Implementation of Partial Evaluation in Stratego/XT

Eelco Visser & Karina Olmos & Martin Bravenboer

Center for Software Technology, Utrecht University, Utrecht, The Netherlands

http://www.stratego-language.org

Workshop on Functional Refactoring

Canterbury

February 10, 2004

Program Transformation by Term Rewriting

- Programs can be represented as trees (or terms)
- Transformation corresponds to tree (or term) rewriting
- Rewrite rules correspond to basic transformations (derived from equations over expressions)
- Rewrite rules compose into rewrite system
- Rewrite engine automatically applies rules

Limitations of Pure Term Rewriting

- Lack of control over application of rules
  - Non-termination
  - Non-confluence
  - Common solution: encode with extra rules
  - Stratego: programmable rewriting strategies
- Context-free nature of rewrite rules
  - Example: inlining, constant propagation
  - Common solution: encode with extra rules
  - Stratego: scoped dynamic rewrite rules

Stratego/XT

- Stratego: language for program transformation
  - rewrite rules
  - programmable strategies
  - generic traversal
  - scoped dynamic rewrite rules
  - concrete object syntax
- XT: bundle of tools for program transformation
  - Stratego: library, compiler, interpreter
  - SDF: syntax definition
  - GPP: pretty-printing
  - ATerms: exchange format
  - XTC: transformation tool composition

Applications (Transformations)

- Bound variable renaming (RULE'01)
- Function inlining (RULE'01)
- Dead code elimination (RULE'01)
- Interpretation (LDTA'02)
- Constant propagation (WRS'02)
- Instruction selection (RTA'02)
- Partial evaluation: strategy specialization (WRS'01), type specialization (SCAM'03)
- Loop optimizations
- Simplification in functional compiler
- Optimizations in Stratego compiler

Applications (Systems)

- Tiger compiler
- Octave compiler
- CodeBoost (C++ transformation system)
- Helium simplifier
- AutoBayes optimizer
- Stratego compiler
- XT tools

Application: Partial Evaluation

- Partial evaluation
  - Function specialization
  - Binding-time annotations
  - Binding-time analysis
- Demonstration
- Constant folding
  - rewrite rules, strategies, concrete syntax
- Unfolding
  - dynamic rules, undefining rules
- Function specialization
- Binding-time annotation

Function Specialization

```
let function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n / 2))
      else x * power(x, n - 1)
in ... power(z, 5) ... end
```

Function specialization

```
let function power5(a : int) : int = a * power4(a)
    function power4(l : int) : int = square(power2(l))
    function power2(v : int) : int = square(power1(v))
    function power1(c : int) : int = c * power0(c)
    function power0(j : int) : int = 1
 in ... power5(z) ... end
```

Specialization with transition compression

```
... z * square(square(z * 1)) ...
```

Binding-Time Analysis

```
let function power(x : int, n : int) : int =
      if n = 0
      then 1
      else if even(n)
           then square(power(x, n / 2))
           else x * power(x, n - 1)
in ... power(z, 5) ... end
```

Binding-time annotation indicates `~static` and `<dynamic>` code

```
let function power_sd(x: int, n : <int>) : <int> =
      <if n = ~0
       then ~1
       else if even_d(n)
            then square_d(power_sd(~x, n / ~2))
            else ~x * power_sd(~x, n - ~1)>
in ... <power_sd(<z>, ~5)> ... end
```

Demonstration

- Tiger transformation tools
  - run, pretty-print, expand-imports
  - specialize, bta, ...
- Interactive environment
  - command-line tools called from XEmacs menu
  - whole program transformations
- Examples
  - power
  - ackermann
  - string/term pattern matching
  - term rewriting

Constant Folding

```
if 5 = 0 then 1
else if even(5)
     then square(power(x, 5 / 2))
     else x * power(x, 5 - 1)
```

