A High-Level IR Transformation System

Herbert Jordan, Peter Thoman, and Thomas Fahringer

University of Innsbruck, Institute of Computer Science
Technikerstrasse 21a, 6020 Innsbruck, Austria
{herbert,petert,tf}@dps.uibk.ac.at

Abstract. The development and implementation of source-to-source code transformations based on high-level intermediate representations (IRs) is a labor-intensive, error-prone task. It tends to result in large code bases which are hard to read and maintain. Although not encountered within widely used source-to-source compiler infrastructures, transformation systems tackle this issue by utilizing declarative descriptions of unification-based term rewriting rules. However, unification lacks the ability of (arbitrarily) deep term inspection and is limited to tree nodes exhibiting a fixed arity – a property not generally satisfied by high-level IRs or programming languages.
In this paper we present a declarative tree transformation system which exceeds the expressiveness of conventional unification based systems by operating on arbitrary tree structures. We define its formal foundation, describe its implementation within the Insieme compiler project and provide real-world code examples. The latter demonstrate the expressive power, usability and intuitiveness of our solution.

Keywords: Compiler, Program Transformation, Term Rewriting, Pattern Matching, Regular Expressions

1 Introduction

Within contemporary compiler infrastructures, source-to-source transformations are typically implemented by a (long) hand-coded sequence of operations inspecting and manipulating some AST-like internal representation (IR). This approach is not only labor-intensive and error-prone, thus limiting productivity, but even more crucially it also reduces maintainability due to its tendency to result in obscure code [10].

Although rarely encountered within widely utilized compiler infrastructures, transformation systems tackle this issue by offering a declarative, rule based interface for the definition of transformations [2, 8, 11]. Fundamentally, each rule consists of a pattern and a replacement template. Any term matching the pattern is replaced by an instantiation of the template. In general, the pattern matching is based on unification, resulting in two restrictions. On the one hand, unification can only impose limited constraints on the matched term. For instance, it can not check whether within an arbitrary nesting level of a matched term a given subterm occurs. On the other hand, unification builds upon an algebraic structure,

hence tree nodes are restricted to a fixed arity [2, 11]. Yet, within AST-like IRs, constructs which do not naturally exhibit a fixed arity (e.g. compound statements or argument lists) are omnipresent. Frequently this issue is circumvented by representing variable-sized lists using artificial function symbols (or grammar rules) linking individual elements to recursively composed lists. However, not only does this approach conceal the structure of the internal representation, it also increases the complexity of defining patterns and rewrite rules.

Within this paper we present a novel combination of unification-based term rewriting rules and regular expressions allowing the declarative description of complex transformations for arbitrary tree structures – in particular high-level compiler IRs.

2 Design Goals

This section provides an informal overview of the intended capabilities of our pattern matching and rewriting system by discussing a list of example use cases. A simple case, which is already not directly supported by unification based approaches, is to check whether a given variable $v1$ is present within some code fragment. We would like to write something similar to

$$aT(v1) \tag{1}$$

where the construct aT denotes *"anywhere in the tree"*. A more extended case would be the requirement to check whether the expression $exp1$ is present as a full expression within a given compound statement. In this case we would like to write something similar to

$$\{\_^*, exp1, \_^*\} \tag{2}$$

where $\_$ denotes a wildcard and * the Kleene operator (any number of repetitions, including none).

In many cases patterns will be defined not only to constrain the structure of some input but also to extract information. For instance, it might be necessary to obtain the variable being declared by some declaration statement. In this case we would like to use a pattern involving a variable similar to

$$decl(\$x) \tag{3}$$

to obtain the requested information. Matched against the input declaration

$$int\ a\ =\ 5 \tag{4}$$

pattern (3) should yield the variable mapping $\{x \mapsto a\}$. Furthermore, in a case where all variables declared within a compound statement should be obtained, the pattern

$$\{(\neg decl(\_))^*, (decl(\$x), (\neg decl(\_))^*)^*\} \tag{5}$$

matched against

$$\{int\ a\ =\ 5;\ f(a);\ bool\ b\ =\ true;\ int\ c\ =\ 7;\} \tag{6}$$

should yield $\{x \mapsto [a, b, c]\}$. It should also be possible to constrain the values pattern variables are bound to. For instance, if the selection should be limited

to declarations of integer variables we would like to use a pattern similar to

$$\{(\neg decl(var(int, \_))^*, (decl(\$x : var(int, \_)), (\neg decl(var(int, \_))^*)^*\} \quad (7)$$

Here, $var(t, n)$ is a pattern construct matching an IR variable n of type t and the construct $\$x : y$ defines a pattern variable x matching every structure satisfying the pattern y. Applying this pattern to fragment (6) should result in $\{x \mapsto [a, c]\}$.

