JVM Implementation Strategies

Many programming languages today target the Java Virtual Machine (JVM) as platform. For instance, Clojure, Groovy, JRuby, Rhino JavaScript, SICP Scheme, ABCL Common Lisp all compile to JVM bytecode. The goal of this project is to investigate strategies for realizing certain language features on the JVM.

In particular, you are expected to:

- Perform a literature study on compilation strategies for stack-based VM architectures.
- Study the implementation of existing languages such as those mentioned above.
- Identify and categorize core language features that are challenging realize on the JVM. Amongst others, this may include: efficient dispatch, closures, tail recursion, call-by-reference, non-local control-flow (continuations) etc.
- Perform benchmarks with respect to performance of the various implementation strategies.
- Perform qualitative analyses of the trade-offs involved.

Requirements: affinity with programming language semantics and compiler construction. Very strong programming skills in Java.

Contact: Drs. Paul R. Griffioen (`p.r.griffioen@cwi.nl`), Dr. Tijs van der Storm (`storm@cwi.nl`).

CWI
Centrum Wiskunde & Informatica

Hybrid Partial Evaluation of JavaScript

Partial evaluation is a technique to optimize programs by partitioning the program input into static values and dynamic values. A partial evaluators evaluates a program using only the static value. The residual program is often faster than the original since all processing of static input is eliminated.

A special case of using partial evaluation is to turn an interpreter of a program into a compiler. Recently, this approach has seen increased interest for the implementation of efficient model- or domain-specific language (DSL) compilers.

To illustrate the essence of this approach, consider the following function: $\text{eval}(P, I) \rightarrow O$. This interpeter function takes a program P and some input I and produces some output O. A partial evaluator operates as follows: $\text{peval}(\text{eval}, P) \rightarrow \text{eval}_P$. The partial evaluator peval produces a specialized version of the interpreter eval_P; this program can be considered the *compiled* version of P. All the code to inspect the input program P is compiled away from the interpreter.

The goal of this project is to apply the technique of Hybrid Partial Evaluation [1] to JavaScript. This means you will develop a partial evaluator for JavaScript, so that DSL interpreters implemented in JavaScript can be partially evaluated to obtain compiled code. You will implement this partial evaluator in JavaScript itself. The thesis should reflect a solid understanding of this topic; it should describe the workings and limitations of the partial evaluator in minute detail. You are furthermore expected to evaluate the partial evaluator by measuring the performance gain.

Requirements: affinity with programming language semantics and compilers. Strong skills in JavaScript programming.

References:

1 Amin Shali and William R. Cook. *Hybrid Partial Evaluation*. 2011. `http://www.cs.utexas.edu/~wcook/Drafts/2011/javape.pdf`

2 William R. Cook and Ralf Lämmel. *Tutorial on Online Partial Evaluation*. 2011. `http://softlang.uni-koblenz.de/dsl11/`

3 Arjun Guha et al. *The Essence of JavaScript*. 2010. `http://www.cs.brown.edu/~sk/Publications/Papers/Published/gsk-essence-javascript/`

Contact: Dr. Tijs van der Storm (`storm@cwi.nl`).

NB: this project could also be executed for languages other than JavaScript, e.g., Ruby, Clojure, Dart etc.

Code Completion Framework Rascal

Rascal is a meta programming language and language workbench for source code analysis and transformation. One of the key application areas is the development of domain-specific languages (DSLs). Rascal provides IDE integration with Eclipse, so that you can easily provide IDE features for you DSL. Currently these features include: outlining, error marking, hyperlinking, hover documentation strings and context-menus. Rascal provides these services using call-backs, or hooks: the DSL developers only has to write (Rascal) functions for parsing, name-analysis, outline construction etc. Rascal uses the IMP Meta-IDE framework to connect these functions to Eclipse.

