Transformations for Abstractions

Eelco Visser

Technical Report UU-CS-2005-034
Department of Information and Computing Sciences
Universiteit Utrecht

July 2005

This is an extended version of the following paper (the published paper does not have the appendices):
E. Visser. Tranformations for Abstractions. Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005). IEEE, Budapest, Hungary. September 2005. (Keynote paper)

ISSN 0924-3275

Address:
Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht

Eelco Visser
visser@acm.org
http://www.cs.uu.nl/~visser

Transformations for Abstractions

Eelco Visser
Department of Information and Computing Sciences, Universiteit Utrecht
PO Box 80.089, 3508 TB Utrecht, The Netherlands
http://www.cs.uu.nl/~visser, visser@acm.org

Abstract

The transformation language Stratego provides high-level abstractions for implementation of a wide range of transformations. Our aim is to integrate transformation in the software development process and make it available to programmers. This requires the transformations provided by the programming environment to be extensible. This paper presents a case study in the implementation of extensible programming environments using Stratego, by developing a small collection of language extensions and several typical transformations for these languages.

1. Introduction

Stratego/XT is a language and toolset for the construction of program transformation systems. The goal of the Stratego language is to unify different types of transformations into one language, i.e., to provide *abstractions for* implementing a wide range of *transformations*. The basic assumption underlying Stratego is that transformations should be expressed by rewrite rules that are under the control of a user-definable strategy—as opposed to rule-based systems with an implicit strategy.

Stratego/XT is applied in a number of research projects. First of all the Stratego compiler itself, the Stratego interpreter, and the XT toolset with numerous grammar engineering tools are implemented in Stratego. The CodeBoost transformation system for C++ [2] supports domain-specific optimization of numerical software. A compiler for Tiger includes all aspects of compilation, from type checking, via optimizations, to instruction selection. A compiler for Octave includes loop vectorization, and partial evaluation of types and values [19]. Finally, we are working on a transformation framework for Java geared towards domain-specific language embeddings and meta-programming [8, 6].

In these projects we[1] have implemented numerous types of transformations including desugaring of syntactic abstractions; assimilation of language embeddings [8]; bound variable renaming; optimizations, such as function inlining; data-flow transformations such as constant propagation, copy propagation, common-subexpression elimination, and partial evaluation [5, 20]; instruction selection [7]; and several analyses including type checking [6], escaping variables analysis, and bound-unbound variables analysis for Stratego. Thus, we have been successful in exploring the implementation of individual transformations, and the range of transformations that we know how to encode effectively and elegantly grows. Along the way we keep discovering better idioms and abstractions for implementing transformations.

What is common in the *transformations* that we are exploring is that they are *for* supporting *abstractions*.[2] That is, to enable programming at a higher-level of abstraction. Our aim is to integrate such transformations in the software development process and make them available to programmers. That is, create programming environments with full meta-programming support that let the (meta-) programmer extend the language, the compiler, and other aspects of the environment such as documentation generators. This requires the transformations provided by the programming environment to be extensible. The base programming language should be extensible with new constructs and/or with new (primitive) APIs implementing new abstractions. Likewise the compiler should be extensible with new transformations, and existing transformations should be extensible to new language constructs. New constructs may be implemented by reduction to the base language, or by extension of the back-end of the compiler. APIs may require domain-specific optimizations to ensure efficient execution. While syntactic extensibility is covered by the syntax definition formalism SDF [22], extensibility of transformations is a topic of research. The scale of the projects mentioned

[1] The Stratego/XT group

[2] Note that Stratego/XT is not intrinsically restricted to this application area. There are also applications of Stratego to software evolution. For example, the Proteus project at Lucent has built a transformation system for C/C++ programs [25] using Stratego.

above, prohibits studying these issues 'in vivo'.

This paper presents a case study in bringing together the essence of the techniques from the various application projects, in an 'in vitro' model of an extensible programming environment. The restricted scale makes experimentation with different implementation styles easy. The TFA project is an extensible framework consisting of a family of language extensions to a tiny core language. The rest of this paper presents the TFA languages, shows the implementation of some typical transformations, and discusses how extensibility for these transformations has been achieved. The paper may also be read as a tutorial with an overview of different styles for the implementation of transformations in Stratego. Since an in depth explanation of the Stratego concepts is out of the scope of this paper, the reader is referred to relevant papers.

2. Languages and Transformations

TFA is a collection of languages extending a tiny core language. Figure 1 defines the syntax of a number of these extensions. The *core* language introduces variables, assignments of values to variables, function and procedure calls, and blocks of statements. The *int* extension adds integer constants and the usual arithmetic, relational, and short circuit Boolean operators, where Boolean values are represented by integers as in C. Given this data type, the *control* extension adds control-flow constructs for choice and iteration. The *string* extension adds string literals. The *eblocks* extension adds expression blocks, allowing statements to be used in expressions; an expression `begin st1* return e; st2* end` first executes the statements `st1*`, then computes the result `e` of the expression, and finally executes the statements `st2*`. The *definitions* extension adds function and procedure definitions. The *regexps* extension embeds the *domain-specific language* of regular expressions in the language, introducing the match expression `/re/e`, which matches the string resulting from the evaluation of the expression `e` to the regular expression `re`. Not shown here are extensions adding *stack* operations and *labels and gotos*, to model low-level programs. There is no end to further possible extensions, of course.

In addition to the syntax, the TFA programming environment provides a collection of transformations that can be applied to programs. *Desugarings* reduce a program to use a smaller set of extensions. These types of transformations are also known as assimilation [8], normalization, simplification, or compilation (when the result is a low-level language). The *run* transformations reduce a program and its inputs to its outputs, and thus implement an interpreter. Bound-variable *renaming* is a transformation that assigns unique names to all declared identifiers in a program. *Data-flow* transformations optimize programs using knowledge

p	:=	b	
e	:=	$x \mid f$`(`e_1`, ...,`e_n`)`	
st	:=	`var` x `:` t`;` $\mid x$ `:=` e`;` $\mid f$`(`e_1`,...,`e_n`);` $\mid b$	
b	:=	`begin` sts `end`	
sts	:=	st^*	
t	:=	`void`	core
i	:=	$[0-9]+$	
e	:=	$i \mid e_1$ `+` $e_2 \mid e_1$ `*` $e_2 \mid ... \mid e_1$ `&` $e_2 \mid e_1 \mid e_2$	
t	:=	`int`	int
st	:=	`if` e `then` sts `else` sts `end`	
		`if` e `then` sts `end`	
		`while` e `do` sts `end`	
		`for` x `:=` e_1 `to` e_2 `do` sts `end`	control
str	:=	`"`$\neg[$`"`$]^*$`"`	
e	:=	str	
t	:=	`string`	string
e	:=	eb	
eb	:=	`begin` sts `return` e`;` sts `end`	eblocks
p	:=	def^*	
def	:=	`function` f`(`arg^*`)` `:` t eb	
		`procedure` f`(`arg^*`)` b	
arg	:=	x `:` t	definitions
re	:=	$str \mid$ `alpha` $\mid re_1 \mid re_2 \mid re_1 re_2 \mid re$`*` $\mid re$`+`	
e	:=	`/`re`/` $e \mid$ `/`re`;`f`/` x	
t	:=	`regexp`	regexps

