Language Abstractions for Program Transformations

Karl Trygve Kalleberg
Department of Computer Science
University of Bergen
Norway

IBM T.J. Watson Research Center
Hawthorne, NY
29. Aug 2006
Outline

- Prologue
- Abstractions for Programs as Data
- Abstractions for Rewriting
- Abstractions for Graph-like Structures
- Abstractions for Cross-cutting Concerns
- Applications
- Epilogue
You will see a lot of code
You will see a lot of code

Don’t worry; you will forget it soon enough
Part I

Prologue
Building Abstractions

- **Purpose**
  - Capture and formalize domain
  - Reason about problems in the domain
  - Express solutions to problems in the domain

- **Products**
  - Domain objects, as libraries
  - Domain notation, as syntax
  - Domain-specific language (DSL) = library + syntax
Building Abstractions

- **Purpose**
  - Capture and formalize domain
  - Reason about problems in the domain
  - Express solutions to problems in the domain

- **Products**
  - Domain objects, as libraries
  - Domain notation, as syntax
  - Domain-specific language (DSL) = library + syntax

- **Focus**
  - Abstractions for program transformation (systems)
  - Program transformation systems: compilers, refacturers, interpreters, optimizers, documentation systems, verifiers
  - The DSL of this talk is Stratego
Alternatives for Capturing Domains

Language-centric Techniques for Capturing Domains

<table>
<thead>
<tr>
<th>Library</th>
<th>Notation</th>
<th>Library</th>
<th>Notation</th>
<th>Library</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose Language</td>
<td></td>
<td>General Purpose Language</td>
<td></td>
<td>Meta-Library</td>
<td></td>
</tr>
<tr>
<td>GPL</td>
<td></td>
<td>DSL</td>
<td>DSEL</td>
<td>DSAL</td>
<td></td>
</tr>
</tbody>
</table>

Motto

language = objects + notation
Part II

Abstractions for Programs as Data
Derivation of Language Infrastructure

- Syntax definition declares the structure of the language
  - Composed from modules – supports *language composition*
- Generators produce transformation artifacts from the definition
- Artifacts are used at runtime of the transformation system
The Transformation Pipeline

A Typical Program Transformation Pipeline

1. Source code is parsed into concrete syntax tree (CST).
2. CST is pruned to yield an *abstract syntax tree* (AST).
3. Type information or other analysis results are added.
4. Transformations are applied to the AST.
5. Result is serialized to file.
Terms and Signatures

- Terms
  - Analogous to syntax trees
  - Used to represent programs
  - Terminology taken from universal algebra

- Signatures
  - Analogous to document type definitions (DTDs).
  - Declare the structure of terms
Terms and Signatures

- **Terms**
  - Analogous to syntax trees
  - Used to represent programs
  - Terminology taken from universal algebra

- **Signatures**
  - Analogous to document type definitions (DTDs).
  - Declare the structure of terms

**Concrete Syntax**

```
x := 1 + 2
```

**Abstract Syntax Tree**

```
Assign("x", Plus(Int("1"), Int("2")))
```
### Signature for Cu (excerpt)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>: List(FunDef) -&gt; Program</td>
</tr>
<tr>
<td>FunDef</td>
<td>: Id * List(FunArg) * TypeName * Block -&gt; FunDef</td>
</tr>
<tr>
<td>FunArg</td>
<td>: Id * TypeName -&gt; FunArg</td>
</tr>
<tr>
<td>TypeName</td>
<td>: Id -&gt; TypeName</td>
</tr>
<tr>
<td>Int</td>
<td>: String -&gt; Expr</td>
</tr>
<tr>
<td>Var</td>
<td>: Id -&gt; Expr</td>
</tr>
<tr>
<td>Plus</td>
<td>: Expr * Expr -&gt; Expr</td>
</tr>
<tr>
<td>Multiply</td>
<td>: Expr * Expr -&gt; Expr</td>
</tr>
<tr>
<td>Call</td>
<td>: Id * List(Expr) -&gt; Expr</td>
</tr>
<tr>
<td></td>
<td>: Expr -&gt; Stm</td>
</tr>
<tr>
<td>Assign</td>
<td>: Id * Expr -&gt; Stm</td>
</tr>
<tr>
<td>If</td>
<td>: Expr * Stm * Stm -&gt; Stm</td>
</tr>
<tr>
<td>While</td>
<td>: Expr * Stm -&gt; Stm</td>
</tr>
<tr>
<td>Block</td>
<td>: List(Stm) -&gt; Stm</td>
</tr>
</tbody>
</table>
Part III

Abstractions for Rewriting
Rewrite Rules

Features
- basic atoms for building transformations
- transform one term to another
- based on structural pattern matching

Definition
- $R : l \to r$ where $c$
  - $R$ is the rule name
  - $R$ is the rule name
  - pattern $r$ (right-hand side) replaces pattern $l$ (left-hand side)
  - $c$ is the rule condition
Example: Simple Program Specialization

Goal
▶ Specialize matrix operations based on matrix layout.