In some cases we want more. For instance, we might need a pattern identifying variables being declared but never used. This should be covered by

$$\{decl(\$x), (\neg aT(\$x))^*\} \quad (8)$$

Hence, a declaration of some IR variable captured by the pattern variable $\$x$ followed by a sequence of statements not including this particular variable. Note the difference to the previous examples. In pattern (5) the pattern variable $\$x$ is bound to a list of sub-trees, once for each declaration in the list, while in the current example $\$x$ should be bound only once to the variable being declared at the beginning of the compound statement. In the subsequent repetition $(\neg aT(\$x))^*$ the variable is supposed to be the fixed to the value previously bound to $\$x$.

From this observation we derived the following desirable rule for variable bindings: within every iteration of a repeating sub-pattern p, variables may be re-bound to new values, unless already bound before entering p the first time. If they were already bound, then, within all repetitions, previously bound variables remain bound to the value determined before entering p.

Of coarse, pattern (8) is limited to situations where the unused variable is declared by the first statement in a compound. This restriction is lifted by using

$$aT(decl(\$x)) \wedge \{(\neg aT(\neg decl(\$x) \wedge \_(\_^*, \$x, \_^*)))^*\} \quad (9)$$

where $\wedge$ is the conjunction of patterns and $\_(\_^*, p, \_^*)$ matches any node with a child matching the pattern p. The pattern searches for any sub-tree declaring a variable $\$x$ which is never referenced outside the declaration.

As a final challenge for the pattern syntax we would like to define a pattern capable of listing all for loops within a perfectly nested loop nest. The problem here is that the loop nest might be arbitrarily deep. The Kleene operator is limited to horizontal matching, along the list of children of a single node. For this class of use cases an operator defining a recursively nested tree structure is required. We define an additional primitive $rT.x(p)$ which is equivalent to p with the additional effect of binding the pattern p to the recursive variable x. Within p the term $rec.x$ can be used as a placeholder for the full pattern p. Based on this operators we can define a pattern for a for-loop nest using

$$rT.x(\$l : forStmt(\neg forStmt(\_) \vee rec.x)) \quad (10)$$

where $forStmt(b)$ matches any for-loop with a body matching the pattern b. The variable $\$l$ will be bound to a list of all loops of the matched loop nest.

Finally, patterns should provide a means to match the input for transformation rules. For instance, the rule

$$\{\$xs, \{\}, \$ys\} \rightarrow \{\$xs, \$ys\} \quad (11)$$

is designed to match any compound statement which includes an empty compound statement and to eliminate this inner statement. On the right hand side

of the rule the replacement is specified by a template utilizing the variables of the left hand side. Also, unlike all previous examples, in this case the pattern variables $\$xs$ and $\$ys$ match lists of trees instead of individual trees.

3 Method

Our approach is divided into two layers – the *core primitives*, defining its expressive power and a variety of *derived constructs* created by composing core primitives to provide more user-friendly, domain-specific connectors. This separation enables essential algorithms including the pattern matching to focus on a minimal set of constructs while the developers utilizing our system can define patterns and rules using constructs customized for their specific domain, e.g. an IR. Furthermore, potential modifications on the target structure during the course of its development can be reflected within the layer of derived constructs, thereby effectively adjusting all patterns built on top of those.

Within this sections the formal foundation and the core primitives of our patterns and replacements are specified while derived constructs are covered within the following two sections.

3.1 Tree Structure

Before defining a grammar for patterns and replacements a definition of the structures to be operated on has to be provided.

Let $\mathbb{A}$ be a set of atomic values (e.g. the union of integers, names and booleans) and $\mathbb{K}$ be a set of the node types to be distinguished. Any tree generated by the production

$$T ::= a \mid k(T^*)$$

where $a \in \mathbb{A}$ and $k \in \mathbb{K}$ is a valid input for our infrastructure.