Figure 1. Core language with extensions

	syn	desugar	run	ren.	d-flow	...
core	X		X	X	X	
int	L/S	X	A		E	
control	X	X	X	X	X	
string	L		A		E	
eblocks	S	R		X	X	
defs	X		X	X	X	
regexps	S	R				
stack		T	A			
gotos	X	T	X		X	
...						

X	=	general extension
L	=	literals added
S	=	syntactic sugar
R	=	extension is reduced to underlying base language
A	=	an API for the new data type has been added
E	=	evaluation rules (for the API) have been added
T	=	target of reduction for other extensions

Figure 2. Extensions and transformations

of the flow of data in the program. Examples are constant propagation and common-subexpression elimination. *Partial evaluation* or *specialization* specializes a program to its inputs. It can be seen as a special case of constant propagation. A *translation* transforms a program to a program in another language. These transformations can be combined into various complete transformation systems, such as a complete compiler, a source-to-source partial evaluator, an interpreter, and a source-to-source preprocessor (supporting a language extension or embedding).

The matrix in Figure 2 shows which languages extend which transformations. An implementation is not needed everywhere. The symbols in the matrix indicate the type of extension. For instance, the string extension only adds literals to the syntax, and requires an extension of the interpreter to implement an API for string manipulation. In the following sections we examine a selection of points in the matrix focusing on the following issues: how to concisely implement a transformation, and how to make these implementations extensible.

3. Desugaring

A simple but effective way to implement language extensions is by *desugaring*, that is, by reducing a syntactic abstraction (also known as syntactic sugar) to constructs in the base language. It depends on the complexity of the abstraction, how difficult the transformation is. Preferably, desugarings can be implemented by means of simple *rewrite rules*. Figure 3 shows desugaring rules for the int, control, and eblocks extensions.

A rewrite rule R : p_1 `->` p_2 defines a transformation from a program fragment matching p_1 to a program fragment p_2. The `BinOpToCall` rule generically rewrites an abstract syntax term representing a binary operator application to a function call. The `ForToWhile` rule defines `for` loops in terms of `while` loops. The `HoistEBlock` rules define how eblocks in expressions can be lifted to statement level. The auxiliary `collect-eblocks` strategy lifts all eblocks from a list of expressions at once, producing a list of statements consisting of the statements from the eblocks and a list of expressions without eblocks. These rules define single transformation steps. The `desugar` *strategy* combines these rules using the generic `innermost` strategy with `Desugar` as argument. Since there is nothing to desugar in the core language, `Desugar` is initially defined to be `fail`, the transformation that always fails. By simply extending the definition of `Desugar` in the extensions, the transformation is adapted to cover the extended languages. The `innermost` strategy exhaustively applies its argument transformation rules, starting with the innermost nodes of the abstract syntax tree of a program, and thus *normalizes* to a form in which none of the rules is applicable.

```
desugar = innermost(Desugar)
Desugar = fail                                          core
```

```
Desugar = BinOpToCall
BinOpToCall :
  f#([e1, e2]) -> |[ f(e1, e2) ]|
  where <is-bin-op> f                                   int
```

```
Desugar = ForToWhile <+ IfThenToIfElse
ForToWhile :
  |[ for x := e1 to e2 do st* end ]| ->
  |[ begin
       var x : int; var y : int;
       x := e1; y := e2;
       while x <= y do
         st* x := x + 1;
       end
     end ]|
  where new => y

IfThenToIfElse :
  |[ if e then st* end ]| ->
  |[ if e then st* else end ]|                          control
```

```
Desugar = HoistEBlockFromFunCall <+ ...
HoistEBlockFromFunCall :
  |[ f(e1*) ]| ->
  |[ begin st1* return f(e2*); end ]|
  where <collect-eblocks> e1* => (st1*,e2*)

HoistEBlockFromAssign :
  |[ x := begin st1* return e; end; ]| ->
  |[ begin st1* x := e; end ]|

HoistEBlockFromWhile :
  |[ while begin st1* return e; end
        do st2* end ]| ->
  |[ begin
       st1* while e do st2* st1* end
     end ]|                                             eblocks
```

Figure 3. Desugaring

4. Assimilation of Embedded Languages

The desugarings in the previous section were concerned with reducing new types of expressions and statements to old types of expressions and statements. Another kind of language extension is the embedding of a domain-specific language in the base language. The difference with the previous type of extension is that the embedded language is of a completely different type. Also it often involves nonlocal transformations, i.e., that affect the context of the construct. We have explored DSL embedding with Java as base language and described MetaBorg [8], a general approach for DSL embeddings. In that context, we have coined the term *assimilation* for the transformation that melds the embedding with its host code. Here we illustrate this type of

extension and transformation by embedding regular expression matching and its assimilation to expressions and new functions.

The expression `/re/e` matches the regular expression `re` against the string resulting from the evaluation of the expression `e`. To make assimilating this operation easier, the additional 'match with continuation' expression `/re;f/x` has been added to the extension. It succeeds (returns true) for the string in variable `x` if the regular expression `re` matches a prefix of `x` and the Boolean function `f` succeeds for the suffix. Using this construct we can give a compositional definition of regular expression matching by extending our desugaring transformation, that is by adding new rules to the definition of `Desugar`. Figure 4 shows (most of) the rules and Figure 5 shows the result of applying the transformation to an example program.

