Abstract Interpretation of Program Transformations using Regular Tree Grammars

by

J.T. Hidskes

to obtain the degree of Master of Science in Computer Science
at the faculty EEMCS of the Delft University of Technology,
to be defended publicly on Friday November 2, 2018 at 9:00 AM.

Student number:	4335732	
Thesis committee:	Prof. dr. E. Visser,	TU Delft, chair
	Dr. S. Erdweg,	TU Delft, supervisor
	S. Keidel, MSc,	TU Delft, supervisor
	Dr. A. Panichella,	TU Delft

Preface

This thesis has been submitted for the degree of Master of Science in Computer Science at the Delft University of Technology. It is not a usual thesis report. Rather, in agreement with my supervisors Sebastian Erdweg and Sven Keidel, we decided to write a conference paper. This "thesis paper" is the result of this work. We plan to submit the paper after my thesis defense.

My work investigates the analysis of program transformations in order to provide static guarantees about the output programs. To this end, I developed two static analyses: one that keeps track of the *sorts* of transformed programs, and one that maintains a *regular tree grammar* to represent the transformed programs. This work is done within the Programming Languages group of the faculty of Electrical Engineering, Mathematics and Computer Science (EEMS).

First of all, I would like to thank Sebastian for offering me the opportunity to do my thesis under his guidance. While working on this thesis, I was under supervision of Sven, with whom I had great discussions and collaboration. I want to thank Sven for all the support and guidance that he provided during my thesis. I thank the PL group for the lunch lectures that have given me many interesting insights in the field of programming language theory.

Finally, I would like to thank my family and friends for all their unconditional support.

Jente Hidskes
October 26, 2018

Abstract Interpretation of Program Transformations using Regular Tree Grammars

Jente Hidskes, Sven Keidel, and Sebastian Erdweg

Delft University of Technology, The Netherlands

Abstract. Many program transformation languages simplify the implementation of program transformations. However, they give only weak static guarantees about the generated code such as well-sortedness. Well-sortedness guarantees that a program transformation does not generate syntactically ill-formed code, but it is too imprecise for many other scenarios. In this paper, we present a static analysis that allows developers of program transformations to reason about their transformations on a more fine-grained level, namely that of syntactic shape. Specifically, we present an abstract interpreter for the Stratego program transformation language that approximates the syntactic shape of transformed code using regular tree grammars. As a baseline, we also present an abstract interpreter that guarantees well-sortedness. We prove parts of both abstract interpreters sound.

1 Introduction

Program transformations translate code of an input language to code of an output language. Examples of program transformations are desugarings, refactorings, optimizations, and code generators. While transformation languages simplify the implementation of program transformations, many give only weak static guarantees about the generated code to the transformation developer. In this work, we develop a static analysis to help transformation developers reason automatically about program transformations.

Reasoning about program transformations is difficult because we are one additional metalevel removed from the program semantics. For example, instead of showing that a program yields well-typed values, we have to show that a program transformation yields programs that yield well-typed values. At the same time, to be useful in practice, we want to provide feedback automatically and without a heavy annotation burden. Probably for these three reasons, existing transformation languages provide only weak static guarantees [8, 27], namely that of well-sortedness. A generated program is well-sorted if it is syntactically well-formed in the output language. This guarantee is precise enough to ensure that we do not generate statements as operands of an arithmetic operator, yet it is too imprecise for many other scenarios.

For example, consider the following desugaring of Java extended with pairs, which we reproduce from Erdweg et al. [15]:

```
rules
  desugar-type: PairType(t1,t2) → |[ Pair<~t1,~t2> ]|
  desugar-expr: PairExpr(e1,e2) → |[ new Pair<>(~e1,~e2) ]|

strategies
  main = topdown(try(desugar-expr + desugar-type))
```

This desugaring transformation is written in Stratego, a language featuring rewrite rules and strategies [34]. The two rewrite rules above use pattern matching to select pair types and expressions, respectively. They then generate representations of pair types and expressions using the `Pair` class. The main rewriting strategy traverses the input AST top-down and tries to apply both rewrite rules at every node, leaving the node unchanged if neither rule applies.

Developers of desugaring transformations need to ensure that all extension constructs are rewritten to core language constructs. In our example above, we want to guarantee that `main` yields a Java program without pairs. However, Stratego is untyped and does not provide such insurance. Other transformation languages like Maude [8] ensure well-sortedness and statically check if the result is a syntactically well-formed Java program. However, well-sortedness is too imprecise for our example because it cannot distinguish sort `Expr` that allows pair expressions from a sort `Expr` that disallows pairs. It might be possible to encode the desugaring property using sorts, but this would require duplicating the Java grammar and changing the transformation code. We would much rather find a generic approach that supports reasoning about generated code at a more fine-grained level than sorts.

In this paper, we implement a static analysis for program transformations by the means of abstract interpretation [10]. Specifically, we present a generic interpreter for the Stratego program transformation language that is parametric in its domain-specific semantics. We can derive executable static analyses from this generic interpreter by instantiating it with a specific semantics. Our architecture follows the design of Keidel et al., who showed that the generic interpreter simplifies the soundness proof of derived interpreters [26]. To this end, Keidel et al. instantiated a generic interpreter for Stratego to derive a concrete interpreter and a proof of concept tree-shape analysis, which they then showed sound. In this paper, we show that our generic Stratego analysis framework also supports the derivation of realistic static analyses that are relevant for transformation developers.

We present two relevant abstract domains for Stratego and derive corresponding static analyses. First, as a baseline, we realize an abstract domain that represents generated code by their sort. From this abstract domain, we derive a static analysis that ensures the well-sortedness of program transformations. We prove parts of the abstract domain for sorts sound.

To improve precision, we develop a second abstract domain that represents generated code as a regular tree grammar [2]. This abstract domain relies on a regular tree grammar describing the input language. It then transforms this input grammar to describe the possible output terms as precisely as possible. While regular tree grammars can contain cycles to describe infinitely many programs, many operations on the underlying language are decidable [9]. We explain how Stratego operations map to operations on regular tree grammars and prove their implementation sound. In addition, we adopt a widening operator for regular tree grammars [22] that ensures termination of the analysis. As before, we derive a static analysis by instantiating the generic interpreter.

In summary, we make the following contributions:

– We present a generic interpreter for Stratego and explain how to derive executable static analyses from it (Section 3).
– We realize an abstract domain for Stratego to ensure well-sortedness and prove parts of it sound (Section 4).
– We develop an abstract domain for Stratego based on regular tree grammars and prove parts of it sound (Section 5).

2 The Stratego Program Transformation Language

In this work we focus on Stratego, a domain-specific language to describe program transformations [34]. Stratego consists of a core language [34] and a surface language which provides a richer set of abstractions defined in terms of these core constructs [7]. In this section, we briefly introduce parts of the surface and core language that we revisit throughout the paper.

In Stratego, programs are represented by *terms*, i.e., their abstract syntax trees. The fundamental construct of Stratego is *term rewrite rules*. For example, the following rewrite rule simplifies arithmetic expressions by removing additions with zero and multiplications with one or zero.

```
simplify: Add(0,x) → x
simplify: Add(x,0) → x
simplify: Mul(1,x) → x
simplify: Mul(x,1) → x
simplify: Mul(0,x) → 0
simplify: Mul(x,0) → 0
```

The rewrite rule matches a subject term against the term patterns on the left of the rule. If one of the patterns matches, it binds the variables of the pattern and produces a new term from the pattern on the right by substituting its free variables.

Rewrite rules desugar to constructs of the core language called strategies. For example, the rewrite rule `simplify` from above desugars to the following core constructs:

```
{?Add(0,x); !x} + {?Add(x,0); !x} + {?Mul(1,x); !x} +
{?Mul(x,1); !x} + {?Mul(0,x); !x} + {?Mul(x,0); !x}
```

In this example, the match strategy `?p` attempts to match the subject term against the pattern `p` and binds `p`'s free variables. If a match succeeds, the build strategy `!p` replaces the subject term with the instantiation of the term pattern `p` using the bindings from the environment. The sequence operator `s1;s2` passes the subject term through the first and second strategy. If a match fails, the choice operator `+` executes the next match.

As is, `simplify` only applies to the root of the term. However, to make `simplify` useful, we need to apply it to every position in the term. This can be done with the `top-down` strategy, as shown in the introduction, which traverses the term from top to bottom trying to apply the given strategy. Strategy `top-down` is not a core construct. Instead it is implemented with the generic traversal operator `all(s)`, which applies the strategy `s` to all subterms of the given term.

In the following sections, we revisit these core language constructs again and explain how we implement an analysis for them.

3 An Analysis Framework for Stratego

Creating new static analyses *from scratch* and proving them sound is a laborious and error-prone process. To simplify this process, we created an analysis framework for Stratego. The key idea in this framework is to separate independent semantics from semantics that are specific to concrete and abstract interpreters. The independent semantics is captured in a generic interpreter, which exposes a collection of interfaces for the semantics that are specific to different abstract or concrete interpreters, see figure 1. Both abstract and concrete semantics then instantiate this generic interpreter by implementing its interfaces. The interfaces consist of operations that implement primitive functionality, such as pattern matching of terms or failure of strategies. Creating new analyses in this framework takes less effort and is less error-prone, because only analysis-specific functionality needs to be provided and proven sound with the criteria yielded by the concrete interpreter. Our framework is inspired by a case study in earlier work by Keidel et al. on compositional abstract interpreters [26]. In this work, Keidel et al. developed the generic interpreter for Stratego to verify that their approach scaled to real world languages. The analysis framework is implemented in Haskell and the code is open source.[1]

In the following, we describe our generic interpreter for Stratego. The analysis framework is based on Haskell arrows [24]. Arrows allow different semantics to modify the control-flow and effects of the generic interpreter. For example, semantics can specify how failure of strategies is propagated. On a high-level, arrows describe computations which consume values of a certain input type and produce values of a certain output type.

