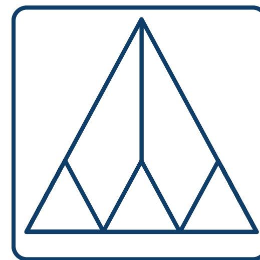
Extending Dynamic Rules

Context-Aware Rewriting Made Easy



Arthur van Dam

adam@cs.uu.nl

Institute of Information and Computing Sciences, University Utrecht, The Netherlands

Talk Overview

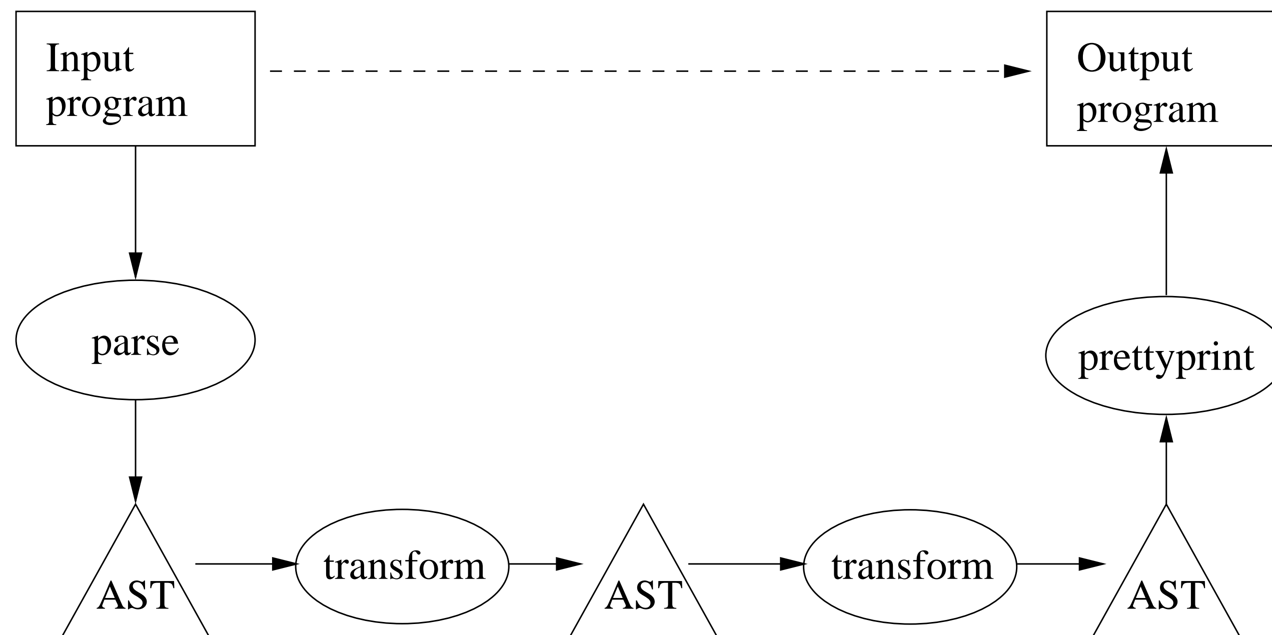
- Introduction into program transformation and the Stratego language
- Rewrite rules, and how they lack context
- Running example: constant propagation
  - Dynamic rule (re-/un-)defining
  - Dynamic rule scoping and labeling
  - Rule set forking and intersection
  - Fixpoint intersection
- Project summary
- Concluding remarks + Questions

Program Transformation

- Many application areas:
  - compilation
  - optimization
  - program synthesis
  - more...
- We focus on source-to-source optimizing transformations
- Automated transformation of input (-program) into new form
- What's important?
  - *representation*
  - local transformation specification (*rewrite rules*)
  - controlling the transformation process (*strategies*)

StrategoXT

- Generic framework for program transformation
- Input to Stratego transformations can be of any language (the *object language*).
- Traversal of trees, rewriting of terms