Rule `ReMch` transforms the match `/re/e` to `x := e; /re;isEmpty/x`, in order to first evaluate `e` and then check that `re` matches the entire string, i.e., with the empty string as suffix. The strategy `new` generates a unique new name. Rule `ReStr` transforms a string regular expression `str` to a check that the prefix of the string in `x` matches with `str`, and that the continuation function `f` succeeds on the suffix. Note that these rules use functions from the basic string API for analysing strings (`isPref`, `getSuf`, `isEmpty`). Rule `ReAlt` simply reduces the regular expression alternative to the Boolean or (`|`), where we need the fact that the `|` operator in our language is a short circuit operator. Also we need the fact that the subject of the match is a variable and not an arbitrary expression, since then we would risk duplicating computations, and possibly even side effects.

While these rules perform *local* transformations, the following rules have a local and a non-local effect. Rule `ReSeq` reduces the sequential composition `/re1 re2;f/x` to `/re1;g/x`, where the continuation `g` is a new function that matches `/re2;f/`. Similarly, rule `ReKle` generates a new function `g` for the Kleene star `/re*;f/`. This function checks whether the prefix of its argument string matches `re`, and then recursively uses `g` as continuation. When the match of `re` fails, the continuation `f` is called. Thus, the function `g` checks whether the prefix of the string matches `re` zero or more times.

These functions are created while rewriting some local expressions and clearly cannot be placed at that point, but should rather be added to the program at top-level; a so called 'local to global' transformation problem [26]. This problem is solved here by means of *dynamic rewrite rules*, rewrite rules that are defined *at run-time*, inheriting information from their definition context. The strategy `add-def` defines a dynamic rule `AddDef` that rewrites a program (a list of definitions) to the same program with the new definition added in front. Since this rule *extends* the existing `AddDef` rule, multiple function definitions can be created.

```
Desugar = ReMatch <+ ReStr <+ ReAlt <+ ReSeq
                  <+ ReKle <+ once-AddDef
ReMch : |[ /re/ e ]|
     -> |[ begin
             var x : string; x := e;
             return /re; isEmpty/ x;
           end ]|
  where new => x

ReStr : |[ /str;f/ x ]|
     -> |[ isPref(str,x) & f(getSuf(str,x)) ]|

ReAlt : |[ /re1|re2;f/ x ]|
     -> |[ /re1;f/x | /re2;f/x ]|

ReSeq : |[ /re1 re2;f/ x ]| -> |[ /re1;g/ x ]|
  where new => g
      ; add-def(|[
          function g(a : string) : int
          begin
            return /re2;f/ a;
          end ]|)

ReKle : |[ /re*;f/ x ]| -> |[ g(x) ]|
  where new => g
      ; add-def(|[
          function g(a : string) : int
          begin
            return /re;g/a | f(a);
          end ]|)

add-def(|def) =
  rules(
    AddDef :+ |[ def* ]| -> |[ def def* ]|
  )                                    regexps/desugar
```

Figure 4. Assimilating regular expressions

```
function match(x : string) : int
begin
  return / ("a" | "b")* "c" / x;
end
```

```
function c_0 (a : string) : int
begin
  return isPref("c", a) & isEmpty(getSuf("c", a));
end
function d_0 (a : string) : int
begin
  return (isPref("a", a) & d_0(getSuf("a", a)))
       | (isPref("b", a) & d_0(getSuf("b", a)))
       | c_0(a);
end
function match (x : string) : int
begin
  var b_0 : string; b_0 := x; return d_0(b_0);
end
```

Figure 5. Regular expressions assimilated

The new definition is added to the program at top-level by an invocation of `once-AddDef`, which produces one of the right-hand sides of the `AddDef` rule at a time, which is then consumed, i.e., will not be produced the next time `AddDef` or `once-AddDef` is invoked. Thus, all generated functions will be added to the program one by one as part of the rewriting process. The new function definitions generated by `ReSeq` and `ReKle` again contain invocations of a regular expression match. These will in turn be normalized after being added to the program, which may give rise to further functions being generated.

5. Renaming

Programs use names to denote entities such as run-time values, functions, and modules. The same identifier can be used to name multiple entities. Bound variable renaming is a transformation that replaces declared identifiers with a unique new name, such that it becomes straightforward to see which entity an occurrence refers to. Figure 6 defines an extensible bound variable renaming strategy with extensions for the control and eblocks languages.

What distinguishes renaming from the previous transformations, is that it is context-sensitive. It requires replacing the name of a variable declaration with a new name, and then consistently replacing all variable uses referring to that declaration with the same new name. Again dynamic rules come to the rescue. Consider the `rename-declaration` rule for renaming variable declarations. Given a declaration of a variable `x`, it uses the `new` strategy to generate a unique new name `y`, and then defines a new `RenameVar` rule, rewriting `x` to `y`. By applying the `RenameVar` rule to the right occurrences of `x`, the renaming can be propagated.

Another issue that plays a role here is that renaming touches only a few types of syntax tree nodes, i.e., variable declarations and variable uses. At the same time, we cannot use a general rewriting approach as in desugarings of the previous sections; we have to carefully control where the renaming rules are applied. The implementation in Figure 6 achieves this by means of a user-defined traversal strategy. The `rename` strategy first checks if some special case applies, otherwise performs a *generic traversal* to the direct subterms of the current term using the generic traversal combinator `all`. The `rename-special` strategy handles the special cases, that is, performing traversals for specific constructs, renaming declarations, or restricting the scope of the dynamic `RenameVar` rules. The latter ensures that these rules are only applied in those places where the renamed variable is in scope. The strategy is extended to the control and eblocks languages by extending the definition of `rename-special`. Only for constructs that introduce new variable bindings the `rename` strategy needs to be extended, since generic traversal takes care of other constructs.

```
rename = rename-special <+ all(rename)

rename-special =
  Var(RenameVar)
  <+ Stats({| RenameVar : map(rename) |})
  <+ |[ <id:RenameVar> := <rename>; ]|
  <+ rename-declaration

rename-declaration :
  |[ var x : t; ]| -> |[ var y : t; ]|
  where new => y
      ; rules( RenameVar+x : x -> y )          core
```

```
rename-special :
  |[ for x := e1 to e2 do st1* end ]| ->
  |[ for y := e3 to e4 do st2* end ]|
  where <rename> e1 => e3; <rename> e2 => e4
      ; new => y
      ; {| RenameVar
         : rules( RenameVar+x : x -> y )
         ; <rename> st1* => st2*
         |}                                    control
```

```
rename-special :
  |[ begin st1* return e1; st2* end ]| ->
  |[ begin st3* return e2; st4* end ]|
  where {| RenameVar
         : <rename> (st1*, e1, st2*)
                => (st3*, e2, st4*)
         |}                                    eblocks
```

Figure 6. Renaming bound variables

6. Evaluation

If part of a program is *constant*, i.e., does not depend on any unknown values, it can be transformed to the result of the computation it defines. This is clearly the case in the implementation of an interpreter, but is also useful in the definition of optimizations such as constant propagation and partial evaluation. This section defines a full interpreter, while Sections 7 and 8 focus on data-flow transformation, and function specialization, respectively.