The generic interpreter is a function `eval` that takes a strategy of type `Strategy` and produces an arrow computation `c t t` that takes a term as input and produces a term as output:

```
eval :: (IsTerm t c, ArrowFail c, ArrowFix (Strategy,t) t c, …)
    ⇒ Strategy → c t t
```

[1] https://github.com/hjdskes/sturdy/tree/master/stratego

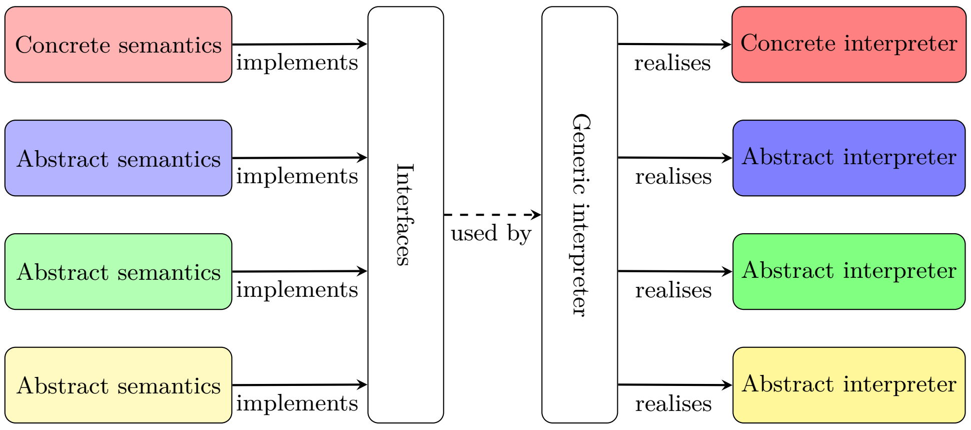


Fig. 1: The framework and its components.

The generic interpreter is parametric in its arrow type `c` and term type `t` to allow different semantics to instantiate these types differently. The interfaces of the generic interpreter consist of the type classes `IsTerm`, which defines operations on terms, `ArrowFail`, which defines an operation `fail` that causes a strategy to fail, `ArrowFix`, which defines a fixpoint computation `fix`, and further type classes not shown here.

The interfaces also consist of type classes not specific to Stratego, such as a type class for ordering and a type class for least upper bounds. Abstract interpretation requires that its abstract domains are ordered, i.e., that we can compare two abstract values a_1, a_2 for precision, written $a_1 \sqsubseteq a_2$. Abstract interpretation also requires that the abstract domain is finitely complete, i.e., that all elements a_1, a_2 have a least upper bound $a_1 \sqcup a_2$. The least upper bound is uniquely defined: for two abstract values a_1, a_2 we have $a_1 \sqsubseteq (a_1 \sqcup a_2)$ as well as $a_2 \sqsubseteq (a_1 \sqcup a_2)$ [30].

The implementation of the generic interpreter uses the pretty notation of arrow computations [31] featured by GHC, similar to do-notation for monads:

```
data Strategy = Seq Strategy Strategy | Match Pat | Build Pat
              | All Strategy | …

eval = fix $ λev strat → case strat of
  Seq s1 s2 → proc t → do
    t'  ← ev s1 ≺ t
    t'' ← ev s2 ≺ t'
    returnA ≺ t''
  Match pat → proc t → match ≺ (pat,t)
  Build pat → proc _ → build ≺ pat
  All s     → all (ev s)
  …
```

For example, the implementation of the sequence operator `Seq` describes an arrow computation (`proc t → ...`) that binds input term `t`. Term `t` is used as input to `ev s1`, which runs strategy `s1` and yields term `t'`. This term is then passed to the second strategy `s2`, whose result is returned using `returnA`. The generic interpreter uses the fixpoint combinator `fix` to allow the semantics to specify how to recurse over strategies. The fixpoint of an analysis is specific to that analysis and hence defined in the `ArrowFix` type class. In the following, we discuss the implementation in the generic interpreter of the core constructs that we discussed in section 2: `Match`, `Build` and `All`. These constructs are implemented with recursive helper functions, whose code is shown in listing 1. A recurring theme in the interfaces is that they destruct a higher-level Stratego construct into smaller, more primitive functionality.

```
data Pattern = Var String | Cons String [Pattern] | …

class Arrow c ⇒ IsTerm t c where
  equal :: c (t,t) t
  matchCons :: c ([p],[t]) [t] → c (String,[p],t) t
  cons :: c (String,[t]) t
  mapSubterms :: c [t] [t] → c t t
  …

match :: (IsTerm t c, IsTermEnv env t c) ⇒ c (Pattern,t) t
match = proc (pat,term) → case pat of
  Var x →
    lookup (proc (t',(x,t)) → do
              t'' ← equal ≺ (t,t')
              insert ≺ (x,t'')
              returnA ≺ t'')
            (proc (x,t) → do
              insert ≺ (x,t)
              returnA ≺ t)
      ≺ (x,(x,t))
  Cons c ps → matchCons (zipWith match) ≺ (c,ps,t)
  …

build :: (IsTerm t c, IsTermEnv env t c) ⇒ c Pattern t
build = proc pattern → case pattern of
  Var x → lookup' returnA fail ≺ x
  Cons c ps → do
    ts ← map build ≺ ps
    cons ≺ (c,ts)
  …

all :: IsTerm t c ⇒ c t t → c t t
all s = mapSubterms (map s)
```

Listing 1: Generic semantics for `Match` and `Build`.

Function `match` matches a term against a given pattern. The pattern can either be a variable, a constructor with subpatterns or a string or number literal (not shown). In case the pattern is a variable, `match` calls `lookup` from the `IsTermEnv` type class (not shown) to retrieve the variable from the current environment. This type class defines functions to manage an environment of variable bindings. Function `lookup` additionally takes two arrow computations. The first computation is called if a variable binding exists in the environment; otherwise, the second computation is called. To implement linear pattern matching [34], in case a variable binding exists the bound term is compared for equality with the given term. Comparing two terms for equality is done with the function `equal`, also defined in the `IsTerm` type class. Afterwards, the new term is bound to the variable in the environment. In case of a constructor pattern, the constructor of the pattern is matched by `matchCons` from the `IsTerm` type class against the top-level constructor of the term. If this succeeds, the implementation of `matchCons` should then recursively match the subterms of `t` respectively against the list of subpatterns `ps`. The generic interpreter thus reduces matching a term to matching a constructor or (not shown in the code) a number literal or a string literal. Binding variables, looking up variables from the environment and performing linearity checks are all handled in the generic interpreter.

Function `build` builds a term from a given pattern and variables bound in the environment. As before, the pattern can either be a variable, a constructor with subpatterns, or a string or number literal. In case the pattern is a variable, `build` returns the term bound to this variable. In case the pattern is a constructor pattern, `build` recursively builds the terms for the subpatterns and with these creates a new term with the given constructor of the pattern as top-level constructor. The assembling of the subterms and constructors is handled by interface operation `cons` from the `IsTerm` type class. As with matching patterns, the generic interpreter manages the environment and variable lookups.

Function `all s` applies strategy `s` to all subterms of the current term. The generic interpreter maps all traversal combinators to that of a single function, `mapSubterms`, defined in the `IsTerm` type class. Function `mapSubterms` maps an arrow-computation over the list of subterms, while leaving the top-level constructor the same. Function `map :: c x y → c [x] [y]` then applies the strategy `s` to each element of the list. The generic interpreter thus reduces all of Stratego's traversal combinators to a single interface function; derived analyses need only define how to traverse subterms.

In conclusion, a new interpreter needs to implement only the required type classes such as `IsTerm`, `IsTermEnv`, `ArrowFail` and `ArrowFix`. It may then call the generic interpreter function `eval` and get all shared semantics for free. In the following sections we show how to instantiate this framework to obtain two analyses.

4 An Abstract Interpreter for Well-Sortedness

In order to establish a baseline for our tree-shape analysis, we instantiated our framework with a sort analysis. This analysis allows us to show that a transformed program is syntactically well-formed in its output language. While this guarantee is precise enough to ensure that no statements are generated as operands of an arithmetic operator, it is too imprecise to guarantee that the desugaring of section 1 produces valid Java code without pairs. We develop our abstract interpreter by implementing the interfaces of the generic interpreter as shown in section 3. In this section, we show how these interfaces are implemented on the abstract domain of sorts.

In Stratego we can differentiate between eight sorts, as shown below:

```
data Sort = Bottom | Top | Lexical | Numerical | Option Sort |
    List Sort | Tuple [Sort] | Sort SortId
```

There are two subtleties in ordering sorts. First, while there is a dedicated sort `Lexical` for string literals, other sorts may also be lexical. For example, the Stratego *signature* `ID: [a-zA-Z]+ → Exp` declares the constructor `ID` to be any combination of one or more upper- and lowercase characters. Hence, the sort is `Sort "Exp"`, but it is also a lexical sort. Second, when comparing two non-identical sorts s_1, s_2 (e.g. s_1 being an expression and s_2 a statement), if sort s_1 can be transformed into sort s_2, then s_1 is more precise than s_2. In order to accommodate these subtleties, sorts need to carry a context that captures all this information. With this context present, a preorder can be defined as required. The least upper bound is then uniquely defined by this ordering. With this machinery in place, we are ready to implement the interfaces of the generic interpreter.

