RAPID LANGUAGE ENGINEERING FOR GREEN-MARL, A DSL FOR HIGHLY PARALLEL GRAPH ANALYSIS

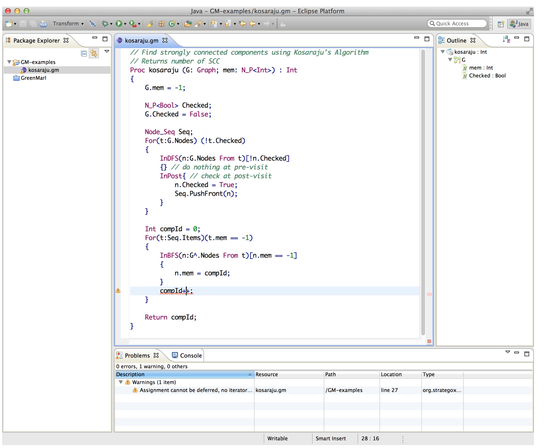
VLAD A. VERGU
EELCO VISSER

Vlad Vergu

Eelco Visser

Adoption of big data analytics relies on technological solutions to the problem of data scale, and depends on increased accessibility for data analysts who are not expert programmers.

Languages specific to the domain of data analytics can be more intuitive to analysts. These domain specific languages must abstract over the vastness of the data and allow analysts to work independently.



Spoofax based IDE for Green-Marl

SPOOFAX

The Spoofax Language Workbench [1] reduces the development effort of software languages. This gives language designers the time to focus on design decisions and language semantics.

Syntax definition, type systems, static analyses, model transformation, and code generation are specified using high-level declarative meta-languages provided by Spoofax. These abstract from the low-level implementation details of editor services and compilers for domain-specific languages.

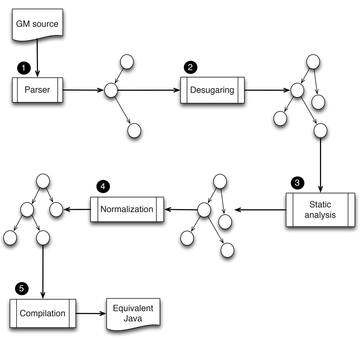
Spoofax-based languages have automatically generated IDEs based on Eclipse. The IDEs are fully-featured an customisable. Languages built with Spoofax are also directly usable from command line interfaces.

Languages can have multiple textual or graphical concrete representations. These are automatically synchronised in real-time [2]. The synchronisation recovers from errors during parsing and text-to-model synchronisation, preserves textual and graphical layout, and provides synchronised editor services such as selection sharing and navigation between editors.

GREEN-MARL

A community edition of Oracle's Green-Marl DSL [6] for graph analysis has been implemented in Spoofax in partnership with Oracle Labs. The DSL allows graph analysis algorithms in their textbook form to be directly translated to Green-Marl programs. Parallelism and workload distribution are implicit in Green-Marl programs eliminating the need for the programmer to reason about them.

The Spoofax based IDE and compiler for Green-Marl have been implemented during a three month internship at Oracle Labs. They are set to be open sourced and become the reference Green-Marl implementation. They will eventually replace the existing 38KLOC implementation in C/C++ with more extendible and maintainable specifications.



Runtime composition of language specifications

1 SYNTAX DEFINITION

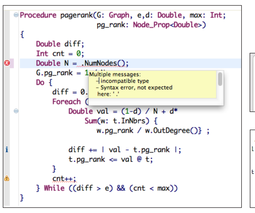
Green-Marl syntax is specified in SDF3 (Syntax Definition Formalism, version 3 [4]). SDF3 captures both the syntactic rules for the language and the layout preferences of the language. Spoofax derives from an SDF3 specification an Eclipse based editor with syntax highlighting, a parser with error reporting and recovery, a syntactic completion engine, outline views and a language specific pretty-printer.

```
templates

  Sentence.For      = <For {<ID> : <IterSources>} <Filter> <Sentence>>
  Sentence.ForEach  = <Foreach {<ID> : <IterSources>} <Filter> <Sentence>>
  Sentence.IfThen =
    <
    If {<Expr>}
      <Sentence>
    >

  Sentence.IfThenElse =
    <
    If {<Expr>}
      <Sentence>
    Else
      <Sentence>
    >
```

