A Systematic Approach to Abstract Interpretation of Program Transformations

Sven Keidel and Sebastian Erdweg

JGU Mainz, Germany

Abstract. Abstract interpretation is a technique to define sound static analyses. While abstract interpretation is generally well-understood, the analysis of program transformations has not seen much attention. The main challenge in developing an abstract interpreter for program transformations is designing good abstractions that capture relevant information about the generated code. However, a complete abstract interpreter must handle many other aspects of the transformation language, such as backtracking and generic traversals, as well as analysis-specific concerns, such as interprocedurality and fixpoints. This deflects attention.
We propose a systematic approach to design and implement abstract interpreters for program transformations that isolates the abstraction for generated code from other analysis aspects. Using our approach, analysis developers can focus on the design of abstractions for generated code, while the rest of the analysis definition can be reused. We show that our approach is feasible and useful by developing three novel interprocedural analyses for the Stratego transformation language: a singleton analysis for constant propagation, a sort analysis for type checking, and a locally-illsorted sort analysis that can additionally validate type changing generic traversals.

1 Introduction

Abstract interpretation is a technique to define sound static analyses [6]. Static analyses have proved useful in providing feedback to developers (e.g., dead code [4], type information), in finding bugs (e.g., uninitialized read [25], type errors [20]), and in enabling compiler optimizations (e.g., constant propagation [3], purity analysis [21]). It is therefore no surprise that the field of abstract interpretation and static analysis has seen significant attention both in academia and industry.

Unfortunately, the analysis of program transformations has not seen much attention so far. Program transformations are a central tool in language engineering and modern software development. For example, they are used for code desugaring, macro expansion, compiler optimization, refactoring, migration scripting, or model-driven development. The development of such program transformations tends to be difficult because they act at the metalevel and should work for a large class of potential input programs. Yet, there are hardly any static analyses

for program transformation languages available, and it appears to be difficult to develop such analyses. To this end, we identified the following challenges:

Domain-Specific Features Program transformation languages such as Stratego [24], Rascal [14], and Maude [5] aim to simplify the development of program transformations. Therefore, they provide domain-specific language features such as rich pattern-matching, backtracking, and generic traversals. These domain-specific language features usually cannot be found in other general-purpose languages and the literature on static analysis provides only little guidance on how to tackle them.

Term Abstraction Programs are first-class in program transformations and are represented as terms (e.g., abstract syntax trees). Therefore, analysis developers need to find a good abstraction for terms, such as syntactic sorts or grammars [8]. This term abstraction heavily influences the precision and usefulness of the analysis and most of the analysis development effort should be spent on the design of this abstraction. We expect analysis developers to experiment with alternative term abstractions: The design of good abstract domains is inherent to the development of any abstract interpreter and cannot be avoided.

Soundness Developing an abstract interpreter that soundly predicts the generated code of program transformations is difficult. This is because real-world transformation languages have many edge cases and an abstract interpreter has to account for all of these edge cases to be sound. Furthermore, transformation languages often do not have a formal semantics, which makes it hard to verify that the abstract interpreter covered all cases.

In this paper we present a systematic approach to develop abstract interpreters for program transformation languages that addresses these challenges. It is based on the well-founded theory of *compositional soundness proofs* [13] and *reusable analysis components* [12]. In particular, our approach captures the core semantics of a transformation language with a *generic interpreter* [13] that does not refer to any analysis-specific details. This simplifies the analysis of the domain-specific language features. Furthermore, our approach decouples the term abstraction from the remainder of the analysis through an interface. This means that any term abstraction that implements this interface gives rise to a complete abstract interpreter. Thus, analysis developers can fully focus on developing good term abstractions. Lastly, our approach reuses language-independent functionality, such as abstractions for environments, exceptions and fixpoints, from the Sturdy standard library. This not only reduces the analysis development effort, but also simplifies its soundness proof as we can rely on the soundness proofs of the Sturdy library [12].

