ABL Code Analytics
Unique Tools for Deep Code Analysis and Inspection

Greg Shah
Golden Code Development

Thursday October 25, 2018
Agenda

- Background
- Handling Flexibility and Scale
- Code Analytics
- Call Graph Analyzer
- Usage Tips
- How to Get Started
- Planned Improvements
Background
FWD is an open source toolset for modernizing 4GL applications.

**Explore**
Tools for reporting, statistics and analysis, which expose the inner workings of applications.

**Transform**
Fully automated transformation and modernization of entire applications without a manual rewrite.

**Transcend**
Compatible replacement for OpenEdge® which leverages an enhanced runtime, an upgraded 4GL language and Java to modernize and evolve applications.
Tools for reporting, statistics and analysis, which expose the inner workings of an application.

Explore

Advantages
- Reduce development effort.
- Improve code quality.
- Deeply understand existing code.
- Compensate for missing documentation.
- Empower developers to more capably handle:
  - The most complex refactoring and modernization projects.
  - Making changes at scale, even with the largest of applications.

Built-in reports and statistics provide hundreds of predefined views of the code and schemata.

Search for arbitrary 4GL syntax to find exact answers to questions that were previously difficult or impossible to answer.

Call Graph Analysis determines program dependencies, missing code and explores all reachable code paths. Reports allow identification and removal of dead code reducing application size by 25% to 40%.

Generates logic flow charts for all code in the application.

Visualize and interactively explore the call graph.
Why Use FWD Analytics?

- Reduce development effort.
- Improve code quality.
- Deeply understand and explore existing code.
- Compensate for missing documentation.
- Empower developers to more capably handle:
  - The most complex refactoring, transformation and modernization problems; AND
  - Making changes at scale, even with the largest of applications.
Handling Flexibility and Scale
Your Source Is Not Helping

- Programmatic analysis of an application needs to be aware of the ABL language syntax.
- Your source code is text. That text is non-regular and ambiguous.
  - different text, same meaning (non-regular code)
  - same text different meaning (ambiguous code)
- The more flexible the language’s grammar, the larger the set of possible valid constructs that can be written by the programmer.
  - Short Term: marginal time savings when writing new code.
  - Long Term: Increased costs of reading, maintenance, debugging, support and refactoring.
- To enable proper analysis of code, we must transform the text into a data structure that represents the purest form of the code.
- ASTs represent the code’s language syntax without syntactic sugar. The result is regular and unambiguous.
File -> Char -> Token -> Tree

Preprocessor

Hello.p

{hello.i &var="txt"}
def input parameter &var as char.
message "Hello " + &var + "!".

Hello.i

def input parameter txt as char.
message "Hello " + txt + "!".

Character Stream

... p n i f e d

Lexer

Token Stream

DOT (.)
STRING ("!")
PLUS (+)
SYMBOL (txt)
PLUS (+)
STRING ("Hello")
KW_MSG (message)
DOT (.)
KW_CHAR (char)
KW_AS (as)
SYMBOL (txt)
KW_PARAM (parameter)
KW_INPUT (input)
KW_DEFINE (def)

Abstract Syntax Tree (AST)

Parser
Scale Multiplies Issues

- Spread that syntax flexibility across tens of thousands of files and millions of lines of code.
- How do you find specific patterns?
- How do you even know if you have found all matches?
- Brute force is not enough.
- **Scale makes hard problems impossible (or at least impractical).**
- Automation is the only practical solution.
TRree Processing Language (TRPL)

- FWD provides tools to parse an entire application.
- Each source file and each schema file (.df) will be represented as an AST.
- TRPL is the analysis and transformation toolset in FWD which can operate on the entire set of ASTs as a batch.
- When you process trees, it is commonly called a tree walk.
- TRPL includes an engine that handles the tree walking for programs written in the TRPL language.
TRee Processing Language (TRPL) Event Model

Event:
- init
- walk
- descent
- next-child
- ascent
- post

Diagram:
- Define Parameter (def)
  - KW_INPUT (input)
  - SYMBOL (txt)
- BLOCK
  - STATEMENT
  - KW.MSG (message)
  - EXPRESSION
    - PLUS (+)
  - STRING ("")
AST Design for Transformation

- At parse time, there is a great deal of knowledge about the code. Encoding that knowledge into the tree makes downstream work easier.

- Resolving data types of each expression component is very important. This allows downstream code to calculate the type of each subexpression or expression in the application.

- By tracking resources by scope and creating linkages between the references and the definition, it becomes easier to work with these resources later.