Program Representation

- ATerm format: memory efficient, term comparison in $O(1)$ by *maximal sharing*
- Tree structured input (compare with user-defined data types)

```
|[ fadd a b = a + b ]|  ≡  FunDef(
                                "fadd"
                              , [Var("a"), Var("b")]
                              , FunApp("(+)",
                                       [Var("a"), Var("b")]
                                      )
                              )
```

- Use concrete syntax: embed object language in meta language (Stratego)

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

// is equivalent to:

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

A Typical Stratego Module

- Static rewrite rules
- Strategies that select and apply rules, and traverse terms

```
module Tiger-FoldConst
imports Tiger lib
strategies
  main = bottomup(try(fold))

  fold = EvalBinOp + EvalRelOp

rules
  EvalBinOp : |[ e + 0 ]| -> |[ e ]|

  EvalBinOp : |[ i * j ]| -> |[ k ]|
              where <mul>(i, j) => k
  // ...
```

Rewrite Rules Are Unaware of Context

- Static rewrite rules are typically local operations
- input can be simply rewritten to output, without additional information
- For more sophisticated rewriting, information from elsewhere is needed

```
rules
  EvalBinOp : |[ x * y ]| -> ??
```

- Maybe x and/or y has a constant value, but the rewrite rule doesn't know that locally

Running example: Constant Propagation

(with constant folding and unreachable code elimination)

Constant Propagation 1 : Basic Blocks

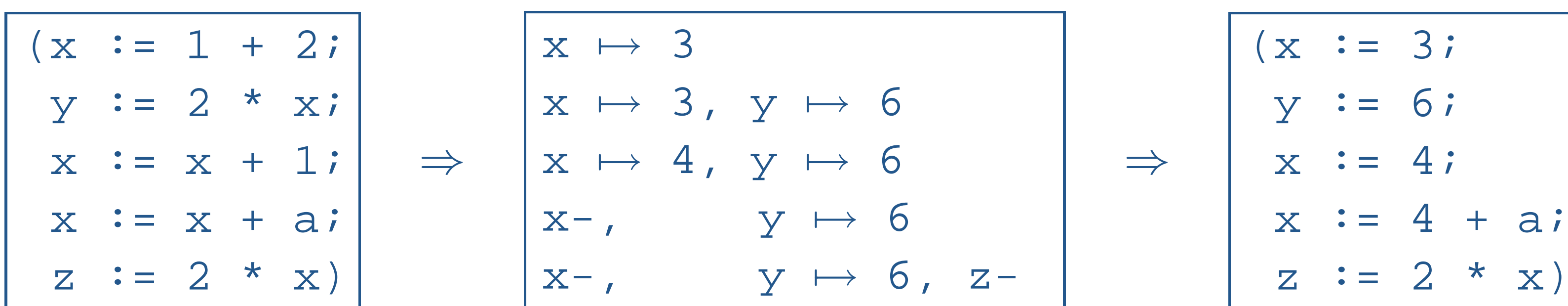
- Basic blocks: sequences of expressions and assignments:

```
(x := 1; y := (p & q) + a; x := x + y)
```

- If a variable is assigned a constant value, future uses of that variable can be replaced by that value
- Define a new rewrite rule for each constant-assignment encountered:

```
rules( PropConst : |[ x ]| -> |[ e ]| )
```

- For each variable use: check whether a constant value is known



Dynamic Rules : Defining and Undefining

- A normal Stratego module consists of strategies and *static* rewrite rules, all *known at compile time*.
- At runtime, more information on the current program becomes available
- Use this information in the *runtime definition of new rewrite rules*

```
rules( RuleName : p1 -> p2 where s )
```

- All parts (left- and right hand side $p1$,$p2$, and condition s) may contain variables that are bound in context (at definition time)
- Previous rules (if any) with same name and left hand side are hereby *redefined*
- To prevent further use of existing dynamic rules, they can be *undefined*:

```
rules( RuleName :- p1 )
```