We demonstrate the feasibility and usefulness of our approach by developing abstract interpreters for Stratego [24]. Stratego is a complex dynamic program transformation language featuring rich pattern matching, backtracking, generic traversals, higher-order transformations, and an untyped program representation. Despite these difficulties, based on our approach we developed three novel

abstract interpreters for Stratego: We developed a constant propagation analysis, a sort analysis, which checks that transformations are well-typed, and an advanced sort analysis, which can even validate type-changing generic traversals which produce ill-sorted intermediate terms. Our systematic approach was crucial in allowing us to focus on each of these abstract domains without being concerned with other aspects of the Stratego language. We implemented the analyses in Haskell in the *Sturdy* analysis framework and the code of the analyses is open-source.[1]

In summary, we make the following contributions:

- We propose a systematic approach to the development of abstract interpreters for program transformations, that lets analysis developers focus on designing the term abstraction.
- We show that many features of program transformation languages can be implemented on top of existing analysis functionality and do not require specific analysis code.
- We demonstrate the feasibility and usefulness of our approach by applying it to Stratego, for which we develop three novel abstract interpreters.

2 Illustrating Example: Singleton Analysis

The static analysis of program transformations can have significant merit helping developers to understand and debug their code and helping compilers to optimize the code. For example, we would like to support the following analyses: *Singleton analysis* to enable constant propagation, *purity analysis* to enable function inlining, *dead code* analysis to discover irrelevant code, *sort analysis* to prevent ill-sorted terms. While these and many other analyses would be useful, their development is complicated. In this section, we illustrate our approach by developing a singleton analysis for Stratego [24].

2.1 Abstract Interpreter for Program Transformations = Generic Interpreter + Term Abstraction

The development of analyses for program transformations is complicated for two reasons. First, each analysis requires a different term abstraction, with which it represents the generated code. The choice of term abstraction is crucial since it directly influences the precision, soundness, and termination of the analysis. Second, program transformation languages provide domain-specific language features such as rich pattern matching, backtracking, and generic traversals. Soundly approximating these features in an analysis is not easy, and resolving this challenge for each analysis anew is impractical.

In this paper, we propose a more systematic approach to developing static analyses for program transformations. To support static analyses for a given transformation language, we first develop a generic interpreter that implements

[1] `https://gitlab.rlp.net/plmz/sturdy/tree/master/stratego`

```
data Pat = Var String | As String Pat | Cons String [Pat]
  | StringLit String | NumLit Int | Explode Pat Pat

match :: (IsTerm term c, ArrowEnv String term c,
          ArrowExcept () c, ...) ⇒ c (Pat,term) term
match = proc (pat,t) → case pat of
  Var "_" → returnA ≺ t
  Var x → lookup
    (proc (t',(x,t)) → do t'' ← equal ≺ (t,t');
      insert ≺ (x,t''); returnA ≺ t'')
    (proc (x,t) → insert ≺ (x,t); returnA ≺ t) ≺ (x,(x,t))
  As v p → do t' ← match ≺ (Var v,t); match ≺ (p,t')
  StringLit s → matchStringLit ≺ (s,t)
  NumLit n → matchNumLit ≺ (n,t)
  Cons c ps → matchCons (zipWith match) ≺ (c,ps,t)
  Explode c ts → matchExplode
    (proc c' →  match ≺ (c,c'))
    (proc ts' → match ≺ (ts,ts')) ≪ t
```

Listing 1: Generic abstract pattern matching for Stratego.

the abstract semantics of the domain-specific language features in terms of standard language features whose abstract semantics is well-understood already. The generic interpreter is parametric in the term abstraction, such that we can derive different static analyses in a second step by providing different term abstractions. This architecture enables analysis developers to separately tackle the challenge of designing a good term abstraction.

We have developed a generic interpreter for Stratego based on the Sturdy analysis framework [13,12] in Haskell. We explain the full details of generic interpreters and background about Sturdy in Section 3. Here, we only illustrate a small part of the generic interpreter, namely pattern matching.

Listing 1 shows the generic analysis code for pattern matching. We parameterized the pattern-matching function `match` using a type class `IsTerm` as an interface. Pattern matching interacts with the term abstraction to deconstruct terms but implements other aspects generically. In Listing 1, we have highlighted all calls to operations of `IsTerm`; the remaining code is generic. We provide a short notational introduction before delving deeper into the analysis code.

