")
            (setq total-correct (read))
            (if (= total-correct (length sol))
                (return t)
              (assert! (notv (equalv sol guess)))
              )
            (assert! (exactlyv total-correct #'equalv guess sol))
            (format t "How many are the right colour ? ~%")
            (setq colour-correct (read))
            (assert! (=v (lengthv (intersectionv guess sol)) colour-correct))
            )
          )
        )
      t
    (format t "Your replies must have been inconsistent.~%")
    )
  )
```

Figure 2: A SCREAMER+ Program to Play the Mastermind Game

4.2 The Car Sequencing Problem

An implementation of a program to solve the car sequencing problem (Dincbas, Simonis & Hentenryck, 1988) also demonstrates some of the facilities of SCREAMER+.

Not all cars on a car manufacturing assembly line require the same set of options. For example, while some cars may require the installation of air conditioning, others may not. The same may be true of sunroofs, stereo equipment, and numerous other options. This can present difficulties for the production line, since the team of technicians responsible for the installation of a particular option cannot cope with a glut of cars on the assembly line which all require that option. To counter this difficulty, the assembly line is designed with a predetermined capacity ratio for each option. This ratio compares the maximum allowable number of cars *with some option* with a corresponding *total throughput* of cars in the same time. So, for example, a capacity ratio of 1 out of 2 for sunroofs would indicate that no two consecutive cars should require a sunroof. Note that a ratio of 1 out of 2 (also written 1 / 2) is different from 2 out of 4, since the latter allows more flexibility in the ordering. The car sequencing problem, then, is to find an ordering for some given set of cars, with their different choices of options, such that the capacity constraints are satisfied.

An instance of the car sequencing problem is given in Table 3. In this problem there are ten cars and five different options. Since some of the cars required the same sets of options, they were subdivided into *car-types*[11]. This step reduces the size of the search space considerably, since we now assign a *car-type* instead of a car to each of the 10 positions in the final ordering. Thus, the ten car problem has a search space size of 10^6 instead of 10^{10}, which would have arisen if the cars had not been clustered into types. The table also lists the capacity ratios for each of the options.

	type-1	**type-2**	**type-3**	**type-4**	**type-5**	**type-6**	**Capacity Ratio**
option-1	yes	no	no	no	yes	yes	1 / 2
option-2	no	no	yes	yes	no	yes	2 / 3
option-3	yes	no	no	no	yes	no	1 / 3
option-4	yes	yes	no	yes	no	no	2 / 5
option-5	no	no	yes	no	no	no	1 / 5
number of cars	1	1	2	2	2	2	-

Table 3: An Instance of the Car Sequencing Problem with 10 Cars

To solve the problem we use four different sorts of constraints (Dincbas, Simonis, and Van Hentenryck, 1988). (The sections of code corresponding to the assertions of each of these four types of constraint are labelled in the implementation in Figure 3.) Firstly, we constrain each of the ten elements in the solution to be one of the given car types (1). Secondly, we constrain the numbers of each car type appearing in the solution to match those given in the bottom row of Table 3 (2). Thirdly, we constrain every possible sublist of the appropriate length to remain within the limits of the capacity ratio requirement (3). Finally, we use some additional constraints, called *surrogate constraints*, which are semantically redundant but nevertheless help to discover fruitless branches of the search tree more quickly (4). Let us illustrate the idea behind this last set of constraints by example. There are three cars requiring option 3 (one of type 1, and two of type 5), which must all fit into the final ordering of ten cars. Since the capacity constraint for option 3 is 1 / 3, there cannot be more than 1 car with option 3 in the last 3 cars of the ordering. Therefore, we would also expect at least two cars with option 3 in the first seven of the ordering. We use the same reasoning to derive the constraint that the first four cars must contain at least one car with option 3. Similar constraints are generated for each of the options.

11. These were called *classes* in (Dincbas, Simonis & Hentenryck, 1988), but we have introduced the term *car-type* so that we can more easily distinguish between “car classes” and the CLOS classes of the implementation.

Notice that we have taken care to avoid any duplication of the problem instance information in the implementation. For example, the program itself computes the number of cars with a particular option, given the option choices for each of the car types and the number of cars of each type.

Our implementation makes use of the SCREAMER+ functions **`make-listv`**, **`constraint-fn`**, **`exactlyv`**, **`at-mostv`**, and **`at-leastv`** resulting in an efficient, concise program solution. It finds all six solutions to the ten car sequencing problem in a CPU time of approximately 0.7 seconds on a quad processor Sun UltraSPARC Enterprise 450 running Franz Allegro LISP 4.3. This compares with approximately 0.25 seconds using CHIP 5 to reproduce the program described by Dincbas, Simonis and Hentenryck on the same machine. The reason for this time discrepancy is threefold. Firstly, and most importantly, CHIP's constraint predicates are directly implemented in C, rather than the host language, Prolog. This results in a considerable speed advantage, but also has the disadvantage that, unlike SCREAMER+, the constraint library is no longer portable across different host language implementations.

Secondly, our implementation is truly symbolic, in the sense that the requirements for options are left as boolean values (**`t`** or **`nil`**) in the program. Dincbas, Simonis and Hentenryck converted all their Boolean values to 1 or 0, so that when checking their capacity constraints, they could sum up these numbers and use a linear inequality on the sum. Checking a truly symbolic constraint is in general more time consuming than checking a linear inequality, since we cannot make assumptions about the types of variables involved in the constraint. (For example, our constraint primitives use **`equal`** rather than, say, **`=`** or **`eq`**.) On the other hand, our approach is more flexible since we can easily change the program to cope with option variables which can take one of *many* values (instead of just the two possible values of a boolean variable), a modification which cannot be carried over easily to the usage of linear inequalities.

Thirdly, our constraint predicates **`at-leastv`** and **`at-mostv`** are more flexible than those of CHIP. The CHIP predicate **`atmost(N, LIST, VAL)`** is an *assertion* that at most **`N`** elements of the *list of natural numbers* **`LIST`** have the value **`VAL`**. Recall that the SCREAMER+ predicate, **`(at-mostv`** $n\ f\ x_1\ x_2\ \ldots\ x_n$**`)`** takes a *predicate function* as an argument, together with the *multiple, symbolic list* arguments to which it should be applied, and returns a *boolean variable* indicating whether the predicate function is true of its arguments at most *n* times. Thus, our implementation is a higher-order function, not confined to constraining only a single list of numbers, which enables the programmer to assert the falsehood as well as the truth of the constraint by an appropriate binding of the output variable (e.g. **`(assert! (notv (at-mostv ...)))`**). The flexibility of our predicates also pays a price in terms of speed for this particular problem, but will gain in other situations where less flexible predicates prove to be insufficient.

Note also that our implementation has used object-orientation to model more closely the structure of the problem. The option information for each car type is held as part of the corresponding object. This can become a great advantage when the number of car types we need to model increases significantly. In our implementation, we could then extend the CLOS class structure of car-types to encompass an inheritance hierarchy

