Visualization: What for?

- Important area with many applications and branches
- Besides graphics:
  - Scientific data visualization,
  - Control flow,
  - Data flow,
  - Data structures,
  - Making debugging easier,
  - …
- We will focus on visualizing CLP program executions
- For a variety of aims
- And exploring several technical and conceptual possibilities
General Motivations

- **Intuitive** view of the execution
  1. Users: better understanding of how their programs behave
  2. Partial view of the underpinnings of the underlying system
  3. Implementors: deeper understanding about their own implementations
  4. For some languages (e.g., the CLP breed), a clearer view of the language mechanism

- (1), (2) and (3) address:
  - Program and system debugging (*correctness debugging*)
  - Tuning and optimizing (*performance debugging*)

- (4) is relevant to teaching and education

- Intuitive representations for (different aspects of) CLP + suitable *abstractions* are sought

Visualization in CLP

- **Challenges:**
  - Mapping program → execution not always straightforward
  - **Dynamic** behavior (in contrast with, e.g., flowcharts)
  - Large executions
  - Should be tailored to each particular case

- Many proposals (see section “Pointers”)


- Most proposals address:
  - “Standard” (e.g., Herbrand) data and bindings [1]
  - “Standard” (e.g., depth-first) control rules [1]
  - Constrained variables [2]
  - Search trees for CLP (performance!) [partially 1]
  - Alternative execution schemes (e.g., parallelism or concurrency) [3]
Viewing Search Trees

- (C)LP execution model usually represented as an And/Or tree
- Different search orders possible
- Actual CLP systems have fixed execution strategies (Prolog adopts depth first)
- Visualizing this tree is helpful for understanding the “big picture”:
  - Show the (programmed) search actually performed
  - Detect which parts of the program dominate the execution
  - Detect where search fails

Also useful to visualize the search inside the enumeration procedures in some constraint domains

Viewing Search Trees (Cont.)

- APT: experimental prototype
- Along the lines of the TPM (*Transparent Prolog Machine*)
- Depict (modified) And/Or tree of (C)LP program executions + variable bindings
Inspecting Nodes

- Nodes can be blown up for greater detail ➞ show data
- Calls and entry / exit substitutions are explicitly shown:

![Node View iso]

- Origin of instantiation marked in the tree (Color Prolog [A. Kusalik]: use color connections for variables!)
- Graphical aid for possible detection of failure origin
- Different execution strategies, navigation

Abstracting Trees

- Large executions: too many nodes
- Subtrees can be selectively collapsed / expanded (e.g., Oz viewer)

![Collapsed nodes tagged according to amount of work]

- Can be restricted to, e.g., failure / success paths
- Smart self-scaling: compress where more objects are present (as in, e.g., the Vista tool), Fish-eye lens, …
- Tagging schemes add more information to any representation
Implementation Details

- Based on:
  - Compile-time program rewriting
  - Plus a (complex) metainterpreter
- Full execution tree completely stored
- Plus all bindings and their origin
- Completely user transparent

Visualization of Constraints

- CLP in practice: Prolog + constraints
- Search tree also in CLP (sometimes hidden)
- Specialized search tree depictions for this case
  - E.g., the CHIP tools
- Data more difficult to represent
  - Not only values, but also relationships (constraints / equations)
  - Other CLP languages / libraries can use similar visualizations
- We will illustrate this with an apparently easy case: Finite Domain constraints (FD)
Finite Domains

- Used in industry: scheduling, planning, logistics...

- **Finite Domain** variables range over *finite* sets of non-negative integers: $x \in \{0, 7, 10..15, 20\}$ (oversimplifying!)

- Operations on FD variables are elementwise extensions of the arithmetic ones: \{3..7, 9\} + \{1,2,5\} = \{4..12, 14\}

- **Equations** relating FD variables are dynamically set up

- Simplified on the fly, but definite solutions need (costly) enumeration $\Rightarrow$ search

- Always finite

- Heuristics can greatly improve efficiency

- Performance influenced by how equations are written

- Understanding the internal behavior of the solver is not easy!

