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

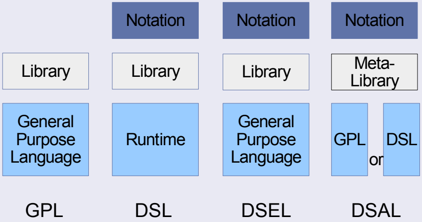
Building Abstractions

- 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, refactorers, interpreters, optimizers, documentation systems, verifiers
  - The DSL of this talk is Stratego

Alternatives for Capturing Domains

Language-centric Techniques for Capturing Domains



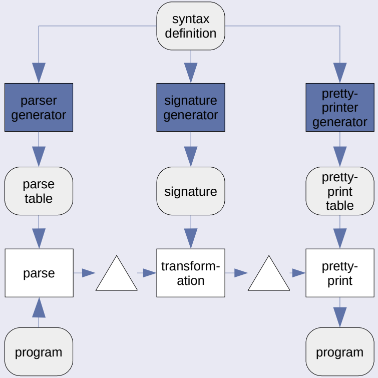
Motto

language = objects + notation

Part II

Abstractions for Programs as Data

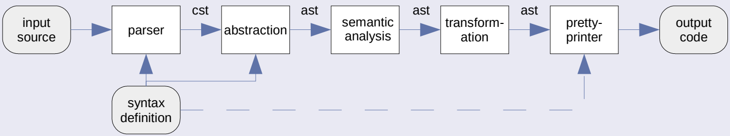
Derivation of Language Infrastructure



- 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
```

$\Downarrow$

Abstract Syntax Tree

```
Assign("x", Plus(Int("1"), Int("2")))
```

Example: Signature for a C-like Language

Signature for Cu (excerpt)

```
Program        : List(FunDef) -> Program
FunDef         : Id * List(FunArg) * TypeName * Block
                 -> FunDef
FunArg         : Id * TypeName -> FunArg
TypeName       : Id -> TypeName
Int            : String -> Expr
Var            : Id -> Expr
Plus           : Expr * Expr -> Expr
Multiply       : Expr * Expr -> Expr
Call           : Id * List(Expr) -> Expr
               : Expr -> Stm
Assign         : Id * Expr -> Stm
If             : Expr * Stm * Stm -> Stm
While          : Expr * Stm -> Stm
Block          : List(Stm) -> Stm
```

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 -> r where c`
  - *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` ⇒ `plus_td_td(x, y)`

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

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
- `try(s) = s <+ id`

Strategies (2)

Identity

- Syntax: `id`
- Always succeed
- Some laws
  - `id` ; $s \equiv s$
  - s ; `id` $\equiv s$
  - `id <+` $s \equiv$ `id`
  - s `<+ id` $\not\equiv s$

Failure

- Syntax: `fail`
- Always fail
- Some laws
  - `fail <+` $s \equiv s$
  - s `<+ fail` $\equiv s$
  - `fail` ; $s \equiv$ `fail`
  - s ; `fail` $\not\equiv$ `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
```

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"))
```

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

```
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)))
```

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

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.

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

```
cyclomatic-complexity =
 occurrences(?If(_, _)
  <+ ?If(_, _, _)
  <+ ?While(_, _)
  <+ ?For(_, _, _ ,_))
```

Totems

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)
```

Example: Totem Propagation (2)

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:
  |[ e0 + e1 ]| -> |[ plus_td_td(e0, e1) ]|
  where
    <is-tri-diagonal> e0
  ; <is-tri-diagonal> e1

Specialize:
  [| e0 * e1 ]| -> |[ mul_td_any(e0, e1) ]|
  where
    <is-tri-diagonal> e0
```

- e_i are meta variables, i.e. Stratego variables

Part IV

Abstractions for Graph-Like Structures

Structural Definition of Annotated Terms

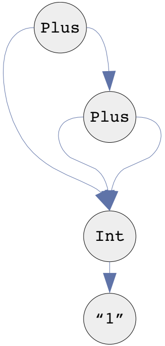
Approximate Grammar for ATerms

```
t  := bt                    -- basic term
    | bt { t }              -- annotated term

bt := C                     -- constant
    | C(t1,...,tn)          -- n-ary constructor
    | (t1,...,tn)           -- n-ary tuple
    | [t1,...,tn]           -- list
    | "ccc"                 -- quoted string
    | int                   -- integer
    | real                  -- floating point number
    | blob                  -- binary large object
```

Maximal Sharing

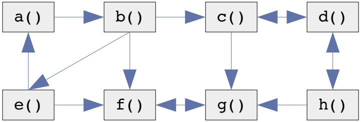
Directed Acyclic Graphs

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

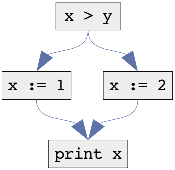


`Plus(Int("1"), Plus(Int("1"), Int("1")))`

Graph-like Structures



Call graph



Control flow graph

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~p(x)`
- Bind or match: `?r~p(x)`
- Dereference: `^r`

Implemented as a Stratego library with syntax extensions.

Term References (2)

Using Reference Operators

```
stratego> !r~Var("a")   // build and bind
Ref(0)
stratego> !^r   // dereference
Var("a")
stratego> !r~Var("b") ; !^r   // build and rebind, deref
Var("b")
stratego> ?r~Var("b")   // match
Ref(0)
stratego> ?r'~Int("1")   // match (fail), not bound to Int("1")
command failed
stratego> ?r'~Var("b") ; !r'   // bind and match, build
Ref(0)
```

Example: Call Graphs

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

dfs

```
dfs(l : a * a -> a, es)      = phase(wall(dfs(l, es | 0)))
dfs(l : a * a -> a, es | n) =
  wrap-phase-ref(where(es => edges)
                ; where(l(|n) => label)
                ; where(<wall(dfs(l, es | <inc> n))> edges)
                ; !label)
```

- `l(|n)` – computes the labels, gets depth as argument
- `es` – computes the edges

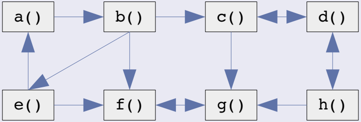
Example: Mutually Recursive Functions

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 subcomptation at that point

Joinpoints

Joinpoint Predicates – Static

- `calls(`*nameexpr* `=>` *n*`)`
- `strategies(`*nameexpr* `=>` *n*`)`
- `rules(`*nameexpr* `=>` *n*`)`
- `matches(`*metapattern* `=>` *t*`)`
- `builds(`*metapattern* `=>` *t*`)`
- `fails`

Joinpoints (2)

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) )
```

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) )
```

Example: Typechecking Specialize

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

$$\Downarrow$$

```
Specialize': Plus(e0, e1) -> Call(...) where ...
Specialize = Specialize' ; ?t ; typecheck(|t)
```

Part VI

Applications

Or: What do we use program transformation systems for, anyway?

Embedding Domain-Specific Aspect Languages

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

Domain-specific Optimization

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`

Compiler and Language Environments

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

Spoofax – Eclipse plugin for Stratego

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`

Plea for Improved Language Infrastructure Reuse

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`