This definition is generic enough to cover arbitrary trees. In particular it does not depend on a fixed arity for any node type $k \in \mathbb{K}$. For the remainder of this paper, let $\mathbb{T}$ be the set of all trees generated by the production rule given above for suitable sets $\mathbb{A}$ and $\mathbb{K}$.

3.2 Patterns

Within our framework we distinguish two kinds of patterns – tree and list patterns. While tree patterns describe the structure of individual trees, list patterns cover the composition of lists of trees (=forests).

Let $\mathbb{V}$ be a set of variable identifiers. Tree patterns ϕ and list patterns ψ are generated by the following pair of mutually recursive production rules

$$\phi ::= \_ \mid t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \_(\psi) \mid k(\psi) \mid x : \phi \mid aT(\phi) \mid rT.x(\phi) \mid rec.x$$
$$\psi ::= \epsilon \mid \phi \mid \psi, \psi \mid \psi \vee \psi \mid x : \psi \mid \psi^*$$

where $t \in \mathbb{T}$, $k \in \mathbb{K}$, $x \in \mathbb{V}$ and ϵ is the empty list. Furthermore, let Φ be the set of all possible tree patterns and Ψ be the set of all list patterns.

Semantics We define the semantics of patterns by defining the sets $T_\phi \subseteq \mathbb{T}$ and $T_\psi \subseteq \mathbb{T}^*$, matched by a tree pattern ϕ and a list pattern ψ respectively.

Let $t, m, n, r \vdash \phi$ denote the fact that the tree $t \in \mathbb{T}$ matches the tree pattern ϕ, and $s, m, n, r \vdash \psi$ denote the fact that the sequence $s \in \mathbb{T}^*$ matches the list pattern ψ, based on the tree variable mapping $m : \mathbb{V} \to \mathbb{T}$, the list variable mapping $n : \mathbb{V} \to \mathbb{T}^*$, and the recursive context map $r : \mathbb{V} \to \Phi \times (\mathbb{V} \to \mathbb{T}) \times (\mathbb{V} \to \mathbb{T}^*)$. The following rules provide a inductive definition of the relation $\vdash$ based on the structure of a tree pattern ϕ

$t, m, n, r \vdash \_$	iff	$true$	(wildcard)
$t, m, n, r \vdash t$	iff	$t = t$	(constant)
$t, m, n, r \vdash \neg\phi$	iff	not $t, m', n', r \vdash \phi$ and $m \subseteq m'$ and $n \subseteq n'$	(negation)
$t, m, n, r \vdash \phi_1 \wedge \phi_2$	iff	$t, m', n', r \vdash \phi_1$ and $t, m, n, r \vdash \phi_2$ and $m' \subseteq m$ and $n' \subseteq n$	(and)
$t, m, n, r \vdash \phi_1 \vee \phi_2$	iff	$t, m, n, r \vdash \phi_1$ or $t, m, n, r \vdash \phi_2$	(or)
$t, m, n, r \vdash \_(\psi)$	iff	$t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(any node)
$t, m, n, r \vdash k(\psi)$	iff	$t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(node)
$t, m, n, r \vdash x : \phi$	iff	$t, m \setminus \{x\}, n, r \vdash \phi$ and $(x \mapsto t) \in m$	(var)
$t, m, n, r \vdash aT(\phi)$	iff	t' subtree of t and $t', m, n, r \vdash \phi$	(any tree)
$t, m, n, r \vdash rT.x(\phi)$	iff	$t, m, n, \{x \mapsto (\phi, m, n)\} \oplus r \vdash \phi$	(recursion)
$t, m, n, r \vdash rec.x$	iff	$x \mapsto (\phi, m', n') \in r$ and $t, m', n', r \vdash \phi$	(rec. end)

and a list pattern ψ

$s, m, n, r \vdash \epsilon$	iff	$s = \epsilon$	(empty)
$s, m, n, r \vdash \phi$	iff	$s = [t]$ and $t, m, n, r \vdash \phi$	(single)
$s, m, n, r \vdash \psi_1, \psi_2$	iff	$s = s_1, s_2$ and $s_1, m, n, r \vdash \psi_1$ and $s_2, m, n, r \vdash \psi_2$	(sequence)
$s, m, n, r \vdash \psi_1 \vee \psi_2$	iff	$s, m, n, r \vdash \psi_1$ or $s, m, n, r \vdash \psi_2$	(or)
$s, m, n, r \vdash x : \psi$	iff	$s, m, n \setminus \{x\}, r \vdash \psi$ and $(x \mapsto s) \in n$	(var)
$s, m, n, r \vdash \psi^*$	iff	$s = \epsilon$ or $s = s_1, s_2$ and $s_1, m_1, n_1, r \vdash \psi$ and $s_2, m_2, n_2, r \vdash \psi*$ and $m \subseteq m_1$ and $m \subseteq m_2$ and $n \subseteq n_1$ and $n \subseteq n_2$	(repetition)