The implementation of the `cons` function creates an arrow that receives a constructor name `c` and a list of sorts `s1 ... sn` as input, and it must create the sort `s` of the term `c`$(s_1 \dots s_n)$. The implementation of `cons` is straightforward, see listing 2. Using the context, it looks up the signatures `c: s1 ... sn → s` of constructor `c`. For each signature, it checks for the correct number and kind of argument sorts. If this holds, the sort of the constructor is returned. If it does not hold, `Top` is returned. The least upper bound is then taken over all matching signatures. The least upper bound of an arrow computation `⨆ f ≺ [a,b,c]` calls `f` on each element on the list and joins the results: `(f ≺ a) ⊔ (f ≺ b) ⊔ (f ≺ c)`. If no signatures are found, the interpreter also returns `Top`.

```
cons = proc (c, ts) → do
  ctx ← askContext ≺ ()
  returnA ≺ case lookup c (signatures ctx) of
    Just sigs →
      ⨆ (arr (λ(ts',s) → if ts ⊑ ts' then s else Top))
        -<< sigs
    Nothing → Top
```

Listing 2: Building a constructor term in the sort analysis.

The `matchCons` function matches a sort against a constructor and calls the match function of the generic interpreter to recursively match the sort's parameters against the subpatterns. The implementation of `matchCons` is also straightforward, see listing 3. The arrow created by `matchCons` receives the constructor `c` to match against, a list of subpatterns `ps` and the sort `s` to match. It then looks up the signatures `c: p1 ... pn → s'` of constructor `c`. For each signature, it checks for the correct number of subpatterns and whether the sort of the pattern is more precise than the sort being matched. If this holds, the sort's parameters are matched recursively against the sorts of the subpatterns. Finally, if this does not fail, the sort of the pattern is returned as the result of the match. One subtlety to note here is that, while the matched term may be of the correct sort, there is no way to ensure that it is also the correct constructor. As a result, the interpreter also executes the failure case, taking the least upper bound of both paths. Since there might be multiple matching signatures, the least upper bound is taken over all computations. If no signatures are found, the interpreter executes both the failure case and the success case with the sort `Top`, taking their least upper bound.

```
matchCons match = proc (c,ps,s) → do
   ctx ← askContext ≺ ()
   case lookup c (signatures ctx) of
     Just sigs →
       ⨆ (proc (ts,s') →
         if length ts == length ps && s' ⊑ s
           then
             (fail ≺ ()) ⊔ (do _ ← match ≺ (ps,ts); returnA ≺ s)
           else fail ≺ ()) -<< sigs
     Nothing → (fail ≺ ()) ⊔ (returnA ≺ Top)
```

Listing 3: Matching a term against a constructor in the sort analysis.

The implementation of `equal` has two subtleties worth mentioning. Two sorts t_1, t_2 are equal if t_1 can be transformed into t_2 or vice versa, i.e., if $t_1 \sqsubseteq t_2$ or $t_2 \sqsubseteq t_1$. Furthermore, while the sorts may be equal, we can never ensure that the actual terms are. For this reason, the interpreter computes both paths and the least upper bound is taken over the respective outcomes. With these subtleties in mind, the implementation is straightforward, as shown in listing 4.

```
equal = proc (s1,s2) →
  if | s1 ⊑ s2 → (fail ≺ ()) ⊔ (returnA ≺ s2)
     | s2 ⊑ s1 → (fail ≺ ()) ⊔ (returnA ≺ s1)
     | otherwise → fail ≺ ()
```

Listing 4: Testing equality of two terms in the sort analysis.

Finally, traversing terms. Recall from section 3 that our generic interpreter maps all traversal combinators to a single function `mapSubterms`. In the sort analysis, however, we have to traverse sorts. Since sorts do not have subterms, we have to resort to an alternative approach. We retrieve all signatures $c : s'_1 \dots s'_n \rightarrow s'$ where $s' \sqsubseteq s$ from the context, and map over the sorts $s'_1 \dots s'_n$ of each signature. The results are combined using the least upper bound operator. In the implementation below, function `signaturesOf` retrieves all signatures $c : s'_1 \dots s'_n \to s'$ where $s' \sqsubseteq s$ from the context.

```
mapSubterms f = proc s → do
  ctx ← askContext ≺ ()
  ⊔ (proc (c,ts) → do
    ts' ← f ≺ ts
    cons ≺ (c,ts')
  ) ≺ ctx `signaturesOf` s
```

Listing 5: Traversing a term in the sort analysis.

4.1 Calculating the fixpoint of the sort analysis

Stratego transformations can be recursive. For example, the `top-down` strategy applies a strategy to every position in an AST and hence needs to be recursive. Because of recursion our static analysis for Stratego needs to take special care, otherwise the analysis might diverge and not produce a result. In this subsection we describe how the sort analysis computes the fixpoint of recursive strategies to avoid non-termination.

The sort analysis has to solve one problem when analyzing recursive transformations: the sort of a recursive transformation can be infinite. For example, consider the following transformation:

```
foo = map(foo)
```

The program defines a strategy `foo`, that calls strategy `map` with `foo` recursively, where `map` applies a strategy over every element of a list. The only valid sort for the output of `foo` would be an infinitely deeply nested list. However, we cannot infer this sort with our sort analysis, because the analysis would not terminate. Hence, our analysis has to detect this case and return a sort which overapproximates the actual sort of the transformation. For example, a valid overapproximation for the output of `foo` would be the sort `List Top`, a list of an unknown element sort.

To analyze recursive transformations we use a fixpoint algorithm for big-step semantics by Darais et al. [14]. The algorithm detects recursive calls to the abstract interpreter and enforces termination. For example, if we analyze `foo` from above with the sort `Top` as input, the abstract interpreter in its first recursive call will call `foo` with `Top` again. This fixpoint algorithm detects this recursive call and returns the result of the first call of `foo` instead of diverging.

4.2 Soundness

In this subsection we show the soundness lemmas of matching and building terms, equality on terms and traversing of terms of the sort analysis. Proofs of these lemmas may be found in appendix A.1.

Lemma 1. *Term construction is sound. In particular, we prove soundness of* `cons`*,* `stringLiteral` *and* `numberLiteral`*.*

Lemma 2. *Matching a term against a term pattern is sound. In particular, we prove soundness of* `matchCons`*,* `matchString` *and* `matchNumber`*.*

Lemma 3. *Term equality is sound. In particular, we prove soundness of* `equal` *for constructor terms.*

Lemma 4. *Mapping over subterms is sound. In particular, we prove soundness of* `mapSubterms`*.*

In the next section, we discuss an analysis that achieves finer grained results than the sort analysis shown in this section.

5 An Abstract Interpreter for Tree-Shape Analysis

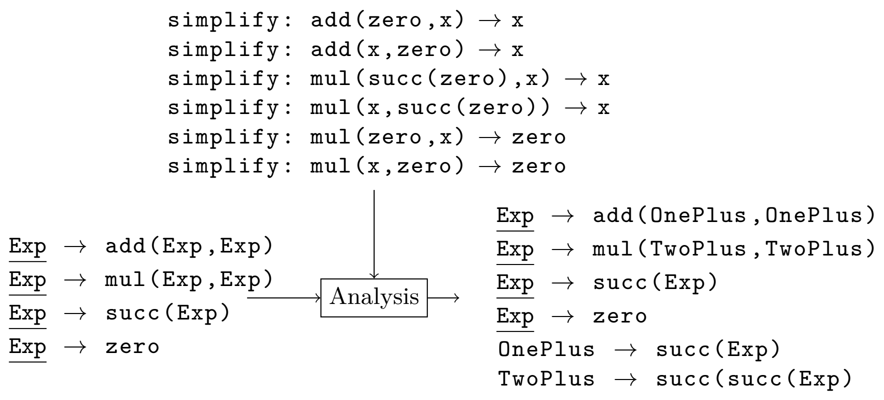
In the previous section we discussed an analysis that ensures the well-sortedness of Stratego program transformations. However, this analysis is not very precise. For example, the sort analysis does not allow us to show that the desugaring of section 1 produces valid Java code without pairs. In other words, we cannot use it to answer questions about the structure of programs produced by a transformation.

To address this short-coming, we define an analysis for Stratego based on a more precise abstract domain: regular tree grammars (RTGs) [2, 9]. For example, in figure 2 we analyze the transformation `simplify` from section 2. We provide a grammar that describes the inputs of `simplify`, i.e., the grammar of all possible arithmetic expressions. As result of the analysis, we obtain a grammar in which all top-level terms contain no additions with zero and no multiplications with one or zero. In the remainder of this section, we define an abstract domain based on RTGs, discuss our implementation of the analysis and prove parts of it sound.

5.1 Regular tree grammars as abstract domain

In this subsection, we define an abstract domain for regular tree grammars based on work by Cousot and Cousot [12]. First we define regular tree grammars more formally.

We repeat the definition of regular tree grammars found in existing literature [2, 9]. An RTG denotes a set of terms constructed from an alphabet $\mathcal{F}$ of ranked terminal symbols. A grammar $(S, \mathcal{F}, \mathcal{N}, \mathcal{R})$ consists of a set of non-terminals $\mathcal{N}$, a start symbol $S \in \mathcal{N}$ and a set of productions $\mathcal{R}$. Each production has the form $N \to \beta$, where $N \in \mathcal{N}$ is a non-terminal and β is either a non-terminal or a term $f(N_1 \ldots N_n)$ constructed from a terminal symbol $f \in \mathcal{F}$ and



```
simplify: add(zero,x) → x
simplify: add(x,zero) → x
simplify: mul(succ(zero),x) → x
simplify: mul(x,succ(zero)) → x
simplify: mul(zero,x) → zero
simplify: mul(x,zero) → zero
```

```
Exp → add(Exp,Exp)
Exp → mul(Exp,Exp)
Exp → succ(Exp)
Exp → zero
```

```
Exp → add(OnePlus,OnePlus)
Exp → mul(TwoPlus,TwoPlus)
Exp → succ(Exp)
Exp → zero
OnePlus → succ(Exp)
TwoPlus → succ(succ(Exp)
```

Fig. 2: An example tree-shape analysis. The inputs are a Stratego transformation (top) and an RTG that generates the input language of this transformation (left). The output is an RTG (right) describing the result of the transformation.

non-terminals $N_1 \dots N_n \in \mathcal{N}$. For ease of presentation, we combine multiple right hand sides for the same non-terminal symbol with the pipe symbol |. A term is derived via successive applications of production rules, starting from the start symbol S of the grammar. We write $L(G)$ for the language of G, i.e., the set of all terms that can be derived from G.