- Structuring the tree is important. This can make it easier to walk the tree, match patterns and transform.
  - Multiple nodes can be rewritten as a single unambiguous node (e.g. KW_DEFINE KW_PARAMETER can be written as DEFINE_PARAMETER).
  - Artificial nodes can be inserted to group multiple related nodes.

- Calculated values and context-specific information are stored in the associated nodes as annotations.

- The ASTs created by FWD were designed with these issues (and others) in mind.
Report Generation

- After the entire application has been parsed, we can run the report generation step.

- This is a non-interactive process that runs a set of pre-defined TRPL programs to calculate a few hundred reports.

- This can take minutes for a small project or hours for a large project.

- Both the parsing and the report generation can be scripted and used in CI or build servers.

- After the reports are generated, they can be accessed via an interactive web interface.
Code Analytics
Reports

- List of predefined reports on left
- Currently viewed report on right
- Most reports are a set of mutually exclusive categories
- Summary statistics for the report at the top
- Individual categories have their own statistics
- Filter and sort columns using the column header
- Click on a row in the current report to see the exact list of matches
- Pagination controls at the bottom
Category Details

- List of predefined reports on left
- Exact list of matches for the selected category on the right
- Grouped by the file in which they appear
- Category statistics at the top

- Each match has line/column numbers in the “cache” file (fully preprocessed file)
- Filter, sort and pagination controls
- Click on a row of a specific match to go to the source view at that exact location
Source/AST View

- Fully preprocessed file on left with the match selected in pink.
- Current selection in the AST on the right.
- Source and AST views are linked, a selection on either side is highlighted and made visible on the other side.
- Hover mouse over an AST node to get details.
- Shift-click on the “root” node of the subtree to traverse up the tree.
- Ctrl-click on a child node to traverse down the tree.
Search

- If grep (regex searching) was fully aware of ABL syntax it would still not be as good as this.
- Write expressions or arbitrary complexity that match based on the full richness of the AST.
- The TRPL engine does the tree walk, you just specify exactly what you want to match.
- The TRPL expression syntax has many features that make it easier to process AST concepts, including the knowledge of the current AST node being visited.
- Code that cannot be implemented in a single expression can be put into a callable TRPL function and accessed from expressions.
- All AST nodes and other data being accessed are actually Java objects. You can call Java instance methods (no statics or generics at this time) on these objects and you can pass those same objects to Java methods or to TRPL functions.
- TRPL has a wide range of advanced AST processing features that can be leveraged.
Search: Field References

All references to guest.last-name:

\[
\text{type} == \text{prog.field_char} \text{ and } \text{getNoteString("schemaname")}.\text{equals("hotel.guest.last-name")}
\]

Assignments to guest.last-name:

\[
\text{type} == \text{prog.field_char} \text{ and } \text{getNoteString("schemaname")}.\text{equals("hotel.guest.last-name")} \text{ and } \text{parent.type} == \text{prog.assign} \text{ and } \text{childIndex} == 0
\]
Search: Buffers That Hide Buffers

Version 1:

```java
    type == prog.define_buffer and
    this.getChildAt(0).text.toLowerCase() ==
    this.getChildAt(1).getChildAt(0).text.toLowerCase()
```

Version 2:

```java
    parent.type == prog.kw_for and parent.parent.type ==
    prog.define_buffer and
    text.equalsIgnoreCase(parent.prevSibling.text)
```

Version 3:

```java
    upPath("DEFINE_BUFFER/KW_FOR") and
    text.equalsIgnoreCase(parent.prevSibling.text)
```
Search: FIND and NO-ERROR

• All FIND statements (62 matches):
  type == prog.kw_find

• FIND statements without NO-ERROR (26 matches)
  type == prog.kw_find and not this.descendant(2, prog.kw_no_error)

• FIND statements with NO-ERROR (36 matches)
  type == prog.kw_find and downPath("RECORD_PHRASE/KW_NO_ERROR")
Custom Reports

- Practice first with Custom Search
- Refine output with Custom Reports
  - Multiplex expressions to define “buckets”
  - Specify “dump” text preferences
- Persist the report definitions you find useful
- Organize by category and title
- Planned: Edit and Delete of custom reports
Custom Reports Example

- **Title:**
  FIND without NO-ERROR (by Buffer Name)

- **Condition:**
  
  `type == prog.kw_find and parent.type == prog.statement and not this.descendant(2, prog.kw_no_error)`

- **Multiplex Expression:**
  
  `this.getImmediateChild(prog.record_phrase, null).getChildAt(0).getAnnotation("schemaname")`

- **Category:**
  Database
Writing a Search Expression

- Look at the AST structure that corresponds to the code you are trying to match.
  - Write a code snippet and parse it, then view it in the source/AST view.
  - Use the predefined reports to find locations that already exist.
- Decide which node is the best situated. Usually this is about finding the node that is most “centrally” located.
- All the context for the expression is written from that node’s “perspective”.
- Use the token type first, to roughly match a set of possible nodes.
- Refine this to get an exact match by adding use of tree structure, annotations and text.
Look at the AST

- Tree visualization of DEFINE BUFFER
Don’t Fight the Tree!