Figure 7 defines evaluation rules that reduce the application of a function with constant arguments to a constant. Thus, these rules define the semantics of the functions of the primitive datatypes. In the next section we see that these rules can be used in optimizations as well. In Figure 8 the rules are used in the definition of interpreters for the core, control, and definitions extensions. The definition of the `eval` strategy follows the same pattern as that for renaming: the `eval-special` strategy deals with special cases, while the default case is to first evaluate the direct subterms, and then apply an evaluation rule added to the definition of `eval-exp`. The special cases deal with variable definition and control-flow constructs. The evaluation of function calls is done using another extension of `eval-exp`.

```
EvalAdd : |[ Add(i, j) ]| -> |[ k ]|
    where <addS>(i,j) => k

EvalMul : |[ Mul(i, j) ]| -> |[ k ]|
    where <mulS>(i,j) => k                          int
```

```
EvalIf  : |[ if i then st1* else st2* end ]|
       -> |[ begin st1* end ]| where <not-zero>i

EvalIf  : |[ if 0 then st1* else st2* end ]|
       -> |[ begin st2* end ]|                      control
```

```
EvalStrapp : |[ strapp(str1, str2) ]|
          -> |[ str3 ]|
where <conc-strings>(str1, str2) => str3

EvalStrlen : |[ strlen(str) ]| -> |[ k ]|
where <string-length> str => k                      string
```

Figure 7. Evaluation rules

The interpreter uses the dynamic `EvalVar` rule to propagate the values of variables from their definitions to their uses. Thus, when encountering an assignment, first the right-hand side expression is evaluated, and then an `EvalVar` rule is defined rewriting an occurrence of the left-hand side variable to its value. When encountering a variable declaration, the `EvalVar` rule for the variable with that name is *undefined*, to prevent that the value assigned to a variable with the same name in an enclosing scope is used in the local scope. When encountering a list of statements in a block (Stats), dynamic rule scope is used to limit the scope of `EvalVar` rules for variables declared in that block.

The `if` control construct is evaluated by first evaluating the condition, and then using the `EvalIf` rules to reduce the construct to one of its branches, which is then further evaluated. The `while` construct is evaluated by unrolling the loop to an `if` statement with an occurrence of the original loop after the body of the loop.

The interpretation of the definitions extension concerns the evaluation of function calls defined by function definitions, rather than being built-in to the interpreter (as is the case for the integer and string functions in Figure 7). To achieve this, a dynamic `EvalFunCall` rule is defined for each function definition in the program. This rule evaluates a function call by creating `EvalVar` rules binding the arguments of the function to its formal parameters, and then evaluating the body of the function resulting in a value `val`.

7. Data-Flow Transformations

Programs can often be simplified and optimized, in particular after desugaring. In imperative languages, optimizations often require knowledge of the data-flow in a program, that is, the relation between uses of variables and their definitions. This is similar to the situation in evaluation, where values of variables are propagated from definitions to uses. In fact, evaluation can be considered a special case of data-flow transformation, where control-flow always reduces to a single branch, and all assignments produce constant values. The generalization of evaluation is *constant propagation*, which propagates constant values when possible, but leaves

```
eval = eval-special <+ all(eval); try(eval-exp)

eval-special =
  EvalVar <+ eval-stats
  <+ eval-assign <+ eval-declaration

eval-assign =
  |[ x := <eval => e> ]|
  ; rules( EvalVar.x : |[ x ]| -> |[ e ]| )

eval-declaration =
  ?|[ var x : t; ]|
  ; rules( EvalVar+x :- |[ x ]| )

eval-stats = Stats({| EvalVar : map(eval) |})
eval-exp = fail                                     core
```

```
eval-special = eval-or <+ eval-and
eval-exp = EvalAdd <+ EvalMul <+ ...                int
```

```
eval-special = eval-if <+ eval-while; eval

eval-if =
  |[ if <eval> then <*:id> else <*:id> end ]|
  ; EvalIf; eval-stat

eval-while :
  st@|[ while e do st* end ]| ->
  |[ if e then st* st else end ]|                   control
```

```
eval-exp = EvalStrapp <+ ...                        string
```

```
eval-program =
  |[ <def*:map(register-def)> ]|
  ; !|[ main(); ]|
  ; eval
register-def =
  ?|[ function f(x*) : t eb ]|
  ; rules(
      EvalFunCall :
        |[ f(e*) ]| -> val
        where {| EvalVar
               : <zip(InitArg)> (x*, e*)
               ; <eval> eb => val
               |}
    )
InitArg =
  ?(|[ x : t ]|, e)
  ; rules( EvalVar+x : |[ x ]| -> e )
eval-exp = EvalFunCall                              definitions
```

Figure 8. Interpretation

programs intact otherwise.

Figure 9 generalizes the interpreter of the previous section to the definition of constant propagation for the core, int, control, and string languages. The organization of the strategy is the same as in the case of evaluation, but now catering for the fact that not all expressions and statements can be fully evaluated. Thus, when the right-hand side of an assignment is not a constant, no constant value can and should be propagated for the left-hand side variable `x`. Also no previous `PropConst` rule for `x` should remain accessible. Therefore, the `propconst-assign` strategy checks whether the result of propagation in the right-hand side expression is a constant value. If not, the `PropConst` rule is *undefined* for `x`.

Similarly, when the condition of an `if` statement does not reduce to a constant, the statement cannot be reduced to one of its branches. Therefore, propagation should proceed in both branches where these transformations should not affect each other. That is, the rule set for `PropConst` should be cloned so that the propagation in each branch can pretend to be the only branch executing. Afterwards, propagation should proceed with those `PropConst` rules that are consistent in both branches. This is achieved using the `/PropConst\` intersection operator. In the case of a loop, propagation should be repeated until a stable rule set has been reached, which is achieved by the `/PropConst\*` operator. This abstract interpretation style of constant propagation, combined with the unreachable code elimination achieved by the `EvalIf` rules, is more powerful than a separate analysis and transformation phase, as is illustrated in Figure 10. The fact that the assignment of `x` in the loop body can be ignored comes only after reducing the `if` statement when assuming that `x` is `10`.