To use RTGs as an abstract domain, we need to define an ordering in which each two elements have a least upper bound. The ordering $G_1 \sqsubseteq G_2$ is defined by the language that the grammars produce, i.e., $L(G_1) \subseteq L(G_2)$. With this ordering the least upper bound of two grammars G_1 and G_2 is the union of these two grammars.

5.2 Implementing an analysis for Stratego based on RTGs

In this subsection, we discuss the implementation of our analysis based on the abstract domain of regular tree grammars. As in section 4, we instantiate our analysis framework to obtain an abstract interpreter for Stratego. In particular, we discuss the implementations of `cons` that constructs a term from a constructor and list of subterms, `matchCons` that matches a term against a constructor and a list of subpatterns, `equal` that checks for equality of two terms and `mapSubterms` which maps a transformation over the subterms of a given term.

To implement these operations, we extended the tree automata library by Adams and Might [1].[2] We added inclusion testing, emptiness testing, a determinization function and other functionality specific to our needs on top of the already existing functions such as union, intersection and equality testing.

[2] The library is open-source and is available at https://github.com/hjdskes/tree-automata/.

We start by describing the implementation of `cons`, an operation that constructs a new term from a constructor c and a list of subterms. In the tree-shape analysis, instead of subterms, we are given *subgrammars* $G_1 = (S_1, \mathcal{F}_1, \mathcal{N}_1, \mathcal{R}_1)$ … $G_n = (S_n, \mathcal{F}_n, \mathcal{N}_n, \mathcal{R}_n)$. The operation assumes that the subgrammars have disjoint sets of non-terminals. The new grammar G is constructed by taking the union of all productions and by adding a new production $S \to c(S_1 \dots S_n)$ from a fresh start symbol S to the start symbols $S_1 \dots S_n$ of the subgrammars. More specifically, the new grammar G is defined by $(S, \bigcup \mathcal{F}_i, \bigcup \mathcal{N}_i, \{S \to c(S_1 \dots S_n)\} \cup (\bigcup \mathcal{R}_i))$. The implementation of `cons` is straightforward. Constructing a new grammar from a constructor name and a list of grammars is implemented in the tree grammar library with the function `addConstructor`.

Next, we discuss `matchCons`, an operation that matches a term against a constructor pattern $c(p_1 \dots p_n)$ and returns a refined term. Before we discuss the implementation of `matchCons`, let us first look at an example of the intended semantics. Assume we match pattern `foo(x,y)` with a grammar G with the two productions $S \to \texttt{foo}(A, B) \mid \texttt{bar}$ reachable from the start symbol. For each right hand side of the production, we need to test whether the constructor of the production is equal to `foo`. This is the case for $S \to \texttt{foo}(A, B)$, but not for $S \to \texttt{bar}$. Hence for this pattern, the match could succeed *or* fail. For the succeeding production $S \to \texttt{foo}(A, B)$, we then recursively match the subpatterns `x` and `y` on the grammar G starting with the non-terminals A and B respectively. For the failing production, we add failure to the result of `matchCons`.

We now discuss the implementation of `matchCons` in listing 6 for a constructor pattern $c(p_1 \dots p_n)$ and a grammar G. Operation `matchCons` first deconstructs G into a list of constructors reachable from the start symbol with lists of subgrammars for the subterms of terms with these constructors. That is, let $S \to f_1(N_{11} \dots N_{1n}) \mid \dots \mid f_m(N_{m1} \dots N_{mo})$ be the productions reachable from the start symbol. The list is constructed as follows

$$\texttt{deconstruct}(G) = [\ (f_1, G(N_{11}) \dots G(N_{1n}))\ \dots\ (f_m, G(N_{m1}) \dots G(N_{mo}))\ ].$$

Here $G(N)$ refers to the grammar G with a changed start symbol N. Operation `matchCons` then checks for each constructor f_i if it is equal to the constructor c of the pattern and if the arity of f_i is the same as the arity of c. Whenever this is the case, `matchCons` recursively matches each subgrammar against the corresponding subpattern. The refined grammars resulting from the recursive match are then recombined with `cons` to a grammar with top-level constructor c. Otherwise, if the constructor or arity does not match, `matchCons` adds failure to the overall result.

```
matchCons match = proc (ctor,patterns,g) → do
  ⨆ (proc (c,ps,c',ts) →
      if c == c' && length ps == length ts
        then do ts' ← match ≺ (ps,ts)
                cons ≺ (c,ts')
        else fail ≺ ())
   ≺ [ (ctor,patterns,c,ts) | (c,ts) ← deconstruct g ]
```

Listing 6: Matching a term against a constructor in the tree-shape analysis.

Next, we discuss the implementation of `equal` in listing 7, that checks if two terms are equal. In our tree-shape analysis, operation `equal` checks if terms in one grammar G_1 are equal to terms in the other grammar G_2:

$$\forall t_1 \in L(G_1), \forall t_2 \in L(G_2), t_1 = t_2.$$

This means `equal` has to distinguish three different cases depicted in figure 3: (i) the grammars are disjoint, (ii) both grammars produce the same single term, or (iii) the intersection of both grammars is not empty. In the first case, because the grammars are disjoint, none of the terms produced by G_1 and G_2 can be equal. Therefore, `equal` calls `fail`. In the second case, `equal` checks if both grammars produce only a single term that is in the intersection of both grammars. In this case, the term produced by G_1 has to be equal to the term produced by G_2 and `equal` succeeds. In the third case, G_1 contains terms which *are not* produced by G_2 for which the equality check fails. Furthermore, G_1 contains terms which *are* produced by G_2 for which the equality check eventually succeeds. Therefore, `equal` returns a result that describes that term equality could have failed or possibly succeeded with terms in the intersection of G_1 and G_2.

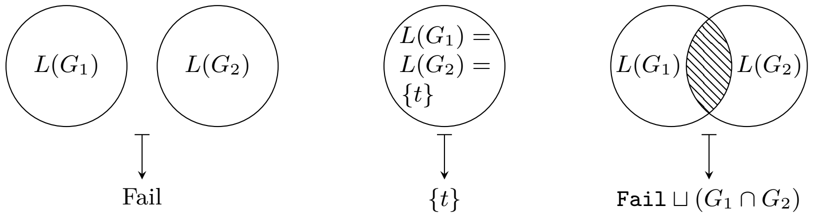


Fig. 3: The three cases of the intersection of the languages described by two grammars G_1, G_2: no intersection (left), equality (middle) and intersection (right).

```
equal = proc (g1, g2) → case intersection g1 g2 of
  g | isEmpty g → fail ≺ ()
    | isSingleton g1 && isSingleton g2 → returnA ≺ g
    | otherwise → (fail ≺ ()) ⊔ (returnA ≺ g)
```

Listing 7: Testing equality of two terms in the tree-shape analysis.

Finally, we discuss the implementation of `mapSubterms` in listing 8 for mapping a transformation over the subterms of the given term. In the tree-shape analysis, operation `mapSubterms` maps a strategy not over a single term, but over the subterms of all terms produced by a grammar. To implement this semantics, `mapSubterms` first deconstructs the grammar into constructors and subgrammars with the `deconstruct` function described above. For each pair of constructor and subgrammars, it then maps transformation `f` over the list of subgrammars and then reconstructs the resulting subgrammars with the old constructor into a new grammar. Operation `reconstruct` is similar to `cons`, except that it also deals with number and string literals.

```
mapSubterms f = proc grammar →
  ⨆ (proc (ctor,subterms) → do
      subterms' ← f ≺ subterms
      returnA ≺ reconstruct [(ctor,subterms')])
    ≺ deconstruct grammar
```

Listing 8: Traversing a term in the tree-shape analysis.

5.3 Calculating the fixpoint of the tree-shape analysis

As for the sort analysis, in the tree-shape analysis we need to take special care when analyzing recursive Stratego transformations. However, compared to the sort analysis the problem for the tree-shape analysis is more complicated. The reason is that the abstract domain of RTGs is infinite, i.e., there are infinite ascending chains $G_1 \sqsubseteq G_2 \sqsubseteq \ldots$ of regular tree grammars. Hence, when calculating the fixpoint, the analysis needs to avoid ascending one of these infinite chains and diverging. In this subsection, we explain how we approximate the fixpoint for the tree-shape analysis, while ensuring termination.

To avoid that the analysis ascends an infinite chain of RTGs, we use a standard technique called a widening operator [11]. In our analysis, a widening operator $G_1 \nabla G_2$ is a binary operator that takes two grammars G_1 and G_2 with $G_1 \sqsubseteq G_2$ and produces an RTG which is greater than G_1 and G_2. More importantly, if we fold ∇ over an infinite ascending chain, the widening operator produces a *finite* ascending chain that is element-wise greater than the original one. This means that the widening operator accelerates fixpoint iteration and in a finite amount of steps ends in a state that overapproximates the true fixpoint of the analysis.

