Constraints

A constraint is simply a relation that we would like to have hold.

Applications in interactive graphics:

- page layout
- drawing editors
- consistency between application and view, and among multiple views

Other application areas include computational linguistics, configuration, decision support, design, diagnosis, graphics, hardware verification, molecular biology, planning, program analysis, qualitative reasoning, real-time systems, resource allocation, robotics, scheduling, software engineering, temporal reasoning, type inference, vision, and visualization.

Constraints in Functional and Logic Programming Languages

Mathematically, a function is just a particular kind of relation.

So we can view functional programming languages as kinds of constraint languages, where the constraints are always solved for a distinguished result variable:

$y = 5$
$x = 10 + y$

In constraint logic programming languages such as CLP(R) we can write constraints directly:

```
X - Y = 10, Y = 5
```

or even:

```
2*X + Y = 7, X + Y = 5
```

Constraints and State

What about state?

Suppose we have a constraint:

`a+b = c`

and currently `a=3, b=5, c=8`.

What happens if we execute `c := 10`?

Constraints and State (2)

One formalism for handling this: constraint hierarchies.

```
required    a+b = c
required    c = 10    <- from assignment statement
weak        a = 3
weak        b = 5
weak        c = 8
```

Constraints and State (3)

The previous situation is so common that we will define two special kinds of constraints, *edit* and *stay* constraints.

```
required    a+b = c
required    c edit
weak        a stay
weak        b stay
weak        c stay
```

`a stay` means that `a` should retain its old value if possible.

`a edit` means that we will be adding a constraint `a=k` for some constant `k` — but we want to plan how to satisfy the constraints for many different values for `k`.

Constraint Hierarchies

Allow both required constraints and preferences (hard and soft constraints)

Important uses:

- stability when moving a figure (use weak constraints that things stay where they are)

- handling user inputs that are outside the range of permitted values

- balancing conflicting desires (e.g. in layout)

Constraint Hierarchy Solutions

There can be an arbitrary number of constraint strengths.

In finding a solution, we *must* satisfy the required constraints. We try to satisfy the stronger non-required constraints in preference to the weaker ones.

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{strong} & x = 8 \\ \text{weak} & x = 0 \\ \text{weak} & y = 0 \end{array}$$

Solution is $\{x \mapsto 8, y \mapsto 2\}$

Constraint Hierarchy Solutions (2)

The error in satisfying a constraint is a non-negative real number which is 0 iff the constraint is satisfied. In the work described here, the error is larger the less satisfied the constraint.

Examples:
The error for $x = y$ is $|x - y|$.
The error for $x \leq y$ is 0 if $x \leq y$ and otherwise $x - y$.

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{strong} & x \geq 2 \\ \text{strong} & x \leq 5 \\ \text{weak} & y = 0 \end{array}$$

The required and the strong constraints allow y to range between 5 and 8.

Solution is $\{x \mapsto 5, y \mapsto 5\}$

Constraint Hierarchy Solutions (3)

What if two constraints of the same strength conflict?

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{strong} & x = 0 \\ \text{strong} & y = 0 \end{array}$$

There are various ways of trading off constraints in this situation. The version used here allows the errors for the strong constraints to be traded off however we wish, as long as there isn't another solution that satisfies all the constraints at least as well and one constraint strictly better.

Solutions include $\{x \mapsto 0, y \mapsto 10\}$
$\{x \mapsto 10, y \mapsto 0\}$, $\{x \mapsto 5, y \mapsto 5\}$
and generally all x and y between 0 and 10 such that $x + y = 10$.

However, $\{x \mapsto 11, y \mapsto -1\}$ is *not* a solution.

Constraint Hierarchy Solutions (4)

For a constraint hierarchy H, let H_0 be the required constraints, H_1 be the strongest non-required constraints, H_2 be the next strongest, etc. Let $e(c\theta)$ be the error in satisfying constraint c for valuation θ.