The Specialize rule set

Specialize:
Plus(e0, e1) -> Call("plus_td_td", [e0, e1])
where
  <is-tri-diagonal> e0
; <is-tri-diagonal> e1

Specialize:
Multiply(e0, e1) -> Call("mul_td_any", [e0, e1])
where
  <is-tri-diagonal> e0

Example: Expression specialization

\[ x + y \Rightarrow \text{plus}_\text{td}_\text{td}(x, y) \]
Control application of rewrite rules

What is a strategy?

- Any rule invocation
- fail, id
- ! (build), ? (match)
- Any congruence (examples later)
- Any generic traversal, next
- Any strategy invocation

Strategy Combinators

- Sequential composition: \( s_1; s_2 \)
- Deterministic (left) choice: \( s_1<s+\ s_2 \)
- Choice: \( s_1<s+<s_2 \)
- \( \text{try}(s) = s+<\ id \)
Strategies

Control application of rewrite rules

What is a strategy?
- Any rule invocation
- fail, id
- ! (build), ? (match)
- Any congruence (examples later)
- Any generic traversal, next
- Any strategy invocation

Strategy Combinators

- Sequential composition: $s_1 \ ; \ s_2$
- Deterministic (left) choice: $s_1 \ <+ \ s_2$
- Choice: $s_1 \ < \ s_2 \ + \ s_3$
- $\text{try}(s) = s \ <+ \ id$
### Identity
- Syntax: `id`
- Always succeed
- Some laws
  - `id ; s ≡ s`
  - `s ; id ≡ s`
  - `id <+ s ≡ id`
  - `s <+ id ≢ s`

### Failure
- Syntax: `fail`
- Always fail
- Some laws
  - `fail <+ s ≡ s`
  - `s <+ fail ≡ s`
  - `fail ; s ≡ fail`
  - `s ; fail ≢ fail`

### Defined Combinators
- `try(s) = s <+ id`
- `repeat(s) = try(s; repeat(s))`
- `while(c, s) = if c then s; while(c,s) end`
- `do-while(s, c) = s; if c then do-while(s, c) end`
Visiting All Subterms

- Syntax: \texttt{all(s)}
- Apply strategy \texttt{s} to all direct sub-terms

\begin{verbatim}
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
\end{verbatim}
## Generic Traversal Strategies

### Visiting All Subterms

- **Syntax:** `all(s)`
- **Apply strategy** `s` to all direct sub-terms

```
Plus(Int("14"),Int("3"))
```

```
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```

```
bottomup(s) = all(bottomup(s)); s
topdown(s) = s; all(topdown(s))
downup(s) = s; all(downup(s)); s
alltd(s) = s <+ all(alltd(s))
innermost(s) = bottomup(try(s; innermost(s)))
```
Generic Traversal Strategies

Visiting All Subterms

- **Syntax:** `all(s)`
- **Apply strategy** `s` to all direct sub-terms

```latex
\text{Plus}(\text{Int("14")}, \text{Int("3")})
\text{stratego)}> \ all(!\text{Var("a")})
\text{Plus}(\text{Var("a")}, \text{Var("a")})
```

```latex
\text{bottomup}(s) = all(\text{bottomup}(s)); s
\text{topdown}(s) = s; all(\text{topdown}(s))
\text{downup}(s) = s; all(\text{downup}(s)); s
\text{alltd}(s) = s \leftarrow all(\text{alltd}(s))
\text{innermost}(s) = \text{bottomup}(\text{try}(s; \text{innermost}(s)))
```

Other Traversal Primitives

- **one(s)** – Apply `s` to one direct subterm
- **some(s)** – Apply `s` to as many direct subterms as possible
McCabe’s cyclomatic complexity

<table>
<thead>
<tr>
<th>Complexity Analysis Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Compute the number of possible execution paths through a function.</td>
</tr>
<tr>
<td>▶ Each control flow construct introduces another possible path.</td>
</tr>
<tr>
<td>▶ Number of control flow constructs determines complexity.</td>
</tr>
</tbody>
</table>
McCabe’s cyclomatic complexity

Complexity Analysis Algorithm

- Compute the number of possible execution paths through a function.
- Each control flow construct introduces another possible path.
- Number of control flow constructs determines complexity.

The cyclomatic-complexity Strategy

\[
\text{cyclomatic-complexity} = \text{occurrences(\text{?If}, \_)} \\
<+ \text{?If(\_, \_, \_)} \\
<+ \text{?While(\_, \_)} \\
<+ \text{?For(\_, \_, \_, \_)}
\]
**Totem Definition**

*emblem consisting of an object; serves as the symbol of a family*

- Used as tags (annotations) on variables
- Placed by the programmer
- Used to give hints to the optimizer
- Aid data-flow analyses and program specialization

**Example: Totem Annotation**