Other data-flow transformations such as copy propagation, common-subexpression elimination, forward substitution, and dead code elimination can be defined in a similar manner. Since these transformations propagate expressions with variables (not just constant values), simply redefining or undefining the dynamic rule for a variable does not work any more. Using dependent dynamic rules [20], the dependencies of a rule on multiple entities such as variables can be declared.

8. Function Specialization

Extending constant propagation to programs with functions and procedures comes down to partial evaluation, i.e., specialization of programs to statically known inputs or to constants in the program. Partial evaluation supports abstraction by allowing programmers to write general programs that are instantiated for many different specific problems [16]. Here we consider one aspect of partial evaluation, namely function specialization. For each function call

```
propconst =
  propconst-special
  <+ all(propconst); try(propconst-eval)

propconst-special =
  PropConst <+ propconst-stats
  <+ propconst-decl <+ propconst-assign

propconst-stats =
  Stats({| PropConst : map(propconst) |})

propconst-decl =
  ?|[ var x : t; ]|
  ; rules( PropConst+x :- |[ x ]| )

propconst-assign =
  |[ x := <propconst => e>; ]|
  ; if <is-value> e then
      rules(PropConst.x : |[ x ]| -> e )
    else
      rules(PropConst.x :- |[ x ]|)
    end                                          core

propconst-eval = EvalExp                         int

propconst-special =
  propconst-if <+ propconst-while

propconst-if =
  If(propconst,id,id)
  ; (EvalIf; propconst
     <+ If(id,propconst,id)
          /PropConst\ If(id,id,propconst))
propconst-while =
  ?While(_, _)
  ; (/PropConst\*
       While(propconst, propconst))              control

propconst-eval = EvalExp                         string
```

Figure 9. Constant propagation

```
begin
  var x : int;
  var y : int;
  x := 10;
  while readInt() do
    if x = 10 then
      y := y + 1;
    else
      x := x + 1;
    end
  end
  writeInt(x, y);
end
```

```
begin
  var x : int;
  var y : int;
  x := 10;
  while readInt() do

    y := y + 1;



  end
  writeInt(10, y);
end
```

Figure 10. Constant propagation applied

a new function is generated that is specialized to the constant valued arguments of the call. For example, the partially constant call to the `power` function in Figure 12 is

```
declare-def =
  ?|[ function f(x1*) : t begin st1* return e1; end ]|
  ; rules(
      SpecializeCall :
        |[ f(e1*) ]| -> |[ g(e2*) ]|
        where <split-static-dynamic-args> (x1*, e1*) => (st2*, (x2*, e2*))
            ; new => g
            ; <propconst>|[ begin st2* st1* return e1; end ]| => eb2
            ; rules(
                Specialization :+ |[ function f(x1*) : t eb1 ]| -> |[ function g(x2*) : t eb2 ]|
              )
    )
propconst-eval = SpecializeCall
propconst-definitions =
  Definitions(
    map(declare-def); where(<SpecializeCall> |[ main(); ]| => st)
    ; ![ |[ procedure main() begin st end ]| | <mapconcat(bagof-Specialization)> ]
  )                                                                    definitions/propconst
```

Figure 11. Function specialization

replaced by a call to a new function `b_0`, which implements the `power` function specialized to the constant argument `3`. This specialization gives rise to an invocation of `power` with constant argument 2, which is specialized to a call the function `d_0`, and so on. Note that the result can be further improved by 'transition compression', i.e., function unfolding. Furthermore, memoization of specializations should avoid multiple function definitions for the same constant arguments. For brevity, these aspects are ignored here.

The program in Figure 11 extends the definition of constant propagation of Section 7 to function specialization. (Since the definition is mostly the same for procedures, only the specialization of functions is shown.) Given the definition of a function `f`, `declare-def` declares a dynamic rule `SpecializeCall`. This dynamic rule rewrites a call `f(e1*)` to a call `g(e2*)`, with `e2*` the non-static arguments of the original call, and the function `g` a specialization of `f` to the static (constant) arguments of the call. The definition of the new function `g` is created in the condition of the `SpecializeCall` rule. The strategy `split-static-dynamic-args` separates the static from the dynamic arguments. The static arguments are translated into a list of statements `st2*` that assign these constant values to the corresponding formal parameters of `f`. Using these statements the body of `f` is instantiated. It would be unsafe to substitute the occurrences of the formal parameters by their values, since the variables may be re-assigned during execution of the body. Constant propagation on the resulting expression block with the invocation of `propconst` takes care of propagating the constant assignments into the body of the function. Of course, any function calls within the body will give rise to further function call specializations. The result is a specialized function body `eb2`. The dynamic rule `Specialization` is extended to rewrite the original definition of `f` to the new definition for `g` which has the remaining non-static arguments `x2*` as parameters and the specialized e-block `eb2` as body.

Specialization of the definition in a program now proceeds as follows (`propconst-definitions`). For each function or procedure definition, a `SpecializeCall` rule is defined using `declare-def`. The specialization of the call to `main()` then ensures that all calls reachable from `main()` are specialized. An application of `bagof-Specialization`, then rewrites each function definition to the list of its specializations. A new definition for `main()` calls the result of specializing the original `main` function.

```
function power(x : int, n : int) : int
begin
  var power : int;
  if n = 0 then power := 1;
  else if (n % 2) = 0 then power := power(x * x, n/2);
  else power := x * power(x, n - 1);
  end end
  return power;
end
procedure main()
begin writeInt(power(readInt(), 3)); end
```

```
procedure main () begin a_0(); end

procedure a_0() begin writeInt(b_0(readInt())); end

function b_0 (x : int) : int
begin return Mul(x, c_0(x)); end

function c_0 (x : int) : int
begin return d_0(Mul(x, x)); end

function d_0 (x : int) : int
begin return Mul(x, 1); end
```

Figure 12. Power function specialized

9. Discussion

Related Work The idea of integrating transformation in the software development process is becoming mainstream, as suggested by recent proposals of approaches such as software factories [15], model-driven software engineering (MDSE) [3], and language oriented-programming (LOP) [10]. All these approaches have in common the development of domain-specific languages (DSLs) for capturing abstractions in a problem domain and the use of transformations to implement DSL programs. However, software factories and MDSE seem to be in the more traditional setting of separate DSLs and software generators, while LOP aims at extending a language with domain abstractions with assimilations for reducing these to the base language. In this respect, the approach is similar to macro systems such as Dylan [21] and metamorphic macros [4] that allow a programmer to define new abstractions within a program. However, macro systems allow only limited syntactic extensions, only support local to local transformations, and most macro systems do not allow the macro programmer to analyse the environment, consider typing information, or transform the arguments in arbitrary ways.