The set S of solutions to a constraint hierarchy is defined as follows:

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0 \; e(c\theta) = 0\} \\ S &= \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \; \neg \mathit{better}(\sigma, \theta, H)\} \end{aligned}$$

$$\begin{aligned} &\mathit{better}(\theta, \sigma, H) \equiv \\ &\quad \exists k > 0 \;\; \text{such that} \\ &\qquad \forall i \in 1 \ldots k-1 \; \forall p \in H_i \;\; e(p\theta) = e(p\sigma) \\ &\qquad \wedge \;\; \exists q \in H_k \;\; e(q\theta) < e(q\sigma) \\ &\qquad \wedge \;\; \forall r \in H_k \;\; e(r\theta) \le e(r\sigma) \end{aligned}$$

(There are a number of variants of *better*; this one is *locally-error-better*.)

Constraint Solvers

- one-way solvers (example: spreadsheets)

- multi-way local propagation solvers (example: DeltaBlue)

- solvers for simultaneous linear equations and inequalities (example: Cassowary)

- solvers using iterative numerical techniques (example: Bramble)

- solvers for finite domain constraints (take the AI class . . .)

Constraint Imperative Programming Languages

- Some parts of an interactive user interface are most easily described using constraints.

  Example: the relation between a scrollbar and the contents of a window.

- Other parts are most easily described using imperative constructs.

  Example: modes in a dialog box interaction.

Kaleidoscope'93

An object-oriented constraint imperative programming language.

- integrates constraints and imperative constructs

- constraints treated in an object-oriented fashion using *constraint constructors* — allows user-defined constraints over user-defined classes

- preserves encapsulation

- Kaleidoscope compiler generates K-codes: instructions for a virtual machine designed for constraint imperative languages

Sample Imperative Code

```
while mouse.button = down do
    old ← mercury.top;
    mercury.top ← mouse.location.y;
    temperature ← mercury.height / scale;
    if old < mercury.top then
        delta_grey( old, mercury.top );
        display_number( temperature );
    elseif old > mercury.top then
        delta_white( mercury.top, old );
        display_number( temperature );
    end if;
end while;
```

Sample Kaleidoscope Code

always: temperature = mercury.height / scale;
always: white_rectangle(thermometer);
always: grey_rectangle(mercury);
always: display_number(temperature);
while mouse.button = down **assert**
    mercury.top = mouse.location.y;
end while;

Constraint Constructors

Support user-defined constraints:

```
constructor +(a, b: Point) = (c: Point);
    a.x + b.x = c.x;
    a.y + b.y = c.y;
end constructor +;

constructor +(a, b: Vector) = (c: Vector);
    local i: Integer;
    for i := 1 to a.size do
        always: c[i] = a[i] + b[i];
    end for;
end constructor +;
```

Thus which constructor for the + constraint is chosen in $a + b$ depends on the types of a and b.

Multi-Methods

Suppose we have the following classes:

Object
    Numeric
        Point
        Vector

and the following code:

```
a, b, c : Numeric
required: c = a+b;
c := 3@4; /* creates a point with x=3, y=4 */
b := 1@1;
```

Which + constructor is invoked?
Multi-methods needed!

Durations

We can annotate constraints with *durations* in addition to strengths:

- always: a+b=c (always is the default)
- once: a+b=c
- while . . . assert . . .

Constraints and Assigment

For use in an imperative programming language, this also gives us a way to integrate assignment (and other kinds of state change) with constraints.

What does `x := x+5` mean?

Wrong: $x = x + 5$

A correct representation: $x_t = x_{t-1}? + 5$

(? means read-only)

Refinement Semantics vs. Perturbation Semantics

An alternative semantics: perturbation semantics.

Rather than describing variables as holding streams of values, we can use a conventional representation — variable refers to an object with state.

```
x := x+5;
```

modelled as

```
once:  temp = x?+5;
once:  x = temp?;
```

All changes in the system are modelled as adding and removing constraints. The only kind of temporal constraints are stay constraints.

Object Identity

Object identity is a fundamental attribute of object-oriented programming (at least OOP with state).

We want to integrate object identity with constraints. (See Gus Lopez, Bjorn Freeman-Benson, and Alan Borning, "Constraints and Object Identity," *Proceedings of the 1994 European Conference on Object-Oriented Programming*, July 1994, for more details.)

So we want identity constraints in addition to equality constraints:

`x == y` vs. `x = y`

In Kaleidoscope "stay" should be an identity constraint — not an equality constraint — so that identity relationships are preserved as time advances.