```
Matrix tri = load_from_file("...");
set_totem(tri, "tri-diagonal");
Matrix m = load_from_file("...");
Matrix r = m * tri;
```
Dynamic Rules

Features
- Capture context during traversal and rewriting
- Introduced/removed at runtime
- Encode context information into a rule set
- Can follow scoping rules of the subject language

Constructs
- Definition: `rules( R : l -> r where c )`
- Scoping: `{ R: s }
- Rule set operators:
  - `s1 \R/ s2` – union
  - `s1 /R\ s2` – intersection
  - `s1 /R\* s2` – fixpoint

Implemented as a Stratego library with syntax extensions.
Example: Totem Propagation

Sketch of Totem Propagation

prop-totem-set =
  ?Call("set_totem", [ Id(n), Lit(String(totem)) ])
; rules( Totem.n : n -> totem )

prop-totem-assign =
  Assign(Id(?n), prop-totem => e)
; if <get-totem> e
  then rules( Totem.n : n -> <get-totem> e )
else rules( Totem.n :- n )

prop-totem-if =
  If(id, prop-totem, id) /Totem\ If(id, id, prop-totem)
prop-totem: Sketch of a Totem Propagator

prop-totem =
    prop-totem-set
<+ prop-totem-assign
<+ prop-totem-if
<+ ( all(prop-totem)
    ; try(Specialize) )

Specialize:
    Plus(e0, e1) ->
    Call("plus_td_td", [e0, e1]) {Totem("tri-diagonal")}
    where
        <is-tri-diagonal> e0
    ; <is-tri-diagonal> e1

is-tri-diagonal = get-totem => Totem("tri-diagonal")

get-totem = Totem <+ \ x{Totem(t)} -> Totem(t) \
Concrete Syntax

Specialize using Concrete Syntax

Specialize:

\[ \lfloor e_0 + e_1 \rfloor \rightarrow \lfloor \text{plus}_td\_td(e_0, e_1) \rfloor \]
where

\(<\text{is-tri-diagonal}> e_0 \); \(<\text{is-tri-diagonal}> e_1 \)

Specialize:

\[ \lfloor e_0 \times e_1 \rfloor \rightarrow \lfloor \text{mul}_td\_\text{any}(e_0, e_1) \rfloor \]
where

\(<\text{is-tri-diagonal}> e_0 \)

▷ \( e_i \) are meta variables, i.e. Stratego variables
Part IV

Abstractions for Graph-Like Structures
Structural Definition of Annotated Terms

Approximate Grammar for ATerms

\[
\begin{align*}
t & := \text{bt} \quad \text{-- basic term} \\
& \quad | \text{bt} \{ t \} \quad \text{-- annotated term} \\
\text{bt} & := \text{C} \quad \text{-- constant} \\
& \quad | \text{C}(t_1,\ldots,t_n) \quad \text{-- n-ary constructor} \\
& \quad | (t_1,\ldots,t_n) \quad \text{-- n-ary tuple} \\
& \quad | [t_1,\ldots,t_n] \quad \text{-- list} \\
& \quad | "ccc" \quad \text{-- quoted string} \\
& \quad | \text{int} \quad \text{-- integer} \\
& \quad | \text{real} \quad \text{-- floating point number} \\
& \quad | \text{blob} \quad \text{-- binary large object}
\end{align*}
\]
Maximal Sharing

Directed Acyclic Graphs
- Subterms always shared
- Ensured at term construction
- "Hash consing"
- O(1) term comparison
- O(1) term copying
- no destructive update
- no cycles

Plus(Int("1"), Plus(Int("1"), Int("1"))
Graph-like Structures

Call graph

Control flow graph

Call graph

Control flow graph

Graphs in program analysis and transformation

- Encode auxiliary models of the program code
- Complement the syntax tree
- Capture context – used to “deposit” analysis results
Term References

Features

- support cycles
- encode concept of edges in a graph
- support creation, binding and dereference
- first class: copy, compare, assign, pass around
- retain generic traversals, rewrite rules, ...

Constructs

- Build and bind: $!r \sim p(x)$
- Bind or match: $?r \sim p(x)$
- Dereference: $^r$

Implemented as a Stratego library with syntax extensions.
Using Reference Operators

stratego> \!r\sim\text{Var}("a") // build and bind
Ref(0)
stratego> \!^r // dereference
\text{Var}("a")
stratego> \!r\sim\text{Var}("b") ; \!^r // build and rebind, deref
\text{Var}("b")
stratego> ?r\sim\text{Var}("b") // match
Ref(0)
stratego> ?r\sim\text{Int}("1") // match (fail), not bound to Int("1")
command failed
stratego> ?r\sim\text{Var}("b") ; \!r' // bind and match, build
Ref(0)
Example: Call Graphs