```
(defclass car-type ()
  ((option-1 :initarg :o1)
   (option-2 :initarg :o2)
   (option-3 :initarg :o3)
   (option-4 :initarg :o4)
   (option-5 :initarg :o5)))

(defun sublists (n x)
  (do ((acc nil)
       (going x (cdr going)))
      ((< (length going) n) acc)
    (push (subseq going 0 n) acc)))

(defun surrogate (sequence cars-with-option which-option)
  (declare (special *capacities*))
  (do* ((capacity (cadr (assoc which-option *capacities*)))
        (capacity-size (caddr (assoc which-option *capacities*)))
        (countdown (length sequence) (- countdown capacity-size))
        (options-left cars-with-option (- options-left capacity))
        (fn (constraint-fn #'(lambda(x) (slot-value x which-option)))))
      ((< countdown 0) t)
    (assert! (at-leastv options-left fn (subseq sequence 0 countdown)))))

(defun assert-capacities (seq which-option)
  (declare (special *capacities*))
  (do* ((option-capacity (cdr (assoc which-option *capacities*)))
        (maxnum-in-sub (first option-capacity))
        (from-sequence (second option-capacity))
        (sub (sublists from-sequence seq) (cdr sub))
        (s (car sub) (car sub)))
      ((endp sub) t)
    (assert!
     (at-mostv maxnum-in-sub
               (constraint-fn #'(lambda(x) (slot-value x which-option))) s))))

(defun count-numbers (car-dist which-option)
  (do* ((count 0)
        (diminish car-dist (cdr diminish))
        (current (car diminish) (car diminish)))
      ((endp diminish) count)
    (when (slot-value (car current) which-option)
      (setq count (+ count (cdr current))))))

(defun solve ()
  (let (*capacities* car-types car-dist seq type1 type2 type3 type4 type5 type6)
    (declare (special *capacities*))
    (setq *capacities* '((option-1 1 2)
                         (option-2 2 3)
                         (option-3 1 3)
                         (option-4 2 5)
                         (option-5 1 5)))
    (setq
     type1 (make-instance 'car-type :o1 t   :o2 nil :o3 t   :o4 t   :o5 nil)
     type2 (make-instance 'car-type :o1 nil :o2 nil :o3 nil :o4 t   :o5 nil)
     type3 (make-instance 'car-type :o1 nil :o2 t   :o3 nil :o4 nil :o5 t)
     type4 (make-instance 'car-type :o1 nil :o2 t   :o3 nil :o4 t   :o5 nil)
     type5 (make-instance 'car-type :o1 t   :o2 nil :o3 t   :o4 nil :o5 nil)
     type6 (make-instance 'car-type :o1 t   :o2 t   :o3 nil :o4 nil :o5 nil))