Another related approach is the modular definition of languages [11], in which language features are defined as individual components and composed in different combinations to form complete programming languages. This may become a useful approach to DSL implementation, although the research has mostly concentrated on the specification of the semantics of programming language features. Also the approach only covers the semantics of language features, not their transformations. Independent composability is a desirable features as it allows extensions to be provided as plugins. In the TFA languages and transformations in this paper, the extensions are not independently composable since knowledge about the base language is assumed.

Extensible compilers such as the Polyglot [18] compiler for Java are aimed at language designers and compiler implementers to support the implementation of new language features. The `abc` [1] compiler is an extension of Polyglot to an aspect compiler. An important point in the design of PolyGlot is scalable extensibility; the effort of creating an extension should be proportional to the number of abstract syntax tree nodes affected by the extension. In the Stratego setting this property is achieved by relying on generic traversals, which require only to define non-standard behaviour. One of the problems with existing extensible compilers is that they lack the right abstractions for program transformation and compilation, such as pattern matching, abstractions for scoping, generic traversals, and are therefore more difficult to extend.

Attribute grammars provide a declarative formalism for specifying analyses and translations for abstract syntax trees. Attribute grammars modularize well into either separate rules for a specific attribute, or into rules for a language construct. Attribute grammars with *forwarding* [27] allow the specification of replacement nodes to compute attribute values where the forwarding node provides none. Thus, new constructs can be defined in terms of old ones, in a similar way to desugarings. JastAdd [12, 13] extends attribute grammars with rewrite rules for normalizing tree nodes, which has a similar function as forwarding with the difference that nodes are replaced by their normal form. An issue with this set up is that the application of rewrite rules cannot be controlled.

Conclusion This paper presented a case study of language extensibility using Stratego/XT, showing that Stratego supports various styles of transformation: basic term rewriting applying local transformations; term rewriting with local-to-global transformations to add function definitions to the top-level; global-to-local transformations propagating context information in renaming, evaluation, and constant propagation; the merging of context information in constant propagation; and a combination of global-to-local with local-to-global transformation (specialization) where one dynamic rule defines another.

The use of extending strategy definitions to extend transformations works well. The implementations of the various transformations are neatly modularized per extension. The main issue with this feature is that the order of evaluation of extensions is undefined. This was intentional to encourage definition of mutually exclusive rules, but forces an extra level of indirection (e.g., the `propconst-special` hook) when a default case should be specialized. It would be preferable to add a case to the main strategy, e.g., `propconst = eval-while`, since that would also allow unanticipated extensibility. Realizing this requires more explicit control over the order of evaluation of extensions, similar to overriding in TXL [9]. Even better would be an invasive approach to extending definitions, which allows unanticipated extension at the exact location where needed.

Another issue is the fact that extensions are currently integrated based on the Stratego sources, rather than compiled components. Binary deployable extensions would be preferable since that would allow extensions to be distributed as plugins. This also requires some way to check for interferences between extensions.

While further experimentation with other language extensions and with other transformations both in the realm of standard language features and with domain-specific language embeddings is needed to test the approach sketched in this paper, it appears to be a promissing path to realize transformations for abstractions.

Availability Stratego/XT is available from `http://www.stratego-language.org`. The sources of the TFA project are available for experimentation from `http://www.stratego-language.org/Stratego/TransformationsForAbstractions`

Acknowledgements I would like to thank the SCAM 2005 program chairs, Jens Krinke and Giulio Antoniol, for inviting me as keynote speaker. The TFA project grew out of the TIL language designed together with Jim Cordy at Dagstuhl Seminar 05161 on Transformation Techniques in Software Engineering. I would like to thank Martin Bravenboer and Karl Trygve Kalleberg for numerous discussions on program transformation and extensibility, and for their help with this paper. This work is part of NWO/JACQUARD project 638.001.201 TraCE: Transparent Configuration Environments.

A Stratego

The design of the Stratego language follows a similar pattern as the small imperative languages in Part I. A small core language is extended to a full-fledged transformation language, providing expressive abstractions for implementing transformations. In practice the design of the language has evolved over time, and the implementation does not follow the clean separation of concerns exhibited by the TFA framework. However, one of the goals of that project is to recast the implementation of Stratego in order to make it into an easily extensible language. In this part we briefly discuss the elements of the language following the syntax definition in Figure 13. A more thorough definition of the language, including an operational semantics can be found in [5].

B Matching and Building

Stratego programs transform terms, which are isomorphic to trees, and are convenient for representing abstract syntax trees produced by parsers. For example, the statement `if 0 then else a := d; end` is represented by the term `If(Int("0"),[],[Assign("a",Var("d"))])`. The basic concept of Stratego is that of a transformation *strategy*, which is a function that transforms a term into another term, or *fails* at doing so. Strategies were introduced to control the application of rewrite rules. One of the key insights in the first version of Stratego [24] was that rewrite rules are not primitive operations, but that these are composed from more atomic operations, namely matching (`?`p) and building (`!`p) term patterns. For instance, the first `EvalIf` rule from Figure 7 consist of the operations `?If(Int("0"),st1*,st2*)` and `!Block(st2*)`, using the abstract syntax representation of the program fragments. A strategy is implicitly applied to a term, called the *subject* or *current* term. The effect of matching a pattern against the subject term is either failure when there is no match, or a binding of the pattern variables to the corresponding subterms. The effect of a build `!`p is the replacement of the subject term with the instantiation of the pattern p. Once a variable is bound by a match operation, it cannot be rebound to another term. To reuse variables multiple times, their scope should be limited. The pattern variable scope construct `{`x_1`,...,`x_n`:`s`}` limits the scope of pattern variables x_i to the strategy s.