Our widening operator is based on the topological clash widening introduced by Hentenryck et al. [22]. This is a widening on regular tree grammars that is guided by differences in the shapes of the grammars of successive steps of the interpreter. The idea underlying their approach is to determine in what direction the new grammar is growing compared to the old grammar. This information allows the widening operation to make an informed choice as to when to let a grammar grow and when and how to prevent growth.

In order to do this, the widening operation traverses the old and new grammars G_{old} and G_{new} in lockstep fashion, processing pairs of non-terminals ($N \in G_{old}, N' \in G_{new}$). When a cycle is detected, or when two non-terminal symbols are found producing a different set of constructors, the traversal is halted along that path. For some pairs $(\mathtt{N}, \mathtt{N}')$ where the traversal is halted, the new grammar is growing compared to the old grammar. In this case, the widening searches for an ancestor of $\mathtt{N}'$. When an ancestor of $\mathtt{N}'$ is found, the grammar G_{new} can be transformed by either replacing $\mathtt{N}'$ with its ancestor in a single right hand side (thus introducing a cycle), or by replacing $\mathtt{N}'$ with its ancestor throughout the grammar. The former is preferred, since it is the least drastic measure and thus preserves more precision. The grammar formed this way is a safe approximation of G_{new}.

For example, consider the situation reproduced from Hentenryck et al. [22]. Two successive iterations of the analysis have produced the grammars displayed in the top-left and top-right of figure 4. The widening operator starts to traverse these grammars in a lockstep fashion, starting with the pair of start symbols $(\mathtt{T}_0, \mathtt{T}'_0)$. Since there is no cycle, and both non-terminal symbols generate the constructors `nil` and `cons`, the traversal continues with the pairs $(\mathtt{T}_1, \mathtt{T}'_1)$ and $(\mathtt{T}_2, \mathtt{T}'_2)$. For $(\mathtt{T}_1, \mathtt{T}'_1)$, there is no cycle and both non-terminals produce the constructor `true`. Since there are no more non-terminals on this branch, the traversal halts here without finding a pair signaling growth. For the pair $(\mathtt{T}_2, \mathtt{T}'_2)$ the traversal finds that, while there is no cycle, the sets of generated constructors are not identical: $\mathtt{T}_2$ produces `nil`, while $\mathtt{T}'_2$ produces `nil` and `cons`. Hence, the traversal now halts along this path as well. In this case, however, the pair $(\mathtt{T}_2, \mathtt{T}'_2)$ does indicate growth. The widening then searches for an ancestor of $\mathtt{T}'_2$ and finds $\mathtt{T}'_0$. Since both $\mathtt{T}'_2$ and $\mathtt{T}'_0$ produce the constructors `nil` and `cons` and $\mathtt{T}'_1$ and $\mathtt{T}'_3$ both produce the constructor `true`, the ancestor $\mathtt{T}'_0$ overapproximates $\mathtt{T}'_2$. The widening then replaces $\mathtt{T}'_2$ with $\mathtt{T}'_0$ in the production rule $\mathtt{T}'_0 \rightarrow \mathtt{cons}(T'_1, T'_2)$, thus introducing a cycle from $\mathtt{T}'_0$ to $\mathtt{T}'_0$, producing the result as shown in bottom of figure 4.

```
                                       T0' → nil | cons(T1',T2')
T0 → nil | cons(T1,T2)                 T1' → true
T1 → true                              T2' → nil | cons(T3',T4')
T2 → nil                               T3' → true
                                       T4' → nil
```

```
Tr → nil | cons(T1,Tr)
T1 → true
```

Fig. 4: One iteration of the widening operator, reproduced from Hentenryck et al. [22].

5.4 Soundness

In this subsection we show the soundness lemmas of matching and building terms, equality on terms and traversing of terms of the tree-shape analysis. Proofs of these lemmas may be found in appendix A.2.

Lemma 5. *Term construction is sound. In particular, we prove soundness of `cons`, `stringLiteral` and `numberLiteral`.*

Lemma 6. *Matching a term against a term pattern is sound. In particular, we prove soundness of* `matchCons`*,* `matchString` *and* `matchNumber`*.*

Lemma 7. *Term equality is sound. In particular, we prove soundness of* `equal` *for constructor terms.*

Lemma 8. *Mapping over subterms is sound. In particular, we prove soundness of* `mapSubterms`*.*

6 Related Work

As stated in the introduction, other program transformation languages exist that give some form of guarantees. For example, Maude [8] and Rascal [27] ensure well-sortedness. However, well-sortedness is too imprecise for our use case because it cannot distinguish different language constructs with the same sort. For example, an extension construct that adds a new kind of expression has the same sort `Expr` as core language expressions. Our analysis has a more generic approach and provides reasoning about program transformations at a more fine-grained level. PLT Redex [29] is a declarative domain-specific language for specifying context-sensitive rewriting systems. It allows developers to annotate the input and output domains of term transformations with types. However, these type annotations are only checked at runtime. In contrast, we developed a static analysis such that program transformations may be verified before running them.

Al-Sibahi et al. present the design and implementation of a tool that verifies inductive type and shape properties for program transformations written in the Rascal transformation language [3]. This tool is similar to our work in both its implementation and its purpose.[3] Rabit infers an inductive refinement type that represents the shape of possible output of a transformation, given the shape of its input. The implementation is said to "extend standard regular tree grammar operations", but it is not shown how this is done, nor how an RTG is used as an inductive refinement type. To compute a fixpoint of their abstract interpreter, Al-Sibahi et al. use an approach similar to ours, where their abstract interpreter detects paths where execution recursively meets similar input and then reuses previous results, if any. Unfortunately they do not show what widening operator is being used. Nevertheless, the evaluation shows that Rabit can verify the properties that we want to verify in our work and does so with good runtime performance.

Haselhorst developed a type system based on regular tree grammars on top of a calculus for program transformation languages [21]. Regular tree grammars as types describe the abstract syntax of the manipulated programs, for example the grammar `N` $\rightarrow$ `Zero | Succ(N)` describes expressions that are natural numbers. In this way, the input and output language of a transformation can be specified and the validity of the transformation can be verified. Their type system provides

[3] We learned about this work at a late stage in the development of this paper.

stronger guarantees than well-sortedness. For example, the type describing the output language can ensure that all extension constructs are rewritten to core language constructs. However, their core calculus is not as expressive as Stratego. For example, their language does not support generic traversals, nor does it have a guarded choice (“try”). Furthermore, their type system is specific to their core calculus whereas our approach is a generic static analysis.

In similar spirit, XDuce [23] is a statically typed, functional tree transformation language designed for processing and transforming XML data. It features *regular expression types* and, correspondingly, *regular expression pattern matching*. Regular expression types describe structures in XML documents using regular expressions, similar to how tree grammar types describe terms in the work of Haselhorst. These types are equivalent in expressiveness to regular tree grammars. Regular expression pattern matching is similar to ML’s pattern matching but more powerful, since a pattern can include regular expression types to dynamically match values on those types. In essence, a pattern is thus a type and a value matches a pattern if it has the same type. ℂDuce [6] is a general purpose, statically typed, XML-oriented programming language based on XDuce. It is an attempt to generalize XDuce to a more general-purpose language. To this end, ℂDuce extends XDuce’s type system with a richer set of more general types and adds language constructs useful for general-purpose programming. Both XDuce and ℂDuce thus allow one to write XML transformations in a type-safe manner, on a level that is more fine-grained than well-sortedness. However, these languages are inherently tied to XML due to their design. They cannot be used as program transformation languages. In constrast, our static analysis can be applied to *any* transformation written in Stratego, whether it transforms programs or structured data such as XML.

Swiersta et al. have a long line of research on “extensible abstract syntax”. In his functional pearl “Data Types à la Carte”, Swierstra presents a technique for constructing data types and functions in a modular fashion [33]. Bahr and Tvitved [5] build on Swierstra’s work and present a library of compositional data types. Most notable in relation to our work is how they show that generic programming techniques can be implemented on top of their compositional data type framework. They show how this can be used to implement a desugaring function with stricter types that reflect the underlying transformation, thus providing a mechanism to reason about (generated) terms at a more fine-grained level than sorts. Axelsson presents the Syntactic library [4], which is a similar library partly derived from Swiersta’s *Data Types à la Carte*. Like Bahr and Tvitved’s framework, Syntactic provides extensible data types and generic traversals, but uses an application tree instead of a type-level fixed-point as initially defined by Swierstra. Although it is not shown in their paper, Axelsson states that Syntactic can also ensure that certain constructs are present or absent after certain transformations. This work in extensible abstract syntax thus shows a result similar to our goal in this work: if the extended language has a type `f'` (specified as the co-product of smaller, independent domains) and the core language has a similar type `f`, then both Bahr and Hvitved’s framework and

Axelsson's Syntactic can ensure that a transformation has the type signature `f'` $\rightarrow$ `f`. This line of work, however, is fundamentally about extending existing code without recompilation and creating new (domain-specific) programming languages with their abstract syntax trees open to extension. In contrast, our static analysis is created to verify existing program transformations.

The idea of using regular tree grammars for program analysis is due Jones and Muchnick [25]. Cousot and Cousot unified grammar-based static analyses with abstract interpretation-based analyses [12]. Cousot and Cousot have also shown that program transformation can be formalized within the theory of abstract interpretation [13]. In this work, they introduce a general, uniform and language-independent framework for reasoning on semantics-based program transformation through abstract interpretation. Amongst others, they argue that abstract interpretation accounts for the correctness of transformations which should preserve the semantics at some level of abstraction of irrelevant details. However, their framework takes the place of the program transformation language, i.e., their approach leads to a design methodology for program transformations.