The goal of this project is to design a generic interface for language-parametric content-completion (autocomplete, "intellisense") in Rascal. When the user of a DSL hits Ctrl-Space a popup should appear with a selection of valid completions. The GUI part is handled by Eclipse/IMP; your goal is to allow the developers of a DSL to provide a function that returns a list of completions based on the (partial) parse tree of the program that is being edited.

In particular, you are expected to:

- Perform a solid related work study on content-completion approaches, such as implemented by xText, Spoofax, JetBrains MPS, Eclipse-JDT.
- Design and implementation of the content-completion interface in Rascal.
- Evaluation using case studies using real-life DSLs and programming languages. This will include performance evaluation.

Requirements: excellent Java programming skills. Affinity with domain-specific languages and model-driven engineering.

Contact: Dr. Jurgen Vinju (`jurgenv@cwi.nl`), Dr. Tijs van der Storm (`storm@cwi.nl`).

Grammar Debugging in Rascal

Rascal is a meta programming language and language workbench for source code analysis and transformation. One of the unique features of Rascal is its support for context-free grammars to define the syntax of programming languages. Developing a grammar, however, can be daunting task.

In this project the goal is to design a debugging interface for the Rascal parser. This should allow a grammar developer to step through the parsing process in order to understand the causes of parse errors and ambiguities.

In particular, you are required to:

- Perform a literature study on parser debugging and parser state visualization.
- Study and understand the Rascal parsing algorithm.
- Implement the necessary debugger hooks into the Rascal parser.
- Use the Rascal visualization library to visualize the parser state; the link with the context-free grammar should be explicit.
- Show that the debugger helps in finding the cause of parse errors and ambiguities.

Requirements: excellent Java and Rascal programming skills. Affinity with parsing algorithms is useful.

Contact: Contact: Prof. Dr. Paul Klint (`paulk@cwi.nl`), Dr. Jurgen Vinju (`jurgenv@cwi.nl`), Dr. Tijs van der Storm (`storm@cwi.nl`).

ATL Model Transformation in Rascal

In model-driven engineering (MDE) software is developed by constructing high-level (possibly domain-specific) models and subsequently transforming these to low-level models or code. A popular language for model transformation is [1]. Models are described using object-oriented meta models (class diagrams). An ATL script takes a number of models (instances) as input and produces one or more target models. In this project, the goal is to develop an interpreter prototype for ATL in [Rascal], which does not use object-oriented instance graphs as underlying representation, but values typed as algebraic datatypes (ADTs), sets and relations. A requirement is a textual notation for meta models (see e.g., [2]) and models. Your task is to evaluate if ADTs and relations form suitable building blocks for representing and transforming models in Rascal. You will test your prototype by executing existing ATL model transformations.

References:

1 ATL. `http://www.eclipse.org/atl/`

2 KM3. `http://en.wikipedia.org/wiki/KM3`

Contact: Dr. Tijs van der Storm (`storm@cwi.nl`)

Batches for JavaScript

Batch services are a new approach to distributed computation in which clients send batches of operations for execution on a server and receive hierarchical results sets in response. Batch services provide a simple and powerful interface to relational databases, with support for arbitrary nested queries and bulk updates. One important property of the system is that a single batch statement always generates a constant number of SQL queries, no matter how many nested loops are used. Additionally, batches can be used for arbitrary remote computation too.

The goal of this project is implement batches in JavaScript. This means extending JavaScript with a "batch" statement (similar to a for-loop). The statements within a batch will be analyzed and lifted into a script to be sent to the server. When the server returns the result to the client, the data has to be reintegrated in the client code. This means that batch code will be split into code that is run on the server and code that is run on the client.

You will evaluate your implementation by comparing the performance of typical and worst-case database usage scenarios against existing ORM solutions.

References:

1 Ben Wiedermann and William R. Cook. *Remote Batch Invocation for SQL Databases*. The 13th International Symposium on Database Programming Languages (DBPL), 2011. `http://www.cs.utexas.edu/~wcook/Drafts/2011/batchdb.pdf`

2 `http://www.cs.utexas.edu/~wcook/projects/batches/index.htm`

Translating Java to Ruby?