where all free variables are existentially quantified, s_1, s_2 denotes the concatenation of two sequences s_1 and s_2, and $A \oplus B$ denotes the mapping obtained by overriding and extending B by the relations in A. A tree $t \in \mathbb{T}$ is an element of T_ϕ if there are mappings m and n such that $t, m, n, \emptyset \vdash \phi$ holds. Correspondingly, a sequence $s \in \mathbb{T}^*$ is an element of T_ψ if there are mappings m and n such that $s, m, n, \emptyset \vdash \psi$ holds.

3.3 Matches

Let $\mathbb{U}_0 = \mathbb{T} \uplus \{\bot\}$ where $\bot$ is the value assigned to unbound variables. Let $\mathbb{U}_{n+1} = \mathbb{U}_n^*$ for all $n \in \mathbb{N}$. The set $\mathbb{U}$ of potential values assigned to variables is given by

$$\mathbb{U} = \bigcup_{0 \leq i} \mathbb{U}_i$$

Furthermore, let $d_\phi : \mathbb{V} \to \mathbb{N}_0$ be the function assigning every variable $x \in \mathbb{V}$ the repetition-depth of its leftmost, outermost occurrence within a pattern ϕ. Hence, whenever encountering a repetition (ψ^*) or a recursion ($rT.x(\phi)$) along the path from the root of the parse tree of pattern ϕ to the first occurrence of x, the depth is increased by one.

A valid variable match $m : \mathbb{V} \to \mathbb{U}$ of a pattern ϕ assigns each variable $x \in \mathbb{V}$ within ϕ an element of $\mathbb{U}_{d_\phi(x)}$ if x is a tree variable and an element of $\mathbb{U}_{d_\phi(x)+1}$ if x is a list or recursion variable. Let $\mathbb{M}$ denote the set of all variable matches. A matching algorithm needs to verify whether a given tree t matches a pattern ϕ and compute a variable match $m \in \mathbb{M}$ recording the necessary instantiations of the involved variables.

3.4 Replacements

The replacement expression is a representation of a script turning a matched tree $t_m \in \mathbb{T}$ and a match result $m \in \mathbb{M}$ into a new tree substituting t_m. For the definition of replacement expressions we distinguish three types of generator expressions: expressions computing results of type $\mathbb{T}$ (tree generators, τ), results of type $\mathbb{T}^*$ (list generators, σ), and results of type $\mathbb{U}$ (value generators, υ). Their structure is defined by the three production rules

$$\begin{aligned}
\tau &::= \upsilon \mid k\,(\sigma) \mid \tau\,[\tau/\tau] \\
\sigma &::= \upsilon \mid \epsilon \mid [\tau] \mid \sigma, \sigma \\
\upsilon &::= \lambda_c \mid \lambda_t\,(\upsilon) \mid \tau \mid \sigma \mid \mathit{let}\ x = \upsilon\ \mathit{in}\ \upsilon \mid \forall x \in \upsilon\ .\ \upsilon
\end{aligned}$$

where $k \in \mathbb{K}$ is a node type, $\lambda_c : \mathbb{T} \times \mathbb{M} \to \mathbb{U}$ is a function creating values, $\lambda_t : \mathbb{U} \to \mathbb{U}$ is a function transforming values and $x \in \mathbb{V}$ is a variable. Let Δ, Σ and Υ denote the set of expressions being generated by τ, σ and υ respectively.