There is interesting related work in approximating the set of reachable terms of a given term rewriting system. Genet et al. have a long line of research in this area [16, 18, 20]. It is shown that their technique can build a tree automaton that over-approximates the set of reachable terms. If the rewrite system preserves regularity, it can even build an automaton that recognizes *exactly* the set of reachable terms. This work is based on an equational tree automata completion algorithm [20]. To define these approximations, their completion algorithm uses an additional set of equations besides taking a tree automaton and a left-linear rewrite system as input. These equations are used to simplify an automaton, which has the effect of over-approximating the language that it recognizes [20]. Their work is implemented in the "Timbuk" tree automata library [18, 19]. This library can also compute the required set of equations if the term rewriting system encodes a functional program. Other work in reachability analysis is that of Gallagher and Rosendahl [17], who encode both tree automata and term writing systems into Horn clauses, allowing them to use static analysis tools for logic programs to perform approximations. Reachability analysis, however, is fundamentally a different approach. Given a set of terms and a term rewrite system, reachability analysis will compute *all* reachable terms. This set includes intermediate terms, which may be valid in the target language of the program transformation. This, in turn, will give developers of program transformations the wrong kind of feedback.

7 Conclusion and Future Work

In this work we presented a static analysis that allows developers of program transformations to reason about their transformations on a finer grained level than well-sortedness. We developed a generic interpreter for the Stratego program transformation language that is parametric in its domain-specific semantics and instantiated it with two abstract semantics. The first semantics realizes a sort analysis and is meant as a baseline for the second semantics. In order to increase precision and allow a finer-grained analysis, the second semantics approximates the syntactic shape of code using regular tree grammars. Since the focus in this work was on correctness of the tree-shape analysis, the implementation may be optimized leading to a smaller runtime footprint. Specifically, the algorithms for regular tree grammar operations may be optimized. Furthermore, there exist more performant data structures to represent regular tree grammars. Finally, the design of widening operators is experimental in nature, affecting both the performance and precision of the analysis. There is a large design space to consider, as is demonstrated by Mildner [30]. In his thesis, Mildner investigates a number of abstract domains based on regular tree grammars that differ, amongst others, in the widening operation that is used. Certainly, the other widening operations investigated in this thesis should be studied and perhaps compared for efficiency and precision.

A Soundness Proofs

The approach to compositional soundness proofs described by Keidel et al. reduces a soundness proof to proving smaller soundness lemmas over the primitive, analysis-specific implementation of the interfaces of the generic interpreter. Soundness of the interpreter as a whole then follows from a generic free theorem [26].

The concrete interpreter yields the soundness criteria that need to be proved [26]. Hence, before we show the proofs, we list the implementation of those functions of the concrete interpreter that are relevant to the proofs in listing 9.

```
data Term = Cons Constructor [Term] | …

cons = arr (uncurry Cons)

matchCons f = proc (c,ps,t) → case t of
  Cons c' ts | c == c' && length ps == length ts → do
    ts' ← f ≺ (ps,ts)
    cons ≺ (c,ts')
  _ → fail ≺ ()

equal = proc (t1,t2) →
  case (t1,t2) of
    (Cons c ts, Cons c' ts')
        | c == c' && length ts == length ts' → do
          ts'' ← zipWithA equal ≺ (ts,ts')
          cons ≺ (c,ts'')
    …
    (_,_) → fail ≺ ()

mapSubterms f = proc t →
```

```
  case t of
    Cons c ts → do
      ts' ← f ≺ ts
      cons ≺ (c,ts')
    …
```

Listing 9: The implementation of `cons`, `matchCons`, `equal` and `mapSubterms` in the concrete semantics.

We define Galois connections [10] $\alpha : \mathcal{P}A \rightleftarrows \hat{A} : \gamma$ for both analyses. The concretization function $\gamma(a)$ takes an abstract value a and returns the corresponding set of concrete values. The abstraction function $\alpha(c)$ takes a concrete value c (or set thereof) and returns the unique and most precise approximating abstract value. We write $e \mathrel{\dot{\sqsubseteq}} \hat{e}$ to mean that e is soundly approximated by $\hat{e}$. Abstract implementations are denoted with a *hat* $\hat{e}$.

A.1 Soundness proofs of the well-sortedness analysis

Definition 1. *We define the Galois connection $\alpha : \mathcal{P}A \rightleftarrows \hat{A} : \gamma$ by defining the concretization and abstraction functions. The concretization of a sort is the set of all terms that are of that sort. This depends on the context, i.e., for a context Γ containing signatures of the form $c : s_1 \dots s_2 \to s$ the concretization is defined $\gamma(s) = \{c(\gamma(s_1) \dots \gamma(s_n)) \mid c : s_1 \dots s_n \to s \in \Gamma\}$. The abstraction of a set of terms is defined as the most precise sort that represents all terms in the set, i.e., for a context Γ and a set of terms X the abstraction is defined $\alpha(X) = \bigsqcup\{t \mid \forall x \in X.\ x : s_1 \dots s_n \to t \in \Gamma.\ \sqsubseteq t\}$.*

Lemma 1. *Term construction is sound. In particular, we prove soundness of `cons`, `stringLiteral` and `numberLiteral`.*

Proof. Let us repeat the definition of $\widehat{\texttt{cons}}$:

```
$\widehat{\texttt{cons}}$ = proc (c, ts) → do
  ctx ← askContext ≺ ()
  case lookup c (signatures ctx) of
    Just sigs →
      ⨆ (arr (λ(ts',s) → if ts ⊑ ts' then s else Top))
          -<< sigs
    Nothing → returnA ≺ Top
```

We show that $\alpha(\texttt{cons}) \sqsubseteq \widehat{\texttt{cons}}$. That is, for all constructors c, lists of terms $[t_1 \dots t_n]$ and lists of sorts $[s_1 \dots s_n]$ where each t_i has sort s_i, we show $\alpha(\texttt{cons} \prec (c, [t_1 \dots t_n])) \sqsubseteq (\widehat{\texttt{cons}} \prec (c, [s_1 \dots s_n]))$. Furthermore, let Γ be the sort context passed to $\widehat{\texttt{cons}}$.

We distinguish the following three cases:

– In case the constructor is in the context and its number of arguments is the same as the length the list of sorts $[s_1 \dots s_n]$, i.e., $(c : s'_1 \dots s'_n \to s) \in \Gamma$ and

$s_1 \sqsubseteq s'_1 \ \ldots \ s_n \sqsubseteq s'_n$, then $c[t_1 \ldots t_n]$ must have a sort s $(\alpha(c[t_1 \ldots t_m]) = s)$. It follows,

$$\begin{aligned}\alpha(\texttt{cons} \prec (c, [t_1 \ldots t_m])) &= \alpha(\texttt{returnA} \prec c[t_1 \ldots t_m]) \\ &= \texttt{returnA} \prec s = \widehat{\texttt{cons}} \prec (c, [s_1 \ldots s_m]).\end{aligned}$$

– In case the constructor is in the context but the number of arguments differ from the length of the list of sorts $[s_1 \ldots s_n]$ or one of the argument sorts is not smaller than one of the sorts in $[s_1 \ldots s_n]$, then $\widehat{\texttt{cons}} \prec (c, [s_1 \ldots s_m])$ returns the sort `Top`, which is greater than any other sort.

$$\alpha(\texttt{cons} \prec (c, [t_1 \ldots t_m])) \sqsubseteq \texttt{returnA} \prec \texttt{Top} = \widehat{\texttt{cons}} \prec (c, [s_1 \ldots s_m]).$$

– In case the constructor does not occur in the context, we get

$$\alpha(\texttt{cons} \prec (c, [t_1 \ldots t_m])) \sqsubseteq \texttt{returnA} \prec \texttt{Top} = \widehat{\texttt{cons}} \prec (c, [s_1 \ldots s_m]).$$

The proofs of `stringLiteral` and `numberLiteral` are analogous. □

Lemma 2. *Matching a term against a term pattern is sound. In particular, we prove soundness of* `matchCons`*,* `matchString` *and* `matchNumber`*.*

Proof. Let us repeat the definition of $\widehat{\texttt{matchCons}}$:

```
matchCons f = proc (c,ps,s) → do
  ctx ← askContext ≺ ()
  case lookup c (signatures ctx) of
    Just sigs →
      ⨆ (proc (ts,s') →
        if length ts == length ps && s' ⊑ s
          then
            (fail ≺ ()) ⊔ (do _ ← f ≺ (ps,ts); returnA ≺ s)
          else fail ≺ ()) -<< sigs
    Nothing →
      (fail ≺ ()) ⊔ (returnA ≺ Top)
```

We show that $\alpha(f) \sqsubseteq \widehat{f} \implies \alpha(\texttt{matchCons}\ f) \sqsubseteq \widehat{\texttt{matchCons}}\ \widehat{f}$. That is, for all constructors c, list of patterns $[p_1 \ldots p_m]$, sorts s and terms $t = c'[t_1 \ldots t_n]$ of sort s, we show $\alpha(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t)) \sqsubseteq \widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], s)$. We distinguish the following three cases:

– In case the top-level constructor of t matches $(c' = c)$ and the number of subterms matches the number of patterns $(n = m)$, then there exists a signature $(c : s_1 \ldots s_n \to s') \in \Gamma$ with $\alpha(t_1) \sqsubseteq s_1 \ \ldots \ \alpha(t_n) \sqsubseteq s_n$.

$$\begin{aligned}&\alpha(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t)) \\ &\quad = \alpha(\texttt{cons} \lll \texttt{second}\ f \prec (c, [p_1 \ldots p_m], [t_1 \ldots t_m])) \\ &\quad \sqsubseteq (\texttt{returnA} \prec s) \lll \texttt{second}\ \widehat{f} \prec (c, [p_1 \ldots p_m], [s_1 \ldots s_m]) \\ &\quad \sqsubseteq \widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], s).\end{aligned}$$