$\Rightarrow$

```
x * power(x, 4)
```

Constant Folding: Rewrite Rules

```
rules

  EvalBinOp :
    |[ +(i, j) ]| -> |[ k ]|
    where <add>(i, j) => k

  EvalIf :
    |[ if i then e1 else e2 ]| -> e2
    where <eq>(i,0)

  EvalIf :
    |[ if i then e1 else e2 ]| -> e1
    where <not(eq)> (i, 0)
```

Concrete vs Abstract Syntax

```
rules

  EvalBinOp :
    BinOp(PLUS(), Int(i), Int(j)) -> Int(k)
    where <add>(i, j) => k

  EvalIf() :
    If(Int(i), e1, e2) -> e2
    where <eq>(i, 0)

  EvalIf() :
    If(Int(i), e1, e2) -> e1
    where <not(eq)>(i, 0)
```

Rewriting Strategy

Standard rewrite engines

- exhaustively apply all rules

Stratego

- select rewrite rules to apply
- apply with appropriate strategy
- strategies are user-definable

Constant Folding Strategy

An exhaustive strategy for constant folding

```
strategies
  constant-fold =
    innermost(
      EvalBinOp <+ EvalRelOp <+ EvalIf
    )
```

Single bottom-up pass is sufficient for constant folding

```
strategies
  constant-fold =
    bottomup(try(
      EvalBinOp <+ EvalRelOp <+ EvalIf
    ))
```

Definition of Strategies

```
strategies

  bottomup(s) =
    rec x(all(x); s)

  topdown(s) =
    rec x(s; all(x))

  downup(s1, s2) =
    rec x(s1; all(x); s2)

  oncetd(s) =
    rec x(s <+ one(x))

  alltd(s) =
    rec x(s <+ all(x))

  innermost(s) =
    rec x(bottomup(try(s; x)))
```

Constant Folding only Static Code

```
reduce-static =
     ?Dynamic(<reduce-dynamic>)

  <+ If(reduce-static, id, id); EvalIf; reduce-static

  <+ all(reduce-static)
     ; try(EvalBinOp <+ EvalRelOp <+ EvalInt)

reduce-dynamic =
     ?Static(<reduce-static>)

  <+ all(reduce-dynamic)
```

Substituting Variables

```
ReduceLetVar :
  |[ let var x ta := e1 in e2 end ]| -> |[ e2 ]|
  where <is-value + Var(id)> e1
      ; rules(ReduceVar : |[ x ]| -> |[ e1 ]|)


reduce-static =
     ...

  <+ ReduceVar

  <+ Let([VarDec(id, id, reduce-static)], reduce-static)
     ; ({| ReduceVar : ReduceLetVar; reduce-dynamic |}
        <+ Let(id, reduce-dynamic))

  <+ ...
```

Unfolding Function Calls

```
DeclareReduceCall =
  ?fdec@|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ let d* in e end ]|
        where <zip(\(FArg|[ x : tid ]|, e) -> |[ var x : tid := e ]|\)>
                (x*, a*) => d*
    )

reduce-static =
     ...

  <+ Let([FunDecs(map(DeclareReduceCall))], reduce-static)

  <+ Call(id, map(reduce-static)); UnfoldCall

  <+ ...
```

Function Specialization (1)

Replace function declarations by specialized function declarations

```
reduce-dynamic =
  ...
  <+ Let([FunDecs(map(DeclareReduceCall))], reduce-dynamic)
     ; Let([FunDecs(map(?FunDec(<id>,_,_,_); bagof-Specialization); concat)], id)
  ...

DeclareReduceCall =
  ?fdec@|[ function f(x*) ta = e ]|
  ; extend rules(
      Specialization :
        f -> Undefined
    )
  ; ...
```

Function Specialization (2)

Generate specializations when dynamic function call is encountered

```
reduce-dynamic =
  ...
  <+ Call(id, map(Static(reduce-static) <+ !Dynamic(<reduce-dynamic>)))
     ; ReduceCallDynamic
  ...

ReduceCallDynamic :
  |[ f(a1*) ]| -> |[ g(a3*) ]|
  where <dummy-dynamic-args> a1* => a2*
      ; <RetrieveSpecialization <+ GenerateSpecialization> |[ f(a2*) ]| => g
      ; <select-dynamic-args> a1* => a3*
```