The semantics of our generator expressions are given by the functions $\Gamma_\Delta : \mathbb{T} \times \mathbb{M} \times \Delta \to \mathbb{T}$, $\Gamma_\Sigma : \mathbb{T} \times \mathbb{M} \times \Sigma \to \mathbb{T}^*$ and $\Gamma_\Upsilon : \mathbb{T} \times \mathbb{M} \times \Upsilon \to \mathbb{U}$ defined by

$$\Gamma_\Delta\,(t, m, \tau) = \begin{cases}
\Gamma_\Upsilon(t, m, \upsilon) & \text{if } \tau \text{ is } \upsilon \text{ and } \Gamma_\Upsilon(t, m, \upsilon) \in \mathbb{T} \quad (\mathit{expr}) \\
k\,(\Gamma_\Sigma(t, m, \sigma)) & \text{if } \tau \text{ is } k\,(\sigma) \quad (\mathit{node}) \\
\mathit{let}\ t_1 = \Gamma_\Delta\,(t, m, \tau_1)\ \mathit{in} & \\
\mathit{let}\ t_2 = \Gamma_\Delta\,(t, m, \tau_2)\ \mathit{in} & \\
\mathit{let}\ t_3 = \Gamma_\Delta\,(t, m, \tau_3)\ \mathit{in} & \\
t_1[t_2/t_3] & \text{if } \tau \text{ is } \tau_1\,[\tau_2/\tau_3] \quad (\mathit{substitution})
\end{cases}$$

$$\Gamma_\Sigma\,(t, m, \sigma) = \begin{cases}
\Gamma_\Upsilon(t, m, \upsilon) & \text{if } \sigma \text{ is } \upsilon \text{ and } \Gamma_\Upsilon(t, m, \upsilon) \in \mathbb{T}^* \quad (\mathit{expr}) \\
\epsilon & \text{if } \sigma \text{ is } \epsilon \quad (\mathit{empty}) \\
[\Gamma_\Delta\,(t, m, \tau)] & \text{if } \sigma \text{ is } [\tau] \quad (\mathit{single}) \\
\Gamma_\Sigma\,(t, m, \sigma_1)\,, \Gamma_\Sigma\,(t, m, \sigma_2) & \text{if } \sigma \text{ is } \sigma_1, \sigma_2 \quad (\mathit{sequence})
\end{cases}$$

$$\Gamma_\Upsilon\,(t, m, \upsilon) = \begin{cases}
\lambda_c\,(t, m) & \text{if } \upsilon \text{ is } \lambda_c \quad (\mathit{ctor}) \\
\lambda_t\,(\Gamma_\Upsilon(t, m, \upsilon')) & \text{if } \upsilon \text{ is } \lambda_t\,(\upsilon') \quad (\mathit{transform}) \\
\Gamma_\Delta\,(t, m, \tau) & \text{if } \upsilon \text{ is } \tau \quad (\mathit{tree}) \\
\Gamma_\Sigma\,(t, m, \sigma) & \text{if } \upsilon \text{ is } \sigma \quad (\mathit{list}) \\
\Gamma_\Upsilon\,(t, \{x \mapsto \Gamma_\Upsilon(t, m, \upsilon_1)\} \oplus m, \upsilon_2) & \text{if } \upsilon \text{ is } \mathit{let}\ x = \upsilon_1\ \mathit{in}\ \upsilon_2 \quad (\mathit{let}) \\
\mathit{let}\ [u_1, \ldots, u_n] = \Gamma_\Upsilon\,(t, m, \upsilon_1)\ \mathit{in} & \\
\mathit{let}\ f = \lambda y.\Gamma_\Upsilon\,(t, \{x \mapsto y\} \oplus m, \upsilon_2) & \\
\mathit{in}\ [f\,(u_1), \ldots, f\,(u_n)] & \text{if } \upsilon \text{ is } \forall x \in \upsilon_1\ .\ \upsilon_2 \quad (\mathit{foreach})
\end{cases}$$

where $t_1[t_2/t_3] \in \mathbb{T}$ denotes the tree obtained by replacing all occurrences of $t_3 \in \mathbb{T}$ within $t_1 \in \mathbb{T}$ by $t_2 \in \mathbb{T}$.

3.5 Bringing it all together: Rules

Finally, a *rule* is defined by a pair $\phi \rightarrow \tau$ where $\phi \in \Phi$ is a tree pattern and $\tau \in \Delta$ is a tree generator expression as defined above. When applying a rule on a tree $t \in \mathbb{T}$ it is determined whether there are mappings m and n such that $t, m, n, \emptyset \vdash \phi$. If so, the match result $m' \in \mathbb{M}$ is computed and utilized to generate the replacement $\Gamma_\Delta(t, m', \tau) \in \mathbb{T}$.