– In case the top-level constructor of t do not match ($c' \neq c$) or the number of subterms is not the same as the number of patterns ($n \neq m$).

$$\begin{aligned}&\alpha(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t))\\ &\quad = \alpha(\texttt{fail} \prec ())\\ &\quad \sqsubseteq (\widehat{\texttt{fail}} \prec ())\\ &\quad \sqsubseteq \widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], s).\end{aligned}$$

– In case the constructor is not in the context, then

$$\begin{aligned}\alpha(\texttt{matchCons}\ f \prec (c, [t_1 \ldots t_m])) &\sqsubseteq (\widehat{\texttt{fail}} \prec ()) \sqcup (\texttt{returnA} \prec \texttt{Top})\\ &= \widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [s_1 \ldots s_m]).\end{aligned}$$

The proofs of `matchString` and `matchNumber` is analogous. □

Lemma 3. *Term equality is sound. In particular, we prove soundness of* `equal` *for constructor terms.*

Proof. Let us repeat the definition of $\widehat{\texttt{equal}}$:

```
equal = proc (s1,s2) →
  if | s1 ⊑ s2 → (fail ≺ ()) ⊔ (returnA ≺ s2)
     | s2 ⊑ s1 → (fail ≺ ()) ⊔ (returnA ≺ s1)
     | otherwise → fail ≺ ()
```

We show that $\alpha(\texttt{equal}) \sqsubseteq \widehat{\texttt{equal}}$. That is, for all terms $t = \texttt{Cons}\ c\ [t_1 \ldots t_n]$ and $t' = \texttt{Cons}\ c'\ [t'_1 \ldots t'_n]$ where $\alpha(t) \sqsubseteq s$ and $\alpha(t') \sqsubseteq s'$, $\alpha(\texttt{equal} \prec (t, t')) \sqsubseteq \widehat{\texttt{equal}} \prec (s, s')$. We distinguish two cases based on if the terms t and t' are equal or not and if s is a subsort of s' or vice versa.

– In case $t = t'$ and $s \sqsubseteq s'$, then $\texttt{equal} \prec (t, t)$ returns the term t and $\widehat{\texttt{equal}}$ fails and returns s'.

$$\begin{aligned}\alpha(\texttt{equal} \prec (t, t')) &= \alpha(\texttt{returnA} \prec t)\\ &\sqsubseteq (\texttt{returnA} \prec s)\\ &\sqsubseteq (\texttt{returnA} \prec s')\\ &\sqsubseteq (\widehat{\texttt{fail}} \prec ()) \sqcup (\texttt{returnA} \prec s')\\ &= \widehat{\texttt{equal}} \prec (s_1, s_2).\end{aligned}$$

– In case $t = t'$ and $s' \sqsubseteq s$, then $\texttt{equal} \prec (t, t)$ returns the term t and $\widehat{\texttt{equal}}$ fails and returns s. The proof is even simpler than the previous case.
– In case $t \neq t'$, it follows

$$\alpha(\texttt{equal} \prec (t, t')) = \alpha(\texttt{fail} \prec ()) \sqsubseteq (\widehat{\texttt{fail}} \prec ()) \sqsubseteq \widehat{\texttt{equal}} \prec (s_1, s_2).$$

The proof of `equal` for string and number literals is analagous. □

Lemma 4. *Mapping over subterms is sound. In particular, we prove soundness of* `mapSubterms`*.*

Proof. Let us repeat the definition of $\widehat{\texttt{mapSubterms}}$:

```
mapSubterms f = proc s → do
 ctx ← askContext ≺ ()
 ⨆ (proc (c,ts) → do
     ts' ← f ≺ ts
     cons ≺ (c,ts')
   ) ≺ ctx `signaturesOf` s
```

We show that $\alpha(f) \sqsubseteq \widehat{f} \implies \alpha(\texttt{mapSubterms}\ f) \sqsubseteq \widehat{\texttt{mapSubterms}}\ \widehat{f}$. That is, for all sorts s and terms $t = \texttt{Cons}\ c\ [t_1 \dots t_n]$ of sort s, $\alpha(\texttt{mapSubterms}\ f \prec t) \sqsubseteq \widehat{\texttt{mapSubterms}}\ \widehat{f} \prec s$. There exists signatures $c : s_1 \dots s_n \to s \in \mathit{\Gamma}$ with $\alpha(t_1) \sqsubseteq s_1 \dots \alpha(t_n) \sqsubseteq s_n$. It follows

$$\begin{aligned}\alpha(\texttt{mapSubterms}\ f \prec t) &= \alpha(\texttt{cons} \lll \texttt{second}\ f \prec (c, [t_1 \dots t_n])) \\ &\sqsubseteq (\texttt{cons} \lll \texttt{second}\ \widehat{f} \prec (c, [s_1 \dots s_n])) = (\widehat{\texttt{mapSubterms}}\ \widehat{f} \prec s)\end{aligned}$$

The proof of mapSubterms for string and number literals is analogous. □

A.2 Soundness proofs of the tree-shape analysis

Definition 2. *We define the Galois connection $\alpha : \mathcal{P}A \rightleftarrows \hat{A} : \gamma$ by defining the concretization and abstraction functions. The concretization of a regular tree grammar G is simply $L(G)$. The abstraction function of a set of terms X is defined $\alpha(X) = \bigsqcup\{G \mid X \subseteq L(G)\}$.*

We prove soundness using the concretization function, because the abstraction function may not always be well-defined on certain terms. This is different from the approach to proving soundness in appendix A.1 and in Keidel et al.'s earlier work in which the abstraction function is used. However, both formulations of soundness are correct. When using the abstraction function, one shows that the concrete value c is contained within the abstract value a, i.e., that $\alpha(c) \sqsubseteq a$. In contrast, when using the concretization function, we show that an abstract value a contains at least the concrete value c, i.e., that $c \in \gamma(a)$.

Lemma 5. *Term construction is sound. In particular, we prove soundness of* `cons`*,* `stringLiteral` *and* `numberLiteral`*.*

Proof. Let us repeat the definition of $\widehat{\texttt{cons}}$:

```
cons^ = proc (c,ts) → returnA ≺ addConstructor c ts
```

We show that $\texttt{cons} \in \gamma(\widehat{\texttt{cons}})$. That is, for all constructors c, lists of RTGs $[G_1 \dots G_n]$ and lists of terms $[t_1 \dots t_n]$ with $t_i \in L(G_i)$, we have $(\texttt{cons} \prec (c, [t_1 \dots t_n])) \in (\gamma(\widehat{\texttt{cons}}) \prec (c, [G_1 \dots G_n]))$.

Function $\widehat{\texttt{cons}}$ takes a constructor c and a list of RTGs $[G_1 \dots G_n]$ and creates a new RTG $G' = (S, \bigcup \mathcal{F}_i, \bigcup \mathcal{N}_i, \{R\} \cup (\bigcup \mathcal{R}_i))$ with unique start symbol S and production $R = S \to c(\text{start}(G_1) \dots \text{start}(G_n))$. Then, for each $t_1 \in L(G_1) \dots t_n \in L(G_n)$ it holds that $c(t_1 \dots t_n) \in L(\widehat{\texttt{cons}} \prec (c, [G_1 \dots G_n])$. We conclude $\texttt{cons} \in \gamma(\widehat{\texttt{cons}})$.

The proofs of stringLiteral and numberLiteral are analogous. □

Lemma 6. *Matching a term against a term pattern is sound. In particular, we prove soundness of* `matchCons`*,* `matchString` *and* `matchNumber`*.*

Proof. Let us repeat the definition of $\widehat{\texttt{matchCons}}$:

```
matchCons f = proc (c,ps,g) → do
  ⨆ (proc (c,ps,c',ts) →
      if c == c' && length ps == length ts
      then do
        ts' ← f ≺ (ps,ts)
        cons ≺ (c,ts')
      else fail ≺ ())
    ≺ [ (c,ps,c',ts) | (c',ts') ← deconstruct g ]
```

We show that $f \subseteq \gamma(\widehat{f}) \implies \texttt{matchCons}\ f \in \gamma(\widehat{\texttt{matchCons}}\ \widehat{f})$. That is, for all constructors c, lists of patterns $[p_1 \ldots p_m]$, RTGs G and terms $t = \texttt{Cons}\ c'\ [t_1 \ldots t_n] \in L(G)$, $(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t)) \in \gamma(\widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], G))$. Furthermore, because $t \in L(G)$, there exists a $(c', [G_1 \ldots G_n]) \in \texttt{deconstruct}(G)$ with $t_1 \in L(G_1)\ \ldots\ t_n \in L(G_n)$.

We distinguish two cases based on if the top-level constructor c' of t matches c and if the number of the subterms m is the same as the number of subpatterns n.

– In case $c = c'$ and $m = n$, it follows

$$\begin{aligned}
&(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t))\\
&\quad = (\texttt{cons} \lll \texttt{second}\ f \prec (c, ([p_1 \ldots p_m], [t_1 \ldots t_m])))\\
&\quad \in \gamma(\widehat{\texttt{cons}} \lll \texttt{second}\ \widehat{f} \prec (c, ([p_1 \ldots p_m], [G_1 \ldots G_m])))\\
&\quad \subseteq \gamma(\widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], G)).
\end{aligned}$$

– In case $c \neq c'$ or $m \neq n$, it follows

$$\begin{aligned}
&(\texttt{matchCons}\ f \prec (c, [p_1 \ldots p_m], t)) = (\texttt{fail} \prec ())\\
&\quad \in \gamma(\widehat{\texttt{fail}} \prec ()) = \gamma(\widehat{\texttt{matchCons}}\ \widehat{f} \prec (c, [p_1 \ldots p_m], G)).
\end{aligned}$$

The proofs of `matchString` and `matchNumber` are analogous. □

Lemma 7. *Term equality is sound. In particular, we prove soundness of* `equal` *for constructor terms.*

Proof. Let us repeat the definition of $\widehat{\texttt{equal}}$:

```
equal = proc (g1, g2) → case intersection g1 g2 of
  g | isEmpty g → fail ≺ ()
    | isSingleton g1 && isSingleton g2 → returnA ≺ g
    | otherwise → (fail ≺ ()) ⊔ (returnA ≺ g)
```

We show that $\texttt{equal} \in \gamma(\widehat{\texttt{equal}})$. That is, for all RTGs G, G' and terms $t = \texttt{Cons}\ c\ [t_1 \ldots t_n] \in L(G)$ and $t' = \texttt{Cons}\ c'\ [t'_1 \ldots t'_n] \in L(G')$, $\texttt{equal} \prec (t, t') \in \widehat{\texttt{equal}} \prec (G, G')$. We distinguish three cases based on if the terms t and t' are equal or not and whether the two grammars describe a single term or a set of terms.