Generate Specialized Function

```
GenerateSpecialization :
  |[ f(a1*) ]| -> g
  where
    <map({\|[ <e> ]| -> |[ <x> ]| where new => x\} <+ ...)> a1* => a2*
    ; <UnfoldCall> |[ f(a2*) ]| => e
    ; new => g




    ; <reduce-static> e => e'
    ; ... => x* ; ... => ta
    ; extend override rules(
        Specialization :
          f -> |[ function g(x*) ta = e' ]|
      )
```

Generate Specialized Function and Memoize it

```
GenerateSpecialization :
  |[ f(a1*) ]| -> g
  where
    <map({\|[ <e> ]| -> |[ <x> ]| where new => x\} <+ ...)> a1* => a2*
    ; <UnfoldCall> |[ f(a2*) ]| => e
    ; new => g
    ; <dummy-dynamic-args> a1* => a3*
    ; rules(
        RetrieveSpecialization :
          |[ f(a3*) ]| -> g
      )
    ; <reduce-static> e => e'
    ; ... => x* ; ... => ta
    ; extend override rules(
        Specialization :
          f -> |[ function g(x*) ta = e' ]|
      )
```

Reduce Static Code

```
reduce-static =
     ?Dynamic(<reduce-dynamic>)

  <+ ReduceVar

  <+ Let([VarDec(id, id, reduce-static)], reduce-static)
     ; ({| ReduceVar : ReduceLetVar; reduce-dynamic |}
        <+ Let(id, reduce-dynamic))

  <+ Let([FunDecs(map(DeclareReduceCall))], reduce-static)

  <+ Call(id, map(Dynamic(reduce-dynamic) <+ !Static(<reduce-static>)))
     ; reduce-call-static(reduce-static)

  <+ If(reduce-static, id, id); EvalIf; reduce-static

  <+ all(reduce-static)
     ; try(EvalBinOp <+ EvalRelOp <+ EvalInt)
```

Reduce Dynamic Code

```
reduce-dynamic =
     ?Static(<reduce-static>)

  <+ ReduceVar

  <+ Let([VarDec(id, id, reduce-dynamic)], reduce-dynamic)
     ; ({| ReduceVar : ReduceLetVar; reduce-dynamic |}
        <+ Let(id, reduce-dynamic))

  <+ Let([FunDecs(map(DeclareReduceCall))], reduce-dynamic)

  <+ Call(id, map(Static(reduce-static) <+ Dynamic(reduce-dynamic)
                                         <+ !Dynamic(<reduce-dynamic>)))
     ; ReduceCallDynamic

  <+ all(reduce-dynamic)
```

Binding-Time Analysis

binding-time analysis ‘by transformation’

```
BindingTime :
  |[ bo(~e1, ~e2) ]| -> |[ ~[bo(e1, e2)] ]|

BindingTime :
  |[ bo(e1, e2) ]| -> |[ <bo(e1', e2')> ]|
  where <map1(UnDynamic)> [e1, e2] => [e1', e2']

BindingTime :
  |[ if ~e1 then e2 else e3 ]| -> |[ ~[if e1 then e2' else e3'] ]|
  where <try(UnStatic)> e2 => e2'
      ; <try(UnStatic)> e3 => e3'

BindingTime :
  |[ if <e1> then e2 else e3 ]| -> |[ <if e1 then e2' else e3'> ]|
  where <try(UnDynamic)> e2 => e2'
      ; <try(UnDynamic)> e3 => e3'
```

Conclusions

- Separation of rules and strategy
  - develop transformation rules 'stand-alone'
  - strategy combines rules
  - mix of traversal specific for datatype and generic traversal
  - avoid mixing rules and strategy
- Stratego/XT tools can be used interactively!
  - basic interaction quite easy
- Future work
  - partial evaluator for imperative language (const propagation)
  - more sophisticated interaction with emacs
  - refactoring: open up whole program transformations