4 Implementation

Our approach has been implemented in the Insieme compiler and runtime project [3]. The Insieme project aims to establish a unified platform for conducting research and developing tools in the field of parallel programming languages. It supports OpenMP, OpenCL, MPI and Cilk as well as extensions and hybrids of those. The foundation of the compiler component is laid by a unified, high-level intermediate representation exhibiting unique properties [4]. Crucially, its structure is constrained to be a tree, not including any back or cross edges. This trait turns the Insieme IR into a convenient domain for our system. Yet, our pattern concept is independent of the actual tree structure. Adapters for handling alternative IRs including GCC's GENERIC [5] as well as the Clang [1] or the Rose AST [7] may be implemented.

Our implementation is based on C++11, and represents patterns and replacements as objects, with common connectors mapped to overloaded C++ operators. This approach has multiple advantages: *composability* (simple pattern composition using variables, operators and functions), *extensibility* (new user-defined constructs may be added), *reliability* (pattern fragments can be tested independently using common unit testing frameworks, and their types and arities are checked at compile time), and *productivity* (all integrated development environment features including code completion apply to patterns, and users are not required to learn a new language syntax). The examples in Section 5 illustrate some of these advantages.

Due to overloading, multiple variations of the same pattern construct can be offered, which is particularly useful when defining derived constructs dealing with IR primitives. For instance, there might be two overloaded functions

```
TreePattern forStmt() { ... /* some k(..) primitive */ ... }
TreePattern forStmt(const TreePattern& body) { ... }
```

where the first creates a pattern matching any for loop while the latter allows the user to constrain the body of any matched loop. Based on those, the statements

```
auto noFor = !forStmt();
auto p = forStmt(noFor) | forStmt(forStmt(noFor));
```

create a pattern p matching a single for loop or two perfectly nested loops.

The Matching Algorithm The centerpiece of our system is the pattern matching algorithm. In our implementation it is based on a back-tracking approach following the rules presented in Section 3.2. While for most primitives the check whether a given tree matches the corresponding pattern is straightforward (e.g. the constant t, wildcards, negations and conjunctions) the processing of a few primitives requires more sophisticated steps. For instance, to determine whether a forest s satisfies a pattern ψ_1, ψ_2 the sequence s needs to be split up into two sub-sequences s_1 and s_2. However, the splitting point can only be guessed at this point – and in case it was wrong altered in a back-tracking step.

To increase the probability of guessing right as soon as possible we employ several pruning heuristics. For instance, if a sequence pattern of the shape $\psi_1, \psi_2, \ldots, \psi_n$ contains constant tree patterns ($\psi_i = t$) or patterns demanding a fixed node type (e.g. $\psi_i = k(\ldots)$) these elements are identified within a potential candidate list $t_1, \ldots, t_m$ before the remaining, potentially more complex patterns are matched against the interjacent sub-sequences. Also, memoization is utilized to avoid resolving identical sub-problems multiple times.

Nevertheless, the worst case complexity of our matching algorithm is exponential. However, so far, for real-world patterns encountered within our daily interaction with the system the search space pruning heuristics are effective enough to not impose a substantial performance issue. Also, the system benefits from the limited average length of sequences encountered within ASTs.

5 Examples

In general, when defining sophisticated patterns the definition of auxiliary connectors is beneficial. Therefore, in addition to the constants any (_), anyList (_*), the primitive connector node(...) matching any node with a given child list and the derived connector step(a), which is equivalent to node(*any << a << *any) where the << operator is the sequence connector of list patterns, we have defined the following constructs

```
inline TreePattern all(const TreePattern& a) {
  return rT((a & node(*rec)) | (!a & node(*rec)));
}
inline TreePattern outermost(const TreePattern& a) {
  return rT(a | (!a & node(*rec)));
}
inline TreePattern innermost(const TreePattern& a) {
  return rT((!step(aT(a)) & a) | node(*rec));
}
```

The derived constructs all, outermost and innermost collect all / the outermost / the innermost sub-trees matching a given input pattern. Based on those, patterns locating loops at corresponding code positions can be constructed by

```
auto a = all(var("x", forStmt()));
auto b = outermost(var("x", forStmt()));
auto c = innermost(var("x", forStmt()));
```

Also, a pattern identifying a used variable and all its accesses is created by

```
auto x = var("x");
auto use = !declStmt() & step(x);
auto p = aT(declStmt(x)) & aT(use) & all(var("y",use));
```

The resulting pattern p checks for the presence of a declaration defining a utilized variable x and collects all its uses within the pattern variable y.