```plaintext
compute-call-graph

compute-call-graph = {| FunLookup: add-refs ; add-call-markers |}

add-refs = topdown(try(InsertFunRef))

InsertFunRef:
  x@FunDef(n,args,rt,b) -> r
where
  !r~FunDef(n,args,rt,b)
  ; rules(FunLookup: n -> r)

add-call-markers = {| CalledBy, CurFun:
  with-fundefs(wdownup(try(register-fun), try(AddCallRef)))
  ; with-fundefs(wrap-ref(AddCalledByRef)) |}

with-fundefs(s) = Program(map(s), id)
register-fun = ?r~FunDef(.,.,.,.); rules(CurFun: . -> r)

AddCallRef:
  Call(n, xs) -> Call(n, xs, r)
where
  <FunLookup> n => r
  ; CurFun => z
  ; rules(CalledBy :+ n -> z)

AddCalledByRef:
  FunDef(n,a,t,b) -> FunDef(n,a,t,ns,b)
where
  <bagof-CalledBy> n => ns
```
Phased Traversals

Features

- Reuse generic traversal model from terms
- Deal with cycles (cycles $\Rightarrow$ non-termination)
- Support rewriting

Definitions

- **phase(s)** – traverse a spanning tree, apply $s$ at each “node”
  - mark references while traversing, do not revisit references
- **wrap-ref(s)** – apply $s$ to term of reference $r$ then rebind $r$
  - Semantics: deref unconditionally, rewrite, rebind
- **wrap-phase-ref(s)** – same, but respect phase.
  - Semantics: deref iff node is unmarked, rewrite, rebind

- Phases can be nested.
Example: Depth First Search

\[
\text{dfs}
\]

\[
dfs(l : a \times a -> a, es) = \text{phase}(\text{wall}(dfs(l, es | 0)))
\]

\[
dfs(l : a \times a -> a, es | n) =
\]

\[
\text{wrap-phase-ref}(\text{where}(es => \text{edges})
\]

\[
; \text{where}(l(|n) => \text{label})
\]

\[
; \text{where}(\text{wall}(dfs(l, es | <inc> n)) > \text{edges})
\]

\[
; !\text{label})
\]

- \( l(|n) \) – computes the labels, gets depth as argument
- \( es \) – computes the edges
compute-mutually-recursive-functions

compute-mutually-recursive-functions =
    scc(time-count, calls-as-outbound, calledby-as-outbound)

calls-as-outbound = collect( Call(_,_,x) -> x )
calledby-as-outbound = collect( FunDef(_,_,_,_,x,_) -> x ) ; concat

dfs-collect(l : a * a -> a, es) =
    phase(all(|C: dfs-collect(l, es | 0) ; bagof-C|))

dfs-collect(l : a * a -> a, es | n) =
    ?r~_
    ; wrap-phase-ref(
        where(es => edges)
        ; where(l(|r) => label)
        ; where(<all(dfs-collect(l, es | <inc> n))> edges)
        ; !label)

sort-vertices = sort-list(LSort(where((?r;!^r; FinishTime,?r';!^r'; FinishTime); gt)))
collect-components(|r) = rules(C :+ _ -> r)
inc-time = (Time <+ !0) => n ; where(inc => n'; rules(Time: _ -> n'))
time-count(|n) = ?x; where(inc-time => n'); rules(FinishTime: x -> n')

scc(l : a * a -> a, es, res) = { |FinishTime, Time: dfs(l, es)
    ; sort-vertices
    ; dfs-collect(collect-components, res)
    ; filter(not(?[])) |}
Example: Mutually Recursive Functions (2)

Input: Call Graph

Output: Set of Cliques

\[[(a, b, e), (f, g), (c, d, h)]\]
Part V

Abstractions for Cross-cutting Concerns
## AspectStratego – Aspects in Stratego

### Features
- Support meta-rewriting – rewriting of Stratego code
- “Tame” the rewriting power of Stratego into a DSEL
- Higher-level, more declarative rewriting
- Instantiation of AspectJ-like language in Stratego

### Definitions
- **Pointcut**
  - expressed with logical combinators and joinpoint predicates
  - identify locations in the program
  - based on static and dynamic program properties

- **Joinpoint**
  - a point in the program execution
  - control-flow passes twice
  - into and out of the subcomputation at that point
Joinpoints

Joinpoint Predicates – Static

- calls(nameexpr => n)
- strategies(nameexpr => n)
- rules(nameexpr => n)