- Let the structure of the AST solve the problem for you.
- TRPL will walk the tree for you.
- Your expression is being executed at each possible location in the entire application.
- It is a “callback” model with the events determined by the tree structure.
- The tree structure is the pure form of the language syntax as represented in your code.
- Matching on the tree is matching on the syntax.
- If you are finding yourself doing something “unnatural”, ask: how can the tree structure help me?
Call Graph Analyzer
Call Graph Analyzer

- Uses a graph database.
- Creates a “vertex” for every callable code block (e.g. function or internal procedure) in the application.
- Creates a “vertex” for every call-site (location that invokes one or more code blocks, e.g. RUN statement).
- Creates an “edge” between the call sites and the code blocks.
- Traversing from the a root entry point list (which you provide), we can walk the entire call graph of your application.
- This can be used to answer questions that are otherwise difficult or impossible to answer.
Call Graph Visualization

- Live model of the call tree using a “force directed graph”.
- User can load the graph from arbitrary locations.
- Traverse to “More” links with SHIFT-click (load just that node) or CTRL-click (add node to current graph snippet).
- Use this to explore the application.
- Useful to identify macro patterns that would be hard to see by reading source code.
- Zoom with mouse wheel, pan with drag on background.
- Still in very active development, this is an early version.
- Drag nodes to move them around. Hover to see details.
- SHIFT-click on AST nodes to go to the source/AST view.
Logic Flow Charts

- Live model of the logic flow chart for every callable block of code in the application.
- Accessed via the “Y” flow icon in the call graph visualization.
- Traverse to called code from call sites such as RUN.
- This is a form of documentation.
- Useful to identify macro patterns that would be hard to see by reading source code.
- Zoom with mouse wheel, pan with drag on background.
- Drag nodes to move them around. Hover to see details.
- SHIFT-click on AST nodes to go to the source/AST view.
Call Graph Reports

- **Ambiguous Call Sites**
  - Caused by indirect calling conventions and runtime determination of call targets.
  - To complete the graph, you provide hints to tell the call graph analyzer how to traverse these.
  - Iterative process to define hints, run the analyzer, review the latest ambiguous listing, provide hints… until there are no further ambiguous locations.

- **Dead Code**
  - In our experience, 25% to 40% of every non-trivial application of a certain age (10+ years) is dead code.
  - Once your graph is complete, this is an accurate list of the code you can delete.
  - Delete the code and put it through testing to confirm that the graph hints were correct.

- **Missing Call Targets**
- **External Dependencies**
How to Get Started
How to Get Started

- Download and install FWD.
- Download one of the sample template projects (there is one for ChUI and one for GUI).
- Follow the “Getting Started” instructions to get the template project installed and configured for your application code, including placing your code and schemata into the template project.
- Run the `ant report_server` target.
- Start the report server.
- Access the server at port 9443 via a browser.
- Full details of this process and all documentation: [https://proj.goldencode.com/projects/p2j/wiki/Code_Analytics](https://proj.goldencode.com/projects/p2j/wiki/Code_Analytics)
Planned Improvements
Planned Improvements

- Add more built-in call-graph analysis and reports. One example: identifying all locations that use a specific NEW SHARED variable (and the inverse).

- Move our existing transformation rules that calculate important properties to an early enough location that it can be integrated into reporting. This would include things like buffer scoping, frame scoping, index selection, transaction/block properties and more.

- Duplicate Code Identification. We can identify arbitrary code matches across the entire application using a bottom-up fingerprinting approach for each unique sub-tree in the application. By using fuzzy logic, we can match code that is the same whether it was cut and pasted or just independently coded the same way. Using these fingerprints we can turn duplicated code into common code.

- Improved TRPL syntax and structure, source level debugging.
Find Us On the Web!

- www.beyondabl.com
- facebook.com/beyondabl
- twitter.com/beyondabl
- plus.google.com/+beyondabl
- linkedin.com/company/fwd-project
- youtube.com/channel/UCK3pga7EKxAQVOV_CiYOR7g