A Real-World Transformation To demonstrate the applicability of our system to more complex tasks we present a transformation designed to eliminate redundant sync calls within Cilk applications, which was used in a recent publication [9]. A sync is deemed redundant if there has not been any spawn invocation since the last sync.

Let spawn and sync be constants representing patterns matching applications of the corresponding operators. In a first step we define the pattern

```
auto unsynced = rT(spawn | node(*any << aT(rec) << *!sync));
```

matching code fragments potentially resulting in an unsynchronized state. This is the case for a plain spawn statement or whenever an arbitrarily nested spawn is not followed by a sync on the same or an enclosing scope. In the next step the predicate unsynced is utilized to define the predicate synced as well as the desired pattern p identifying redundant sync statements:

```
auto synced = !unsynced;
auto p = compound(
  opt(*any << sync) << *synced << var("x", sync) << *any
);
```

The pattern p searches code fragments in which an optional (opt(..)) safe sequence of explicitly (on the same scope) or implicitly (nested) synchronized statements is followed by a sync call, which gets bound to variable x. The replacement expression

```
auto r = substitute(root, var("x"), noop);
```

can be utilized to generate a proper substitute for any matched statement. Here root is the root of the matched sub-tree, var("x") extracts the value bound to variable x and noop is a constant representing an expression conducting no operation. It replaces the identified redundant sync call x by a noop. Finally, the following code fragment combines p and r into a (transformation) rule and applies it to a target-code fragment in:

```
Rule syncElimination = Rule(p,r);
auto out = syncElimination(in);
```

The application of the rule includes the computation of a match result for pattern p and forwards it to the replacement expression r to produce the desired result.

6 Related Work

To the best of our knowledge there is no comparable transformation support for high-level source-to-source manipulation frameworks for the IRs of GCC [5],

Clang [1] and Rose [7]. In those popular systems, transformations have to be encoded based on basic IR manipulation utilities. However, some IRs rely on the unification based pattern matching capabilities of their implementation language, including CIL [6] which has been implemented using OCaml. For others, several independent transformation systems are available [8]. Examples include Stratego [11] and TXL (the Tree Transformation Language) [2]. Both are based on unification based term rewriting approaches. The main focus of those transformation systems is on steering the application of rules by fixing *strategies* for applying them, which is beyond the scope of our system.

7 Conclusion

Within this paper we present a novel approach for implementing code transformations within source-to-source compilers. By combining concepts of conventional term rewriting solutions and regular expressions our approach is capable of handling arbitrary tree structures. The expressiveness of our patterns is further extended by the introduction of the aT (any tree) and rT (recursive tree) constructs. We have formalized our approach and provided a list of examples, including a real-world transformation. The latter demonstrated the intuitive, declarative characteristic and practical applicability of our solution.

References

[1] *clang: a C language family frontend for LLVM*. [Online; 30-May-2013]. 2012. URL: `http://clang.llvm.org/`.

[2] James R Cordy. "The TXL source transformation language". In: *Science of Computer Programming* 61.3 (2006), pp. 190–210.

[3] *Insieme Compiler and Runtime Infrastructure*. University of Innsbruck. URL: `http://insieme-compiler.org`.

[4] Herbert Jordan et al. "INSPIRE: The Insieme Parallel Intermediate Representation". In: *PACT*. ACM. 2013, to be published.

[5] J. Merrill. "Generic and gimple: A new tree representation for entire functions". In: *GCC Developers Summit*. 2003, pp. 171–179.

[6] G. Necula et al. "CIL: Intermediate language and tools for analysis and transformation of C programs". In: *CC*. Springer. 2002, pp. 209–265.

[7] D. Quinlan. "ROSE: Compiler support for object-oriented frameworks". In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226.

[8] *The Program Transformation Wiki*. [Online; 30-May-2013]. 2013. URL: `http://program-transformation.org/`.

[9] Peter Thoman et al. "Adaptive Granularity Control in Task Parallel Programs using Multiversioning". In: *Euro-Par*. 2013, to be published.

[10] Eelco Visser. "A survey of rewriting strategies in program transformation systems". In: *ENTCS* 57 (2001), pp. 109–143.

[11] Eelco Visser. "Program transformation with Stratego/XT". In: *Domain-Specific Program Generation*. Springer, 2004, pp. 216–238.