Constant Propagation 1b : Basic Blocks

- For each assignment: inspect right hand side expression and register any constant values
- For each variable use: check whether a constant value is known

```
prop-const =
  PropConst
  <+ |[ <id> := <prop-const> ]|; AssignPropConst
  <+ all(prop-const); try(fold)

AssignPropConst =
  ?|[ x := e ]|
  ; if <is-value> e then
      rules( PropConst :  |[ x ]| -> |[ e ]| )
    else
      rules( PropConst :- |[ x ]| )
    end

is-value = Int(is-int)
```

Constant Propagation 2 : Local Variable Scopes

- Local variable scope: `let` blocks with variable declarations
- Dynamic rules (here: `PropConst`) are only valid within the scope of the variable they apply to
- Below: variables in bold are in declarations or assignments. Variables with same name and same color are semantically the same:

Program		Rules		Result
`x := 1+2;`		`x ↦ 3`		`x := 3;`
`y := 2*x;`		`x ↦ 3, y ↦ 6`		`y := 6;`
`let var x := x+1;`		`x ↦ 4, y ↦ 6`		`let var x := 4;`
`    var y := a`		`x ↦ 4, y-`		`    var y := a`
` in x := x+a;`		`x-,    y-`		` in x := 4+a;`
`    let z := 3`	⇒	`x-,    y-,    z ↦ 3`	⇒	`    let z:= 3`
`     in y := z`		`x-,    y ↦ 3, z ↦ 3`		`     in y := 3`
`    end;`		`x-,    y ↦ 3, z-`		`    end;`
`    x := y+z`		`x-,    y ↦ 3, z-`		`    x := 7+z`
`end;`		`x ↦ 3, y ↦ 6, z-`		`end;`
`z := x*y`		`x ↦ 3, y ↦ 6, z ↦ 18`		`z := 18`

Dynamic Rules : Scoping and Labeling

- Without scopes, dynamic rules are *globally available*
- Dynamic rule scopes can be defined per rule name, all generated rules within, with the same name, will be *retracted upon exit of the scope*
- A scoped dynamic rule definition *shadows* any previous rule definitions in surrounding scopes with the same name and left hand side

```
{| RuleName : s |}
```

- Scopes may be assigned additional labels for *refined scope association*
- Labels can be any term at runtime, assigned upon entry or in the middle of it

```
{| RuleName.p : s |}
rules( RuleName+p )
```

- Rules can be associated to a specific scope by *labeled rule definition*:

```
rules( RuleName.p : p1 -> p2 where s )
rules( RuleName+p : p1 -> p2 where s )
```

Constant Propagation 2b : Local Variable Scopes

- Before defaulting to bottomup behaviour, first check whether current term introduces new variable scope
- If so, introduce a dynamic rule scope for `PropConst`:

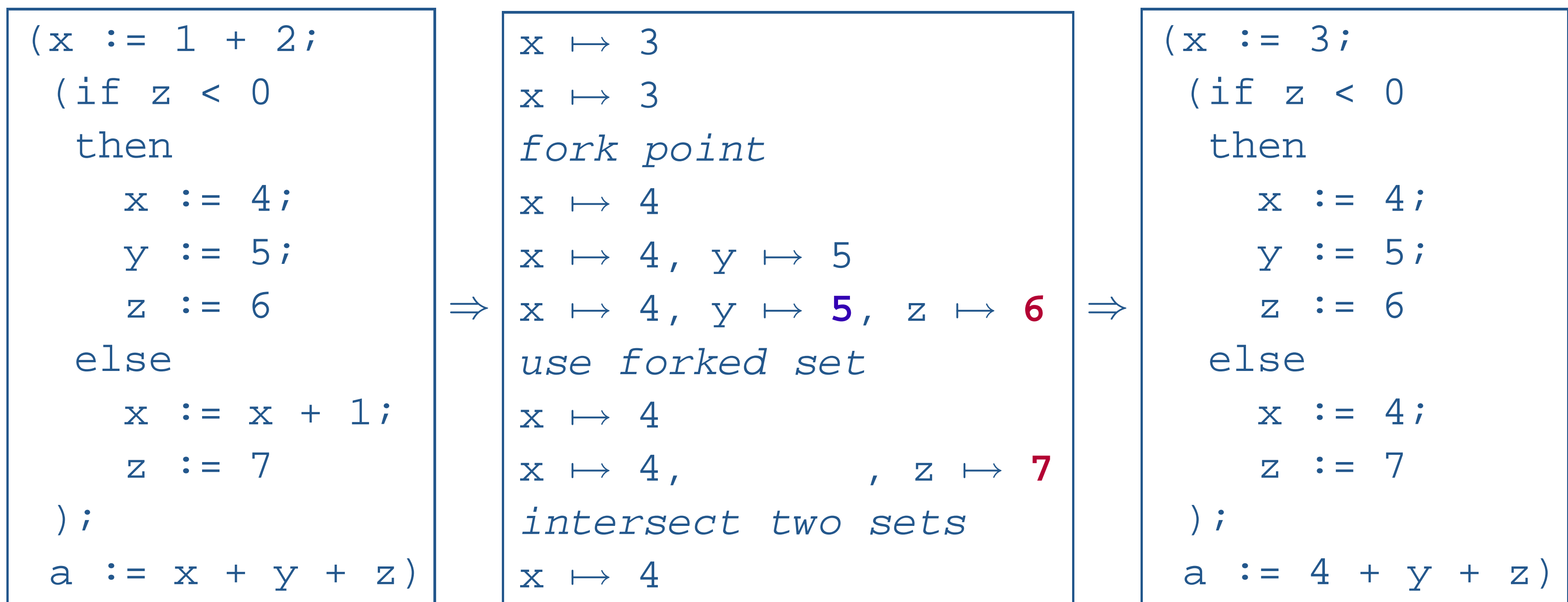
```
prop-const =
  PropConst
  <+ |[ <id> := <prop-const> ]|; AssignPropConst
  <+ |[ let <*id> in <*id> end ]|; {| PropConst : all(prop-const) |}
  <+ all(prop-const); try(DeclarePropConst <+ fold)

DeclarePropConst =
  ?|[ var x ta := e ]|
  ; if <is-value> e
    then rules( PropConst+x :  |[ x ]| -> |[ e ]| )
    else rules( PropConst+x :- |[ x ]| ) end

AssignPropConst =
  ?|[ x := e ]|
  ; if <is-value> e
    then rules( PropConst.x :  |[ x ]| -> |[ e ]| )
    else rules( PropConst.x :- |[ x ]| ) end
```

Constant Propagation 3 : Control Flow Statements

- Until now, there was always one single execution path
- Control flow statements influence the execution path (`if then else`, `while`)
- For data-flow optimizations, propagated analysis information should be *valid along all execution paths*
- Split (fork) the transformation, and intersect results afterwards



- (Other analyses may need *union* afterwards, instead of *intersection*)

Dynamic Rules : Forking and Meeting

- When control flow splits, dynamic rule sets should be split as well (*fork*)
- When the paths *meet* again, the resulting rule sets need to be combined (usually intersection or union)
- Efficient implementation by starting a fresh *change set* for each path at fork point, on top of shared existing ruleset
- All functionality is in API, convenience constructs in syntax:

```
s1 / RuleName \ s2    // Fork and intersection
```

```
s1 \ RuleName / s2    // Fork and union
```

- After meeting the two change sets, the *result is committed to the rule set* from just before the fork point