FD Variables: example

- DONALD + GERALD = ROBERT

  A digit per character, all different, numbers do not start by zero

  ```prolog
dgr(X):-
    [D,O,N,A,L,G,E,R,B,T] = X, % All of our variables
domain(X, 0, 9), % A digit per character
D #> 0, G #> 0, % Numbers do not start by zero
all_different(X), % No repetitions
100000*D + 100000*O + 10000*N + 100*A + 10*L + D +
100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
100000*R + 100000*O + 10000*B + 100*E + 10*R + T,
labeling(X).
```

- Constraints are set up at the beginning

- Domains are progressively reduced, constraints are simplified

- Final step: enumeration (can use heuristics)
Displaying FD Variables

- Basic representation: highlight the actual values inside the whole domain (**Grace** representation for Eclipse):

  ![Variable X](image)

  (Variable X can have values 1, 2, 4, and 5)

- “Shape” of the actual values inside the whole domain easy to see

- Can be animated or stacked (trade space for time): time running from top to bottom

- History of variables statically shown: comparison / view of evolution

Displaying FD variables (Cont.)

- Similarities among variables can be pointed out
- Backtracking is easily spotted
- Variables with weak dependency can also be detected

- **AllDifferent** constraint depicted
- Weakly constrained variables also visible!
Displaying Constraints

- Interaction among variables difficult to perceive from textual representation of constraints
- But should be taken into consideration:
  - (Wrong) independence among variables
  - Incorrect relationship among variables
  - Additional constraints may improve performance
- We want to know (intuitively) how variables affect each other
- Constraint depiction can be based on the representation of variables
  - Allow variable updating
  - Pass updated variable to the solver
  - Show how other variables are affected
- Can be used in any constraint domain with a graphical representation for variables!

Approximating Constraints

An example with FD variables \(x, y, z\):

The user updates the domain of \(y\):

- \(x\) and \(z\) are (visually) updated as well
- More “dynamic” than stacking variable representations
- Lacking dependencies easy to spot!
Approximating Constraints

- This process can be partially automated
- Select a variable, make a value active at a time, see how other variables react to this. (can be done with a slider, for fast feedback)
- The example on the right updates variable $Y$:

Approximating Constraints

- Generalization:
  - 2-D plot of variable enumeration
  - The user can update variables not plotted
  - Grid changes dynamically

\[ X, Y, Z \in \{1..6\} \land X \neq 6 \land X \neq 3 \land Z = 2X - Y \]

- Allow the user to post arbitrary constraints
- Specific visualizations for “global constraints”
Abstracting Values

- Variables in real examples can have large domains
- Previously proposed representations convey too much information
- Compression / zooming / navigation is a possibility but (maybe interesting) details can be blindly lost
- Highlight interesting properties in the representation:
  - Select only certain variables
  - Filter domains (filters can be expressed as a constraint!)
  - Show only domain sizes
- This last one has interesting properties:
  - Captures how near a variable is from having a definite solution
  - Captures backtracking
  - And the whole execution can be rendered as a 3D model

3-D Visualization of the Store

- Evolution of program data rendered as 3-D (discrete) surface
- Showing simultaneously:
  - \textbf{X}: The CLP(FD) variables
  - \textbf{Y}: An abstraction of the variable (the size of its domain)
  - \textbf{Z}: Time

- Different orderings yield altogether different search profiles
Abstracting Values: A 3-D Model

- DONALD + GERALD = ROBERT (rotation helps a lot)

- Same program, different heuristic in the selection of variables
Abstracting Constraints (Cont.)

- Plots of filtered domains
- Depict store: program variables represented as graph nodes
- (Hyper-)Links represent constraints
- Depiction can be animated!

\[ \Rightarrow \text{Store evolution} \]

- Graph density \(\rightarrow\) connectivity
- Graph links also propagation paths: change program to favor locality
- Distributed constraint solving: helps in deciding right placement
- Other features can be embedded in the graph: weights, tags, ...