C Strategy Composition

Other basic strategies are the *identity* strategy `id`, which always succeeds; and the *failure* strategy `fail`, which always fails. These basic operations can be combined into compound transformation strategies using a number of strategy *combinators*. The sequential composition (s_1`;`s_2), first applies s_1 to the subject term and then s_2 to the result. For example, the strategy expression

```
{st1*,st2*:?If(Int("0"),st1*,st2*);!Block(st2*)}
```

implements the `EvalIf` rewrite rule from Figure 7, by first matching the left-hand side of the rule, and then building

p	::=	str | i | r | x | $c(p^*_,)$ | `(`$p^*_,$`)` | `[`$p^*_,$`|`p`]` | `[`$p^*_,$`]`	
s	::=	`?`p | `!`p | `{`x^*`:`s`}`	
x	::=	variable	
c	::=	constructor	match/build
s	::=	`id` | `fail` | s_1`;`s_2 | s_1`<+`s_2	
		| s_1 `<` s_2 `+` s_3 | `if` s_1 `then` s_2 `else` s_3 `end`	
		| `where(`s`)` | `<`s`>`p | s `=>` p	composition
P	::=	$d_1 ... d_n$	
d	::=	$dsig$ `=` s	
$dsig$	::=	f`(`$sd^*_,$`|`$vd^*_,$`)` | f`(`$sd^*_,$`)` | f	
sd	::=	f | f`:`tp	
vd	::=	x | x`:`tp	
s	::=	`let` d^* `in` s `end` | f`(`$s^*_,$`|`$p^*_,$`)` | f`(`$s^*_,$`)` | f	
f	::=	strategy operator	definitions
d	::=	$dsig$ `:` p_1 `->` p_2 (`where` s)?	rules
s	::=	$c(s^*_,)$ | $tr(s)$	
tr	::=	`all` | `one`	traversals
s	::=	`rules(`drd_1 ... drd_n`)` | `{|`f_1`,...,`f_n`:`s`|}`	
		| s_1`/`$f^*_,$`\`s_2 | s_1`\`$f^*_,$`/`s_2 | `/`$f^*_,$`\*`s | `\`$f^*_,$`/*`s	
drd	::=	$drsig$ `:` p_1 `->` p_2 (`where` s)?	
		| $drsig$ `:+` p_1 `->` p_2 (`where` s)?	
		| $drsig$ `:` p | $drsig$ `:-` p | f`+`p	
$drsig$	::=	sig | sig`.`p | sig`+`p	dynamic rules

Figure 13. Syntax of Stratego

the right-hand side. The sequential composition is typically also used to chain a number of transformations as the

```
If(eval, id, id); EvalIf; eval
```

composition from Figure 8.

Choice operators allow choosing between transformations. The core choice operator is the *guarded left choice* s_1<s_2+s_3, which first applies s_1 and if that succeeds s_2, otherwise s_3. The other choice operators are defined in terms of guarded choice. The *left choice* s_1<+s_3 is equivalent to s_1<id+s_3. This is the most common choice combinator and is typically used in compositions such as

```
ReMatch <+ ReStr <+ ReAlt <+ ReSeq <+ ReKle
```

to try a number of rules one after the other.

The conditional choice `if` s_1 `then` s_2 `else` s_3 `end` is equivalent to `where(`s_1`) <` s_2 `+` s_3. Here, the `where(`s`)` combinator is an abstraction for {x`:?`x`;`s`;!`x`}`, i.e., it saves the current term by matching it against the variable x, then applies the strategy s, and finally *restores* the original term bound to x. Finally, the strategy *application* <s>p is equivalent to `!`p`;`s, and the apply-match strategy s `=>` p is equivalent to s`;?`p. These combinators are typically used in the condition of a rewrite rule. For instance, the evaluation rules in Figure 7 use conditions like `<addS>(i,j) => k` to apply a strategy `addS` to the pair `(i,j)` and match the result against the variable `k`.

D Strategy Definitions

Strategy definitions of the form $dsig$=s can be used to give names to frequently used strategy expressions. For example, the following definition names the example strategy above:

```
EvalIf = {st1*,st2*: ?If(Int("0"),st1*,st2*)
                   ; !Block(st2*)}
```

There are no global pattern variables in Stratego. Therefore, the scope of pattern variables in a top-level definition is restricted to that definition. Thus, the definition

```
EvalIf = ?If(Int("0"),st1*,st2*); !Block(st2*)
```

is equivalent to the one with an explicit scope above.

A strategy operator may have a list of strategy, and a list of term arguments, which are separated by a `|`. Typical examples of parameterized strategies are the following:

```
try(s)    = s <+ id
repeat(s) = try(s; repeat(s))
```

When a strategy operator has no strategy or term arguments, it can be written without parentheses, e.g., `EvalIf`, instead of `EvalIf(|)`. Strategy definitions may be recursive, as in the definition of `rename` from Figure 6:

```
rename = rename-special <+ all(rename)
```

An uncommon feature of definitions in Stratego is that there may be multiple definitions with the same name, which do not have to be textually adjacent. Definitions extending an existing definition may even be given in a separate module. This is the key feature we have used in Part I to cater for extensibility. For example, to allow for the addition of new desugaring rules to the desugaring transformation the `Desugar` strategy is introduced as an indirection in the definition of `desugar`. Each extension then added new rules to its definition:

```
Desugar = fail                                  core
Desugar = BinOpToCall                           int
Desugar = ForToWhile <+ IfThenToIfElse      control
```

The semantics of definition extension is the choice between the bodies of the definitions. Thus, the definition of `Desugar` in control is equivalent to

```
Desugar = fail <+ BinOpToCall <+ ForToWhile
          <+ IfThenToIfElse
```

Since `fail` is a unit for the choice operator it is safe to use this as the initial definition. There is one caveat with this extension mechanism. In the current design of Stratego the order in which the bodies are combined is not defined. Thus, such compositions are only safe if the definitions are independent, i.e., do not succeed for the same terms. This issue should be addressed to get better control over extensions of transformations.

E Rewrite Rules

A rewrite rule is an abstraction for a specific combination of a match and a build. Thus, the `EvalIf` definition from the previous section can be written as a rewrite rule:

```
EvalIf : If(Int("0"),st1*,st2*) -> Block(st2*)
```

Such rules are desugared to definitions with a match of the left-hand side, followed by a build of the right-hand side. Similarly, a conditional rewrite rule

```
EvalAdd : Call("Add",[Int(i),Int(j)]) -> Int(k)
          where <addS> (i, j) => k
```

is syntactic sugar for a definition

```
EvalAdd = ?Call("Add", [Int(i), Int(j)])
          ; where( <addS> (i, j) => k )
          ; !Int(k)
```

F Concrete Syntax

Specifying transformation rules using the abstract syntax of a language, as in previous sections of Part II, can become tedious to write and hard to read. By embedding the syntax of the object language (TFA) in the syntax of the meta language (Stratego) we can use the concrete syntax of the object language for specifying term patterns. For example, the `EvalAdd` rewrite rule written in abstract syntax above can be rendered with concrete syntax as:

```
EvalAdd : |[ Add(i, j) ]| -> |[ k ]|
          where <addS> (i, j) => k
```

The delimiters `|[...]|` are used to distinghuish concrete syntax fragments from normal Stratego fragments. An additional feature of this embedding is the declaration of *meta-variable schemas*. For example, identifiers `i` `j`, and `k` are used as meta-variables of type integer constant. In the TFA Stratego programs we have used the convention that the non-terminal names in the grammar in Figure 1, are used as meta-variables in concrete syntax fragments. A general approach for creating concrete syntax embeddings for arbitrary context-free object and meta languages is described in [23, 14, 6].