Constant Propagation 3b : Control Flow Statements

- Before defaulting to bottomup behaviour, first check whether current term is a control flow statement (`prop-control`)
- Control flow statements are first checked on reachability:

```
prop-const =
  PropConst
  <+ |[ <id> := <s> ]|; AssignPropConst
  <+ |[ let <*id> in <*id> end ]|; {| PropConst : all(s) |}
  <+ prop-control(prop-const)
  <+ all(prop-const); try(DeclarePropConst <+ fold)

prop-control(s) =
  |[ if <s> then <id> else <id> ]|
  ; (EvalIf; s
      <+  (|[ if <id> then <s> else <id> ]| /PropConst\
           |[ if <id> then <id> else <s> ]|))
```

Dynamic Rules : Fixpoint Meeting

- Conditionals have two paths that are traversed once
- Loop statements are traversed zero or more times
- Dynamic rules should be *valid for any number of iterations*
- Fork and meet until no more changes in rule set occur (*fixpoint*)

```
/ RuleName \* s    // Fixpoint intersection
```

```
\ RuleName /* s    // Fixpoint union
```

Constant Propagation 3c : Loop Statements

- `prop-const` strategy remains the same, but two new variants for `prop-control` are added
- Control flow statements are first checked on reachability:

```
prop-control(s) =
  |[ while <id> do <id> ]|
  ;(|[ while <s> do <id> ]|; EvalWhile
    <+ /PropConst\* |[ while <s> do <s> ]|)
```

One additional feature

and the
closing notes

Extend Rule Sets : Don't Redefine

- Upon normal generation of a dynamic rule, all *previous rules* with same name and left hand side in current scope are *redefined*
- Not always desirable, therefore: *extend the rule set*

```
rules( RuleName :+ p1 -> p2 where s )
```

- All rules with same name, left hand side and scope form one set
- Normal rule application applies the most recent one, or tries older ones until the first successful one

```
<RuleName> p0 => p1
```

- Get *all succesful rewritings* in inner scope with `bagof-`L:

```
<bagof-RuleName> t => [p1, p7, .., pm]
```

- Applications: function specialization, common subexpression elimination, higher-order matching

Project Summary

- From end-user of dynamic rules to co-developer of their redesign
- Three case studies led to new feature-requests and inspired entire redesign
  - lambda calculus with records : dead variable elimination, record-field selection, shrinking function elimination (algorithm by Andrew Appel)
  - Tiger imperative language : constant propagation
  - First-order functional language : deforestation / elimination of intermediate trees (Algorithm by Philip Wadler)
  - Side-track : A type-inferencer for TFOF
- Within StrategoXT: realize the dynamic rule system redesign
- Benchmark performance

Concluding remarks

- We saw dynamic rule *(un-/re-)defining, scoping, labeling, fork-and-meeting, and extending*
- Many existing application benefit from new dynamic rule system, new applications have also been implemented
- Dynamic rules are relatively ‘transparent’: no effort needed for dragging around context-information
- Future work includes generalization over rule generation and application, storing of true closures, and more
- Formal discussion of operational semantics in submitted paper (Fundamenta Informaticae, 2004), and MSc thesis
- Questions?

Fixpoint Intersection: Reaching Steady State

```
(v := 0; w := 0; x := 0; y := 0;
 z := 1;
 while (v = 0 & p) do (
   if (w = 1) then v := 1 else v := 0;
   if (x = 1) then w := 1 else w := 0;
   if (y = 1) then x := 1 else x := 0;
   if (z = 1) then y := 1 else y := 0
 ) ;
 printint(v + w + x + y + z))
```

#	v	w	x	y	z		v	w	x	y	z
i	0	0	0	0	1						
1	0	0	0	1	1	→	0	0	0	-	1
2	0	0	-	1	1	→	0	0	-	-	1
3	0	-	-	1	1	→	0	-	-	-	1
4	-	-	-	1	1	→	-	-	-	-	1
5	-	-	-	1	1	→	-	-	-	-	1