Architecture

- Simple visualization calls inserted in user (or system) code
- Visualization routines keep track of active views, stepping, and access to constraint store
- Toplevel window: global control, manages other windows
- Each window gives feedback to the toplevel one
Visualization of Parallel Execution

- Several types of parallel execution in (C)LP:
  - Unification parallelism, Or-parallelism, stream parallelism, (determinate) dependent And-parallelism, independent And-parallelism, restricted And-parallelism

- Target:
  - Parallel search tree (a sort of abstraction of the sequential one)
  - Actual execution time
  - Potential and actual parallelism, speculative computation
  - Scheduling

- Will focus on visualization for RAP (visualization for other schemes also available in implemented tool)
- Many other tools of course available

The Basic Scheme for Restricted IAP

- Structure: fork/join
- No data dependencies among parallel goals
  
  \[
  a : - b \land c, g. \\
  c : - d \land e \land f. \\
  \]

- Tool has additional features:
  - Color coding to pinpoint processors/agents
  - Zooming, accurate time measurement
  - Communication with other tools
Visualization of And-parallelism

\[ p :- q \land r \land s. \]

Execution on 1 processor

Visualization of And-parallelism

\[ p :- q \land r \land s. \]

Execution on 3 processors
Visualization of And-parallelism

Multiplication of 2 4×4 matrices, 1 processor

Multiplication of 2 4×4 matrices, 4 processors

Visualization of And-parallelism

QuickSort, 1 processor

QuickSort, 4 processors
Implementation Details

- Low-level instrumentation of parallel bytecode interpreter
- Needed to have a very fast, non-intrusive data collection
- Incore data storage
- Dumped to file afterward
- Can be read and displayed
- Or used by a separate tool (e.g., IDRA)

Unifying Structure: Events

- Events: "interesting" points in the execution
  - They correspond to \((\text{intended}) \text{ observables}\)
- Events relate executions with depictions:

  \[
  \text{Observables} \rightarrow \text{Events} \rightarrow \text{Visualization(s)}
  \]

- Experimented with different event collection methods

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Collection</th>
<th>Event Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Low level</td>
<td>Start, end of tasks</td>
</tr>
<tr>
<td>Sequential</td>
<td>Metainterpreter</td>
<td>Call, success, . . .</td>
</tr>
<tr>
<td>Constraints</td>
<td>Annotated program</td>
<td>Store update</td>
</tr>
</tbody>
</table>

- Each method has its own merits and drawbacks
Events Bring Flexibility: New Scales

- Events can be *reinterpreted*
- E.g.: *Time View vs. Activity view*

Events Bring Flexibility: Abstraction

- Discard undesired information
- E.g.: FD history vs. variable range

⇑

Same annotations! ⇒
What Implementation Technique?

- Metainterpretative approach:
  - User-transparent, metainterpreter of medium difficulty
  - Slow (unpractical for real programs)
  - Exploit unfolding?
- Annotated program:
  - Some user effort — but user can select visualization target
  - Low burden on execution and library external to native system
  - Automated annotation tool very desirable!
- Low-level instrumentation:
  - Everything under control
  - Mandatory in some cases — but needs (full) source code
  - Quite a lot of delicate work
- Research area, related to debugging (e.g., Opium)

Questions

- How can graphical tools interact easily? (a single view is often not enough)
- How can the relationship execution / visualization be modeled?
- How do different visualizations of the same execution relate to each other?
- What is the best way to obtain data for a visualization? (Avoiding being tied to particular characteristics CLP system, if possible)

- Unsurprisingly, event-based design help to answer many of these questions
- Not only for CLP visualization!
Conclusions & Further Work

• Developed graphical representations for control, variable values, constraints, parallel execution
• Practical tools implemented and tested
• Many others freely available
• Also, developed abstraction of representations
• Some further ideas hard to implement: solver incompleteness / lack of projection, expensive computations, difficult / non-portable implementations . . .
• Several architectural designs
• Synergy visualization and assertion-based debugging: a research topic

Acknowledgments

• For support:
  ● ESPRIT Project ParForce
  ● ESPRIT Project DiSCiPl
  ● CICYT Project IPL-D
• For their work in the implementations herein presented:
  ● Roger Nasr
  ● Luis Gómez
  ● Ángel López Luengo
  ● José Manuel Ramos
  ● Göran Smedbäck
  ● Roland Karlsson
  ● . . . and surely others we are forgetting!


Pointers


Pointers