CONSTRAINTS: A UNIFORM MODEL FOR DATA AND CONTROL

L. Peter Deutsch
Xerox PARC / MIT Artificial Intelligence Laboratory

Most programming systems reflect a model of computation which sharply distinguishes between "passive" data objects and "active" program objects (procedures). Furthermore, procedures describe not only a set of computations but the precise flow of control between them. In contrast, a group at MIT is investigating a new paradigm called *constraints* in which a single kind of object models both data and procedures, and in which the description of procedures minimizes commitment to the order in which computational steps will be executed.

A (primitive) constraint is an object with some *parts*, which correspond to fields of a data structure or input/output parameters of a procedure, and a *body* which describes how to compute the values of some parts from other parts. The body consists of *rules* written in an implementation language (Lisp in the MIT systems). Constraints with no body behave like ordinary data structures; constraints with substantial bodies behave more like procedures. However, unlike procedures, which always compute the same set of outputs from the same set of inputs, a constraint may compute in different directions depending on the available data. A good example of this is the addition constraint which is defined as follows:

```
(constraint plus
    (parts (sum addend1 addend2))
    (body
        (rules
            (<- sum (+ addend1 addend2))
            (<- addend1 (- sum addend2))
            (<- addend2 (- sum addend1))
        ))
    )
```


This constraint computes sum from (addend1 + addend2), or addend1 from (sum - addend2), or addend2 from (sum - addend1), depending on which of sum, addend1, and addend2 have known values. The multiplication constraint is similar but more interesting, since it can determine that the product is zero if either factor is zero, regardless of whether the other factor is known:

```
(constraint times
    (parts (product factor1 factor2))
    (body
        (rules
            (if (= factor1 0) (<- product 0))
            (if (= factor2 0) (<- product 0))
            (<- product (* factor1 factor2))
            (if (~= product 0)
                (if (= factor2 0)
                    (fail)
                    (<- factor1 (/ product factor2))))
            (if (~= product 0)
                (if (= factor1 0)
                    (fail)
                    (<- factor2 (/ product factor1))))
        ))
    )
```

If a rule is run with insufficient information to compute a result, it simply *dismisses* with no effect. If, however, a rule finds that the present state of the constraint is inconsistent, the rule *fails*. For example, the last two rules in the product constraint fail if one factor is zero but the product is non-zero. Failure has many different causes; understanding how to handle it is a subject of current research.

Three familiar abstraction mechanisms allow building up more complex constraints from simpler ones: recursive embedding, parametrization, and type declaration. We obtain recursive embedding by allowing the body of a constraint to be a constraint network, rather than primitive rules, as in the following 3-addend sum constraint. To express this we must introduce the notation (<< x p) to mean the part named p of the constraint x.

```
(constraint 3plus
    (parts (sum addend1 addend2 addend3)
        internal ((plus-1 plus) (plus-2 plus)))
    (body
        (connections
            (= = (<< plus-1 addend1) addend1)
            (= = (<< plus-1 addend2) addend2)
            (= = (<< plus-2 addend1) (<< plus-1 sum))
            (= = (<< plus-2 addend2) addend3)
            (= = (<< plus-2 sum) sum)
        ))
    )
```

3plus, like plus and times, is now a full-fledged constraint that can be instantiated just as plus was instantiated twice within 3plus, with the parts of the instances connected to parts of the constraint within which it appears. Note that the property of multi-directional computation is inherited by more complex constraint networks in a natural way. For example, if a given instance of 3plus is given values for sum, addend1, and addend2, its plus-1 will run "forward" to compute (<< plus-1 sum), and its plus-2 will run "backward" to compute addend3. Note also that instantiation essentially means copying: there is no distinction between "classes" or "prototypes" and other kinds of objects.

Since every constraint is potentially a prototype, and since all data in our system are in the form of constraints, every part has a *type* associated with it, which is simply the name of the constraint which is the prototype for the value of the part. (In the examples so far, all quantities were scalars, so the type specification was elided.) For example, consider the following constraints:

```
(constraint cartesian-complex
    (parts (real imag))
    )

(constraint square
    (parts (root power))
    (body
        (rules
            (<- power (* root root))

            (if (< power 0) (fail) (<- root (sqrt power)))
        ))
    )

(constraint complex-magnitude
    (parts (magnitude (cart cartesian-complex))
        internal ((plus-1 plus) (sq-1 square)
            (sq-2 square) (sq-3 square)))
    (body
        (connections
            (= = (<< sq-1 root) (<< cart real))
            (= = (<< sq-1 power) (<< plus-1 addend1))
            (= = (<< sq-2 root) (<< cart imag))
            (= = (<< sq-2 power) (<< plus-1 addend2))
            (= = (<< plus-1 sum) (<< sq-3 power))
            (= = (<< sq-3 root) magnitude)
        ))
    )
```

The notation "(cart cartesian-complex)" indicates that the value of the v part of the magnitude constraint is not a scalar datum, but a cartesian-complex constraint.