A common task in software engineering and maintenance is porting "old" code to new platforms. This can be an extremely expensive operation, especially if it means translating from one programming language to another. There are tools to (semi-)automatically perform such migrations, for instance from Visual Basic 6 to Visual Basic.NET, or from one dialect of Cobol to another.

In this project we are interested in an tool for the (semi-)automatic migration of Java code to Ruby. You will use Rascal and its JDT library to transform Java code to Ruby. Since in general this is hardly possibly [1], you will have to define a meaningful scope for the translation. In you thesis you should be able to show you understand the semantic differences between Java and Ruby, and elaborate why translation is so hard in this case. You should also accurately document you assumptions and motivations. The resulting tool should be evaluated through running of test suites of existing Java projects against the generated Ruby code.

1 Andrey A. Terekhov and Chris Verhoef. *The Realities of Language Conversions*. IEEE Software, November 2000. `http://www.cs.vu.nl/~x/cnv/s6.pdf`

Contact: Dr. Tijs van der Storm (`storm@cwi.nl`).

Ruby on Google Native Client

Today the Web browser functions as a new kind of application platform. The combination of HTML5 with fast JavaScript interpreters makes the Web a viable platform for desktop-like applications. Nevertheless, existing code that is not developed using these technologies is not easily ported to the (client-side) Web. In this project, the goals is to use Google Chrome's Native Client [1, 2] to run Ruby applications in the browser.

You are required to

- Find out how the latest Ruby interpreter (1.9.3) can be compiled using the Native Client APIs of Google Chrome. Preferably, you provide a *general* strategy for embedding programming language interpreters in Chrome.
- Design and implement an API to allow Ruby programs to access the HTML5 DOM Tree and Canvas element, so that interactive graphical applications can be developed in Ruby.
- Evaluate your approach by showing that it works for a large class of Ruby programs. You will have to elaborate and analyze the limits, restrictions and constraints.

Requirements: affinity with bleeding-edge Web technology; strong programming language skills in C/C++. Knowledge of Ruby.

Contact: Dr. Tijs van der Storm (`storm@cwi.nl`).

References:

1 `http://code.google.com/p/nativeclient`

2 `http://www.chromium.org/nativeclient/reference/research-papers`

TEX.js

TEX (in combination with LATEX) is a popular document typesetting system, originally developed by Donald Knuth in the 1980's [2]. TEX is implemented using a literate program tool called WEB [4]: from a single document both the source code and the documentation is generated. TEXis unique in the sense that its documentation generated from the WEB source is published as a book [3].

The goal of this project is to use Rascal to automatically transform the existing implementation of TEX to a JavaScript-based implementation so that it can be run in a Web-Browser.

Particular challenges include:

- Investigate different implementations of TEX and analyze the trade-offs in using any one of them for producing a JavaScript implementation. For instance, some are in C, others in Pascal. Some of them support Unicode, others do not.
- Develop a high-quality grammar to parse the input code (e.g., Pascal).
- How to compile the Pascal/C code to equivalent JavaScript?
- What are good file-system/OS abstractions that can be used in the browser?
- How to deal with the output of TEX: there are TEX implementations that produce Device-Independent (DVI) files [5], PDF [1] files, Postscript files.
- How to interface with fonts?

To evaluate the result of this research you will show that you can reproduce (vanilla) LATEX documents using TEX.js. An example test could be your thesis.

Requirements: affinity with TEX/LATEX; good skills in both Rascal and JavaScript.

References

1 PDF.js. `https://github.com/mozilla/pdf.js`

2 TEX. `http://en.wikipedia.org/wiki/TeX`

3 The TEXbook `http://www.tex.ac.uk/ctan/systems/knuth/dist/tex/`

4 CWEB. `http://sunburn.stanford.edu/~knuth/cweb.html`

5 DVI. `http://en.wikipedia.org/wiki/Device_independent_file_format`

Contact: Dr. Tijs van der Storm (`storm@cwi.nl`).