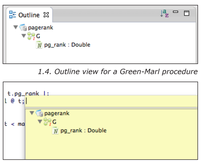
1.1. Example of syntax definition of Green-Marl statements in SDF3

```
Procedure pagerank(G: Graph, e,d: Double, max: Int;
                   pg_rank: Node_Prop<Double>)
{
    Double diff;
    Int cnt = 0;
    Double N = G.NumNodes();
    G.pg_rank = 1 / N;
    Do {
        diff = 0.0;
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {
                    w.pg_rank / w.OutDegree()} ;

            diff += | val - t.pg_rank |;
            t.pg_rank <= val @ t;
        }
        cnt++;
    } While ((diff > e) && (cnt < max))
}
```

1.2. Generated syntax highlighting for Green-Marl



1.3. Generated editor with parser, error reporting and error recovery



1.4. Outline view for a Green-Marl procedure

1.5. Quick-outline view for Green-Marl procedure

2 DESUGARING

Software languages can have many syntactic representations for equivalent semantic constructs, e.g. balanced and unbalanced If-Then-Else constructs. This syntactic sugar is eliminated (desugared) to reduce the variability in syntax and simplify the implementation of other language features.

Desugaring transformations in Spoofax are implemented in the Stratego [9] program transformation language.

```
desugar: |[ exp++; ]|                    -> |[ exp += 1; ]|
desugar: |[ If (exp) sentence ]|         -> |[ If (exp) sentence Else {} ]|
desugar: |[ Count (name : itersrc) filter ]| -> |[ Sum (name : itersrc) filter {1} ]|
```

2.1 Desugaring transformations in Stratego

```
Procedure example(G: Graph, p: Node_Prop<Int>)
{
    Int i = 0;
    i++;

    If(i > 0) {
        i = -i;
    }

    Foreach(n : G.Nodes) (n.p < 0) {
        i = i + n.p;
    }

    i = Count(n : G.Nodes)(n.p > 0);
}
```

2.2 Green-Marl code with syntactic sugar

```
Procedure example(G: Graph, p: Node_Prop<Int>)
{
    Int i = 0;
    i = i + 1;

    If(i > 0) {
        i = -i;
    } Else {}

    Foreach(n : G.Nodes) {
        If (n.p < 0) {
            i = i + n.p;
        }Else {}
    }

    i = Sum(n : G.Nodes)(n.p > 0) { 1 };
}
```

2.3 Green-Marl code with syntactic sugar eliminated

3 NAME AND TYPE ANALYSIS

In Spoofax, name binding rules for languages are declaratively specified in the NaBL meta-DSL [5]. This separates the semantics specification from the actual name resolution algorithm. Spoofax automatically generate a name analysis, resolution and context-sensitive autocompletions from a specification in NaBL. The generated analysis is automatically incremental ensuring the responsiveness of the IDE for larger projects [8].

Green-Marl's type system is implemented in the Stratego program transformation language. A new declarative meta-DSL for type systems will allow this implementation to be replaced with a declarative specification.