Like some other systems that include the notion of type or class, ours includes the notion of an *aspect* or *superclass* of an object (constraint). Thus, for example, one could define a complex number as including a cartesian-complex and a polar-complex aspect, with constraints that related the two:

```
(constraint polar-complex
    (parts (r theta))
    )

(constraint complex
    (parts ((cart cartesian-complex)
        (polar polar-complex))
        aspects (cart polar)
        internal ((mag-1 complex-magnitude)
            (arctan-1 quotient-arctan)))
    (body
        (connections
            (= = (<< mag-1 cart) cart)
            (= = (<< mag-1 magnitude) (<< polar r))
            (= = (<< arctan-1 numerator)
                (<< cart imag))

            (= = (<< arctan-1 denominator)
                (<< cart real))
            (= = (<< arctan-1 angle) (<< polar theta))
        ))
    )
```

The meaning of the aspects declaration is that it is possible to equate any collection of parts with types complex, cartesian-complex, and polar-complex: our system automatically recognizes complex as being the appropriate combined constraint, and the internal constraints in complex enforce consistency between the two views. (This example is due to Gerry Sussman.)

We are investigating both the expressive and pragmatic problems of constraints as a programming paradigm. Here are some of the expressiveness questions we are currently addressing:

- Constraint networks are a data abstraction mechanism, since the user of a constraint cannot tell how a given part is computed or stored. Is there a good way to encapsulate operations, as well as data, within a constraint? This would make constraints similar in power to the class/instance model of data.

- How does one best express constraints on aggregate objects like sets, or recursive objects like lists?

- What is an appropriate model for the notion of default values, or preference of one source of information over another?

- Given several parts that look at different aspects of an object, what does it mean to synthesize an object that has all the desired aspects? No such object (constraint) may have been foreseen. This is similar to the "multiple superclass" problem in languages descended from Simula.

- What applications are particularly suitable or unsuitable for expression as constraints? In particular, are constraints a good framework for expressing queries and/or integrity conditions for data bases?

- Constraint networks can be viewed as a specialized network of Actors: are there natural ways to combine the constraints framework and the more general Actor notions of messages and changes of state? In particular, can constraints model real-world systems in which the notions of time, event, and state are important?

Here are some of the pragmatic questions of current interest to us:

- What is the best way to handle almost-functional constraints, like square root?

- There exist constraint networks in which purely local propagation of values is inadequate to find actual values that satisfy the constraints. What techniques are appropriate for searching for solutions in such cases?

- How does one determine a good order for executing rules, which wastes little time running rules which dismiss but discovers failure as early as possible?

- To what extent can one compile constraint networks into efficient code for existing hardware?

- Are there new hardware architectures (most likely implemented in VLSI) that are better suited to executing programs represented as constraints?

- Given that failure sometimes results from local problems which can be resolved at a higher level, sometimes from bugs, and sometimes from illegal user input, what kind of information must be retained to decide what action is appropriate when failure occurs?

- In a constraint network, whether values are saved or recomputed is purely a pragmatic question. Saving values often allows a change in some (externally supplied) input value to produce the appropriate change in an (externally viewed) output value with much less work than recomputing all the intermediate values. How should the system make, or the user state, decisions about what to store and what to recompute?

ACKNOWLEDGEMENTS

The primary originators of the work described here are Gerald Sussman, Richard Stallman, Guy Steele, and others at the MIT Artificial Intelligence Laboratory.

BIBLIOGRAPHY

Hewitt, Carl
Actors (MIT Artificial Intelligence Lab memo)

Sussman, Gerald, and Stallman, Richard
Constraints (MIT Artificial Intelligence Lab memo)

Cannon, Howard
Tasteful Flavors (MIT Artificial Intelligence Lab memo)

Borning, Alan
ThingLab, a constraint-oriented simulation laboratory. Ph.D. thesis, Stanford University, 1980

Goldberg, Adele, et al.
Smalltalk: Dreams and Schemes. (book in preparation at Xerox Palo Alto Research Center.)