– In case $t = t'$ and $L(G) = L(G') = t$, it follows

$$\begin{aligned}&\texttt{equal} \prec (t, t')\\ &\quad = \texttt{cons} \lll \texttt{second}\ (\texttt{zipWithA equal}) \prec (c, ([t_1 \ldots t_n], [t'_1 \ldots t'_n]))\\ &\quad \in \texttt{returnA} \prec G = \gamma(\widehat{\texttt{equal}} \prec (G, G')).\end{aligned}$$

– In case $t = t'$ and $L(G)$ and $L(G')$ describe a set of terms, it follows

$$\begin{aligned}&\texttt{equal} \prec (t, t')\\ &\quad = \texttt{cons} \lll \texttt{second}\ (\texttt{zipWithA equal}) \prec (c, ([t_1 \ldots t_n], [t'_1 \ldots t'_n]))\\ &\quad \in (\texttt{fail} \prec ()) \sqcup (\texttt{returnA} \prec G) = \gamma(\widehat{\texttt{equal}} \prec (G, G')).\end{aligned}$$

– In case $t \neq t'$, it follows

$$\texttt{equal} \prec (t, t') = \texttt{fail} \prec () \in \gamma(\widehat{\texttt{fail}} \prec ()) = \gamma(\widehat{\texttt{equal}} \prec (G, G')).$$

The proof of `equal` for string and number literals is analagous. □

Lemma 8. *Mapping over subterms is sound. In particular, we prove soundness of* $\widehat{\texttt{mapSubterms}}$*.*

Proof. Let us repeat the definition of $\widehat{\texttt{mapSubterms}}$:

```
mapSubterms f = proc g →
  ⨆ (proc (c,ts) → do
    ts' ← f ≺ ts
    returnA ≺ reconstruct [(c,ts')])
  ≺ deconstruct g
```

We show that $f \subseteq \gamma(\widehat{f}) \implies \texttt{mapSubterms}\ f \in \gamma(\widehat{\texttt{mapSubterms}}\ \widehat{f})$. That is, for all RTGs G and terms $t = \texttt{Cons}\ c\ [t_1 \ldots t_n] \in L(G)$, $\texttt{mapSubterms}\ f \prec t \in \gamma(\widehat{\texttt{mapSubterms}}\ \widehat{f} \prec G)$. Furthermore, because $t \in L(G)$, there exists a $(c, [G_1 \ldots G_n]) \in \texttt{deconstruct}(G)$ with $t_1 \in L(G_1)\ \ldots\ t_n \in L(G_n)$. It follows

$$\begin{aligned}&(\texttt{mapSubterms}\ f \prec t) = (\texttt{cons} \lll \texttt{second}\ f \prec (c, [t_1 \ldots t_n]))\\ &\quad \in \gamma(\texttt{arr}(\texttt{reconstruct}) \lll \texttt{second}\ \widehat{f} \prec (c, [G_1 \ldots G_n]))\\ &\quad = \gamma(\widehat{\texttt{mapSubterms}}\ \widehat{f} \prec G)\end{aligned}$$

The proof of `mapSubterms` for string and number literals is analogous. □

References

1. Adams, M.D., Might, M.: Restricting grammars with tree automata. PACMPL **1**(OOPSLA), 82:1–82:25 (2017). https://doi.org/10.1145/3133906, http://doi.acm.org/10.1145/3133906
2. Aiken, A., Murphy, B.R.: Implementing regular tree expressions. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Lecture Notes in Computer Science, vol. 523, pp. 427–447. Springer (1991)

3. Al-Sibahi, A.S., Jensen, T.P., Dimovski, A.S., Wasowski, A.: Verification of high-level transformations with inductive refinement types. CoRR **abs/1809.06336** (2018), http://arxiv.org/abs/1809.06336
4. Axelsson, E.: A generic abstract syntax model for embedded languages. In: Thiemann, P., Findler, R.B. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012. pp. 323–334. ACM (2012). https://doi.org/10.1145/2364527.2364573, http://doi.acm.org/10.1145/2364527.2364573
5. Bahr, P., Hvitved, T.: Compositional data types. In: Järvi, J., Mu, S. (eds.) Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 83–94. ACM (2011). https://doi.org/10.1145/2036918.2036930, http://doi.acm.org/10.1145/2036918.2036930
6. Benzaken, V., Castagna, G., Frisch, A.: Cduce: an xml-centric general-purpose language. In: Runciman, C., Shivers, O. (eds.) Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003. pp. 51–63. ACM (2003). https://doi.org/10.1145/944705.944711, http://doi.acm.org/10.1145/944705.944711
7. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundam. Inform. **69**(1-2), 123–178 (2006), http://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2), 187–243 (2002). https://doi.org/10.1016/S0304-3975(01)00359-0, https://doi.org/10.1016/S0304-3975(01)00359-0
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata (2007), release October, 12th 2007
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977), http://dl.acm.org/citation.cfm?id=512950
11. Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings. pp. 269–295 (1992)
12. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Williams, J. (ed.) Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995. pp. 170–181. ACM (1995). https://doi.org/10.1145/224164.224199, http://doi.acm.org/10.1145/224164.224199
13. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 178–190. ACM (2002), http://dl.acm.org/citation.cfm?id=503272

14. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). PACMPL **1**(ICFP), 12:1–12:25 (2017)
15. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: Lopes, C.V., Fisher, K. (eds.) Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011. pp. 391–406. ACM (2011)
16. Feuillade, G., Genet, T., Tong, V.V.T.: Reachability analysis over term rewriting systems. J. Autom. Reasoning **33**(3-4), 341–383 (2004). https://doi.org/10.1007/s10817-004-6246-0, https://doi.org/10.1007/s10817-004-6246-0
17. Gallagher, J.P., Rosendahl, M.: Approximating term rewriting systems: A horn clause specification and its implementation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5330, pp. 682–696. Springer (2008). https://doi.org/10.1007/978-3-540-89439-1_47, https://doi.org/10.1007/978-3-540-89439-1_47
18. Genet, T.: Automata completion and regularity preservation. Ph.D. thesis, IRISA, Inria Rennes (2017)
19. Genet, T., Gillard, T., Haudebourg, T., Cong, S.: Extending timbuk to verify functional programs. In: WRLA 2018 (2018)
20. Genet, T., Rusu, V.: Equational approximations for tree automata completion. J. Symb. Comput. **45**(5), 574–597 (2010). https://doi.org/10.1016/j.jsc.2010.01.009, https://doi.org/10.1016/j.jsc.2010.01.009
21. Haselhorst, K.: A Type System for Program Transformations based on Parametric Tree Grammars. Master's thesis, Philipps Universität Marburg (2012)
22. Hentenryck, P.V., Cortesi, A., Charlier, B.L.: Type analysis of prolog using type graphs. J. Log. Program. **22**(3), 179–209 (1995). https://doi.org/10.1016/0743-1066(94)00021-W, https://doi.org/10.1016/0743-1066(94)00021-W
23. Hosoya, H., Pierce, B.C.: Xduce: A statically typed XML processing language. ACM Trans. Internet Techn. **3**(2), 117–148 (2003). https://doi.org/10.1145/767193.767195, http://doi.acm.org/10.1145/767193.767195
24. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000)
25. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of lisp-like structures. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 244–256. ACM Press (1979). https://doi.org/10.1145/567752.567776, http://doi.acm.org/10.1145/567752.567776
26. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. Proc. ACM Program. Lang. **2**(ICFP), 72:1–72:26 (Jul 2018)
27. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009. pp. 168–177. IEEE Computer Society (2009). https://doi.org/10.1109/SCAM.2009.28, https://doi.org/10.1109/SCAM.2009.28

28. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows and their optimization. In: ACM Sigplan Notices. vol. 44, pp. 35–46. ACM (2009)
29. Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A visual environment for developing context-sensitive term rewriting systems. In: van Oostrom, V. (ed.) Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3091, pp. 301–311. Springer (2004). https://doi.org/10.1007/978-3-540-25979-4_21, https://doi.org/10.1007/978-3-540-25979-4_21
30. Mildner, P.: Type Domains for Abstract Interpretation: A critical study. Ph.D. thesis, Uppsala universitet (1999)
31. Paterson, R.: A new notation for arrows. In: Pierce, B.C. (ed.) Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001. pp. 229–240. ACM (2001)
32. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977). https://doi.org/10.1016/0304-3975(77)90044-5, https://doi.org/10.1016/0304-3975(77)90044-5
33. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758, https://doi.org/10.1017/S0956796808006758
34. Visser, E., Benaissa, Z., Tolmach, A.P.: Building program optimizers with rewriting strategies. In: Felleisen, M., Hudak, P., Queinnec, C. (eds.) Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998. pp. 13–26. ACM (1998)