G Traversal Strategies

The strategy combinators discussed above allow the sequential composition of two transformations and the choice between two transformations. These transformations will apply at the root of the subject tree. However, for most transformations it is necessary to apply transformations to subtrees. For this purpose, Stratego provides traversal combinators that descend one level into a tree [17, 24]. These can be combined with the other combinators to define full tree traversals. There are two flavours of traversal combinators, congruence operators and generic traversal operators. For each abstract syntax tree constructor c with arity n there is a corresponding *congruence operator* $c(s_1,\ldots,s_n)$, which can be defined using the following rewrite rule schema:

$$
\begin{array}{l}
c(s_1,\ldots,s_n)\ : \\
\quad c(t_1,\ldots,t_n)\ \texttt{->}\ c(t'_1,\ldots,t'_n) \\
\quad \texttt{where}\ \texttt{<}s_1\texttt{>}\ t_1\ \texttt{=>}\ t'_1\texttt{;}\ \ldots\texttt{;}\ \texttt{<}s_n\texttt{>}\ t_n\ \texttt{=>}\ t'_n
\end{array}
$$

That is, the strategy argument are applied to the corresponding term arguments. For example, the strategy expression `IfElse(propconst, id, id)`, applies the `propconst` strategy to the first argument of an `IfElse` term, and the `id` strategy to the second and third argument. In Part I concrete syntax versions of congruence operators were used, using the `|[...]|` delimiters to distinghuish a TFA fragment and <s> to apply a strategy to one of the arguments. For example, the concrete syntax congruence `|[ x := <eval,=> e> ]|` is equivalent to `Assign(?x, eval => e)`.

While congruence operations define traversal through a specific constructor, we are often not interested in the constructor, but just want to traverse the tree in a mostly uniform manner. For this purpose Stratego provides *generic traversal combinators*. The combinator `all(`s`)` applies the transformation s to each direct subterm of the subject term. The combinator `one(`s`)` applies s to one direct subterm. These combinators can be used to define various full traversal strategies. For example, the following definitions

```
bottomup(s) = all(bottomup(s)); s
topdown(s)  = s; all(topdown(s))
```

define a bottom-up, or post-order traversal in which the s transformation is applied to each node after visiting its direct subterms, and a top-down, or pre-order traversal in which s is applied to a node before visiting its subterms. These one-pass traversals can be used to define normalization strategies, such as the following `innermost` strategy:

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

This strategy is used in Figure 3 to implement the extensible desugaring component of TFA.

In the programs in Part I we have often used the `all` combinator directly in strategies such as

```
eval = eval-special <+ all(eval); try(eval-exp)
```

from Figure 8. This corresponds to a `bottomup` strategy with an escape to allow alternative traversals for specific constructors through `eval-special`.

H Dynamic Rules

Many of the transformations we encountered in Part I are context-senstive, that is, rules need non-local information to perform a transformation. This is a problem in term rewriting since term rewrite rules are context-free, i.e., have only access to the term matched by the left-hand side of the rule. Dynamic rules [5, 20] solve this problem by defining rules at run-time at the place where the context information is available, and then propagating the rule to the place where the information is needed. For example, consider the definition of the dynamic `EvalVar` rule in the following definition of `eval-assign`:

```
eval-assign =
  ?Assign(x, Int(i))
  ; rules( EvalVar : Var(x) -> Int(i) )
```

The dynamic rule is defined after the match `?Assign(x,Int(i))` has succeed and the variables `x` and `i` bound. The new `EvalVar` rule inherits these bindings, and only succeeds when applied to a `Var` term with the same identifier.

When defining a rule, while some other rule with the same left-hand already existed, the older rule is replaced. However, when in *extend* mode (indicated with `:+`), a new rule is added and the dynamic rule can rewrite to multiple right-hand sides. We saw examples of this in regular expression assimilation and function specialization. A rule can also be explicitly *undefined* (indicated with `:-`). We saw examples of rule undefinition in the implementation of constant propagation in Figure 9. The scope of a dynamic rule can be controlled using the scope construct `{|`R`:` s`|}`; any rules added to the current scope while applying s are removed afterwards. Using scope labels one can control the scope in which a rule can be defined. Finally, dynamic rule sets can be forked afterwards pointwise intersected or unified, as we saw in the definition of constant propagation for control-flow constructs.

Dynamic rules are an extension to the base Stratego language. The compiler front-end translates dynamic rule definitions in terms of hashtable insertions and lookups.

References

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[2] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[3] J. Bézivin. On the unification power of models. *Software and Systems Modelling*, 4(2):171 – 188, May 2005.

[4] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'02)*, pages 31–40. ACM Press, 2002.

[5] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 2005. (To appear).

[6] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallin, Estonia, September 2005. Springer. (To appear).

[7] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

[8] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[9] J. Cordy. TXL - a language for programming language tools and applications. In *Proceedings of the 4th International Workshop on Language Descriptions, Tools and Applications*, volume 110 of *Electronic Notes in Theorectical Computer Science*, pages 3–31, April 2004.

[10] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains 'onBoard*, November 2004. `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[11] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

[12] T. Ekman. Separation of concerns in compiler construction using jastadd ii. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 2004.

[13] T. Ekman and G. Hedin. Reusable language specification modules in jastadd ii. In *Proceedings of the Workshop on Evolution and Reuse of Language Specifications for DSLs*, June 2004.

[14] B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 239–253. Spinger-Verlag, 2004.

[15] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[16] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[17] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[18] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, April 2003.

[19] K. Olmos and E. Visser. Turning dynamic typing into static typing by program specialization. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages

141–150, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[20] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, April 2005.

[21] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.

[22] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[23] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[24] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[25] D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Science, April 2005.

[26] J. van Wijngaarden and E. Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University., May 2003.

[27] E. van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.