Results from the static analyses (messages, resolution paths, types, properties) are stored in a project's data store and are made available to subsequent analysis and transformation phases.

```
namespaces

  Procedure
  Variable

properties

  graph of Type                : OptGraphRef

binding rules

  Unit(_, _):
    scopes Procedure

  Block(_):
    scopes Variable

  Decl(t, [var], _):
    defines
      Variable var
        of type t
        of graph g
      in subsequent scope
      where
        t has graph g

  PropRef(exp, prop):
    refers to
      Variable prop
        of graph g
      where
        exp has type t
      where
        t has graph g
```

3.1 Partial name binding rules for Green-Marl in NaBL

```
type-of(!ctx):
  |[ exp0 assignop exp1; ]| -> <fail>
  where
    <?Assign() + ?Defer()> assignop
  with
    lhs-ty := <type-task(!ctx)> exp0;
    rhs-ty := <type-task(!ctx)> exp1;
    m-ty   := <type-match(!ctx, Coerce())> [rhs-ty, lhs-ty];
    <task-create-error-on-failure(!ctx, m-ty, INCOMPATIBLETY())> exp1
```

3.2 Typing rule for Green-Marl assignment

```
Procedure pagerank(G: Graph, e: Double, d: Double, max: Int,
                   pg_rank: N_P<Double>)
{
    Double diff;
    Int cnt = 0;
    Double N = G.NumNode();
    G.pg_rank = 1 / N;
    Do {
        diff = 0.0;
        Foreach (t: G.N
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {w.pg_rank / w.OutDegree()};

        diff += | val - t.pg_rank |;
        t.pg_rank <= val @ v;
    }
    cnt++;
  } While ((5 > e) && (cnt <
}
```

Multiple messages:
- unknown built-in function
- incompatible argument types
- incompatible type

Multiple messages:
- Unresolved reference
- Variable is not an iterator

3.3 Green-Marl editor with reported name and type errors

4 NORMALISATION

A subset of Green-Marl - a core - can be identified. All other semantic constructs can be rewritten in terms of this subset. Reduction of programs to this core (normalisation) helps to keep other parts of the compiler lighter and easier to implement. Normalisation is context-sensitive: scopes, types and names influence the transformation.

Normalisation transformations in Spoofax are implemented in Stratego. They have access to results from the name and type analyses. Spoofax's analysis framework ensures that names introduced during normalisation do not cause conflicts.

```
gm-to-gmcore-collection-group-assignment:
  |[ exp0 = exp1; ]| -> gm |[ Foreach(name2 : name1.Items) { name2.name0 = exp2; } ]|
  where
    PropRef(collection@VarRef(name1), name0) := exp0;
    collection-ty := <type-of> collection;
    <type-is-collection-set> collection-ty
  with
    name2 := <variable-future>;
    exp2 := <alltd(?|[ name1.name3 ]|;!|[ name2.name3 ]|)> exp1
```

4.1 Normalisation transformation for the Green-Marl batch assignment statement

```
Procedure example(G: Graph, prop : N_P<Float>)
{
    Node_Set(G) nodes;
    nodes.prop = 3.14;
}
```

4.2 Green-Marl code before normalisation

```
Procedure example(G: Graph, prop : N_P<Float>)
{
    Node_Set(G) nodes;
    Foreach(_safe_name_0 : nodes.Items) {
        _safe_name_0.prop = 3.14;
    }
}
```

4.3 Green-Marl code after normalisation

5 COMPILATION

The Green-Marl compiler translates correct input programs to functionally equivalent Java programs. Code generation transformations in Spoofax are implemented in Stratego.

Transformation rules in Stratego allows the consumed and emitted code to be written in concrete syntax. In the Green-Marl case, the left hand side of compilation rules uses concrete Green-Marl syntax and their right hand side uses concrete Java syntax. Use of concrete syntax in transformations simplifies specification and improves readability.

```
gm2j-statement:
  |[ name = exp; ]| -> bstm* |[ _.x_name = e_exp; ]|
  with
    x_name := <gm2j-name> name;
    e_exp  := <gm2j-expression> exp

gm2j-statement:
  |[ exp0.name = exp1; ]| -> bstm* |[ e_graph.x_setter(e_member, e_exp1); ]|
  where
    name-ty := <ndbl-collect-one-resolved-def; type-of> name;
    name-gr := <graph-of> name-ty
  with
    e_graph  := <gm2j-expression> name-gr;
    x_setter := <gm2j-setter-name> name;
    e_member := <gm2j-expression> exp0;
    e_exp1   := <gm2j-expression> exp1
```

5.1. Compilation transformation for Green-Marl variable and property assignments

TEST DRIVEN DEVELOPMENT

Spoofax supports TDD of languages. Unit tests for the various IDE/compiler components are written using the Spoofax Testing Language [7]. This supports tests for the parser, static analyses, name resolution and code completions. Test reports are presented in the Eclipse editor or in a JUnit style.