    (setq car-types (list type1 type2 type3 type4 type5 type6))
    (setq car-dist (pairlis car-types '(1 1 2 2 2 2)))

    (setq seq (make-listv 10))            ; Sequence 10 cars
1   (dolist (s seq) (assert! (memberv s car-types)))
    (dolist (ty car-dist)              ; Assert distribution of car types
      (assert!
2      (exactlyv
        (cdr ty)
        (eval `(constraint-fn (function (lambda(x) (equal x ,(car ty))))))
       seq)))
    (dolist (o *capacities*)
3     (assert-capacities seq (car o))
4     (surrogate seq (count-numbers car-dist (car o)) (car o)))
    (all-values (solution seq (static-ordering #'linear-force)))))
```

Figure 3: A SCREAMER+ Program for Solving an Instance of the Car Sequencing Problem with 10 Cars

with appropriate default values. This would provide a more natural summary of the information than extensive sets of lists containing car options, particularly if the list elements are to be referenced positionally.

5 Summary

In this paper, we first described SCREAMER, a package that embeds nondeterminism and explicit constraint handling into Common LISP. We commented on our experiences of applying SCREAMER, and reported some of the problems we encountered. We also disclosed the deficiences of SCREAMER in handling symbolic constraints, and demonstrated how these deficiencies can be addressed by an extension to the SCREAMER library. We call the extended version of SCREAMER "SCREAMER+".

SCREAMER+ enables some forms of reasoning and problem solving which would not have been possible with SCREAMER alone. For example, the SCREAMER+ expressions (**consv** *x y*) and (**carv** *x*) are superior to the corresponding SCREAMER versions (**funcallv #'cons** *x y*) and (**funcallv #'car** *x*) because they can propagate their results earlier:

```
;;; *** SCREAMER Version ***
;;; Construct a list in which the tail is not yet determined
> (setq a (funcallv #'cons 'head (a-member-ofv '((one) (two)))))
[100]
;;; the head of the list is currently unknown
> (funcallv #'car a)
[101]

;;; *** SCREAMER+ Version ***
;;; Construct a list using the same arguments as above
> (setq a (consv 'head (a-member-ofv '((one) (two)))))
[102 nonnumber enumerated-domain:((HEAD ONE) (HEAD TWO))]
;;; the head of the list is already known
> (carv a)
HEAD
```

The higher order functions defined by SCREAMER+ complement the list-oriented functions, enabling the programmer to use ordinary LISP functions to express complex constraints over whole lists, parts of lists, or individual elements of lists. This ability, which can also be merged with the advantages of encapsulation and inheritance offered by the Common LISP object system, provides a powerful, general basis for constraint-based symbolic reasoning.

References

DINCBAS, M., SIMONIS, H., VAN HENTENRYCK, P., (1988), "Solving the Car-Sequencing Problem in Constraint Logic Programming", in proceedings of ECAI-88.

MERELO, J. J., (1996), "Genetic Mastermind, a case of dynamic constraint optimiza-

tion", Technical Report G-96-1, Department of Electronics and Computer Technology, University of Granada, Spain.

SISKIND, J. M., (1991), "Screaming Yellow Zonkers", Draft Technical Report of 29th September 1991, supplied with the SCREAMER code distribution 3.20 at `http://linc.cis.upenn.edu/~screamer-tools/home.html`.

SISKIND, J. M., MCALLESTER, D. A., (1993), "Nondeterministic LISP as a Substrate for Constraint Logic Programming", in proceedings of AAAI-93.

SISKIND, J. M., MCALLESTER, D. A.,(1994), "SCREAMER: A Portable Efficient Implementation of Nondeterministic Common LISP'", Technical Report IRCS-93-03, Uni. of Pennsylvania Inst. for Research in Cognitive Science.

STEELE, G. L. Jr., (1990), "Common LISP the Language", Second Edition, Digital Press, Woburn, MA, USA.

STERLING, L., SHAPIRO, E., (1986), "The Art of Prolog: Advanced Programming Techniques", MIT Press, Cambridge, MA, USA.

VAN HENTENRYCK, P., (1989), "Constraint Satisfaction in Logic Programming", MIT Press, Cambridge, MA, USA.

Index of SCREAMER+ Functions