- matches(metapattern => t)
- builds(metapattern => t)
- fails
Joinpoint Context Predicates – Dynamic

- `withincode (nameexpr => n)`
- `args(n_0, ..., n_n)`
- `rhs(metapattern => t)`
- `lhs(metapattern => t)`

Combinators

- `;` – and
- `+` – or
- `not(j)` – not

Implemented as a “meta”-library with syntax extensions.
Example: Dynamic type checking

The typecheck aspect

module typecheck-aspect
aspects
  pointcut typecheck-rules(n, t) = rules(n) ; rhs(t)

aspect typechecker =
  around : typecheck-rules(n, t) =
    proceed
    ; ( typecheck(|t)
      <+ (log(|incorrect-term(|t)) ; fail) )

The typecheck aspect

module typecheck-aspect
aspects
  pointcut typecheck-rules(n, t) = rules(n) ; rhs(t)

aspect typechecker =
  around : typecheck-rules(n, t) =
    proceed
    ; ( typecheck(|t)
      <+ (log(|incorrect-term(|t)) ; fail) )

Example: Typechecking Specialize

Specialize: Plus(e0, e1) -> Call(...) where ...

Specialize’: Plus(e0, e1) -> Call(...) where ...
Specialize = Specialize’ ; ?t ; typecheck(|t)
Or: What do we use program transformation systems for, anyway?
Moldable Failure Handling

- Domain-specific aspect-language for error handling
- Decouple failure handling mechanism from policy
- Declarative language for specifying handlers
  - on multiple levels of granularity (expression, block, function, namespace)
  - for different policies (pre/post conditions on functions)
  - allows separate (re)declaration of normality and failure

C+Alert

- Language independent – prototype implemented for C
- Built with Transformers (C99 transformation framework) and Stratego/XT
Separation of Concern: Coordinate Free Numerics

- Separate choice of coordinate system from implementation of numerical solvers
- Provides significant flexibility in reformulating solutions to numerical problems
- Build Lego™-like library for numerical analysis
- Extend compiler with domain-knowledge
- Goal: Fortran-like performance, with high-level notation

CodeBoost

- C++ transformation framework
- Built with Stratego/XT on top of OpenC++

www.codeboost.org
Dryad – Java 1.5 front-end
▶ Extensible Java 1.5 grammar
▶ AspectJ extensions
▶ Semantic analysis

Stratego/XT
▶ Stratego – strategic term rewriting language, as shown
▶ XT – transformation components
  ▶ SDF – parser toolkit from CWI
  ▶ Box – pretty-printing language and generator
  ▶ RTG – format checking language and generator
Part VII

Epilogue
Features

- Content assist
- Source code navigation
- Project builder
- Source code outline
- Syntax highlighting for Stratego and SDF
- Accurate parenthesis matching
- Bundled documentation

www.spoofax.org
If I build language infrastructure, who are my clients?

Dividing the World

- **Inspectors** – read-only interface (to data)
  - program checkers, static analyzers, documentation tools
- **Generators** – write-only interface (to data)
  - model-to-code transformers, generative programming tools, some compiler backends
- **Transformers** – r/w access
  - program transformation systems, optimizers, language extensions

Levels of Integration

- Data – serialization: read and write data
- Service – invoke pre-defined interface to compiler logic
- “Complete” – expose full API to internal representations
Conclusion

Experiences

- We (re)build language infrastructure all the time
- Reuse story of language infrastructures is very weak
- New language abstractions do improve expressiveness
- The “DSL = library + syntax” approach is easy and effective
- Eclipse plugin perhaps most practical contribution to the Strategoverse

www.stratego-language.org