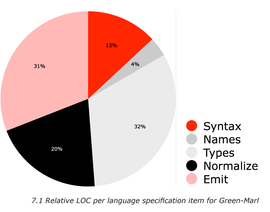
```
language Green-Marl

test parsing Sum [[

  Proc foo(G: Graph, A:N_P<Int>(G))
    {
        Int x;
        x = Sum {t: G.Nodes} {t.A} ;  // Test of parsing Sum
    }

]] parse succeeds

test comparable nodes [[
  Local foobar9(G : Graph, p : N_P<Int>(G)) {
    Node v, w;
    Bool b;
    b = v >= w;
  }
]] 0 errors

test name resolve [[
    Local foo(G : Graph, [p] : N_P<Int>(G)) {
          Node n;
          Int i;
          i = n.[p];
    }
]] resolve 2 to 1
```

6.1. Parsing, analysis and name resolution tests in SPT

EVALUATION

The Spoofax-based specification for Green-Marl required 2.5 months of development for an experienced Spoofax user. The entire specification is 3.5 KLOC in a mix of SDF3, NaBL and Stratego. Spoofax's highly-specialised meta-DSLs for syntax and name binding definition reduced the development effort of the parser and name analysis. Only 13% and 4% of the implementation are due to syntax and name analysis, respectively. The fact that 32% of the codebase implements Green-Marl's type system justifies using a meta-DSL for type systems in the future.

Green-Marl did not previously have an IDE. The Spoofax-based IDE/compiler covers approximately 80% of the Green-Marl language features. As full coverage is achieved this new implementation will become the de facto Green-Marl reference implementation. It will eventually replace the 38 KLOC of C/C++ code of the existing compiler.



7.1 Relative LOC per language specification item for Green-Marl

[1] Kats, Lennart CL, and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs." ACM Sigplan Notices. Vol. 45. No. 10. ACM, 2010.

[2] van Rest, Oskar, et al. "Robust real-time synchronization between textual and graphical editors." Theory and Practice of Model Transformations. Springer Berlin Heidelberg, 2013. 92-107.

[3] Erdweg, Sebastian, et al. "The state of the art in language workbenches." Software Language Engineering. Springer International Publishing, 2013. 197-217.

[4] Vollebregt, Tobi, Lennart CL Kats, and Eelco Visser. "Declarative specification of template-based textual editors." Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. ACM, 2012.

[5] Konat, Gabriël, et al. "Declarative name binding and scope rules." Software Language Engineering. Springer Berlin Heidelberg, 2013. 311-331.

[6] Hong, Sungpack, et al. "Green-Marl: a DSL for easy and efficient graph analysis." ACM SIGARCH Computer Architecture News. Vol. 40. No. 1. ACM, 2012.

[7] Kats, Lennart CL, Rob Vermaas, and Eelco Visser. "Integrated language definition testing: enabling test-driven language development." ACM SIGPLAN Notices. Vol. 46. No. 10. ACM, 2011.

[8] Wachsmuth, Guido H., et al. "A Language Independent Task Engine for Incremental Name and Type Analysis." Software Language Engineering. Springer International Publishing, 2013. 260-280.

[9] Bravenboer, Martin, et al. "Stratego/XT 0.17. A language and toolset for program transformation." Science of Computer Programming 72.1 (2008): 52-70.

TU Delft