Our approach is based on Sturdy, which requires analysis code to be written in *arrow style* [11]. Like monads, arrows `(c x y)` generalize pure functions `(x → y)` to support side-effects in a principled fashion. For users of our approach, this mostly means that they have to use Haskell's built-in syntax for arrows, as shown in Listing 1. Expression `(proc x → e)` introduces an arrow computation similar to the pure `(λx → e)`. Do notation `(do cmd*)` denotes a sequence of arrow commands, where each command takes the form `(y ← f ≺ x)` or `(f ≺ x)` [18]. Command `(y ← f ≺ x)` calls `f` on `x` and stores the result in `y`; `(f ≺ x)` ignores the resulting value but not the potential side-effect of `f`. For

a more in-depth introduction to arrows, we refer to Hughes's original paper [11] and online resources such as `https://www.haskell.org/arrows`.

The generic analysis code for pattern matching in Listing 1 describes a computation `(c (Pat, t) t)` that is parametric in `c` and `t`, but restricts these types through type-class constraints. Type `t` must implement the term abstraction interface `IsTerm`. Type `c` is an arrow that encapsulates the side-effects of the computation and must at least support environments and exception handling. We use these side-effects to implement pattern variables and backtracking in `match`.

Computation `match` takes a pattern and the matchee (the term to match) as input and yields the possibly refined term as output. For a wildcard pattern, we yield the matchee unchanged. For pattern variables, we look up the variable in the environment and distinguish two cases. If the variable is already bound to `t'`, we require the matchee `t` to be equal to `t'`. If the variable is not bound yet, we insert a binding into the environment. For named subpatterns `(As v p)`, we invoke the code for pattern variables recursively. The remaining four cases delegate to the term abstraction, passing the function for matching subterms as needed. When a pattern match fails, it throws an exception to reset all bound pattern variables.

The generic analysis code for pattern matching captures the essence of pattern matching in Stratego and closely follows Stratego's concrete semantics. In fact, the generic code can be instantiated to retrieve a fully functional concrete interpreter for Stratego. This makes the generic interpreter relatively easy to develop: no analysis-specific code is required. All analysis-specific code resides in instances of interfaces like `ArrowExcept` and `IsTerm`. Sturdy further exploits this to support compositional soundness proofs of analyses [13].

2.2 A Singleton Term Abstraction

We can derive complete Stratego analyses from the generic interpreter by instantiation. Specifically, we need to provide implementations for the type classes it is parameterized over. For standard interfaces like `ArrowExcept` and `ArrowEnv`, we provide reusable abstract semantics. However, the term abstraction `IsTerm` is language-specific and analysis-specific. Thus, this interface needs to be implemented by the analysis developer.

To illustrate the definition of term abstractions, here we develop a singleton analysis for Stratego. The analysis determines if (part of) a program transformation yields a constant output, such that the transformation can be optimized by constant propagation. Note that in this paper we are only concerned with the definition of analyses; the implementation of subsequent optimizations is outside the scope of the paper.

Each term abstraction needs to choose a term representation. For the singleton analysis, we use a simple data type $\widehat{\texttt{Term}}$ with two constructors:

```
data Term^ = Single Term | Any
```

```
instance (ArrowExcept () c, ArrowJoin c, ...) ⇒ IsTerm Term^ c where
  matchString = proc (s,t) → case t of
    Single ct → liftConcrete matchString ≺ (s,ct)
    Any → (returnA ≺ t) ⊔ (throw ≺ ())

  matchNum = proc (i,t) → case t of
    Single ct → liftConcrete matchNum ≺ (i,ct)
    Any → (returnA ≺ t) ⊔ (throw ≺ ())

  matchCons matchSub = proc (c,ps,t) → case t of
    Single (Cons d ts) | c == d && eqLen ps ts → do
      ts' ← matchSub ≺ (ps,map Single ts)
      case allSingle ts' of
        Nothing → returnA ≺ Any
        Just cts → returnA ≺ Single (Cons c cts)
    Single _ → throw ≺ ()
    Any → do matchSub ≺ (ps,replicate (length ps) Any)
             (returnA ≺ t) ⊔ (throw ≺ ())
```

Listing 2: Parts of a singleton term abstraction for Stratego.

A term `Single ct` means that the transformation produces a single concrete Stratego term `ct` of type `Term`. In contrast, `Any` means that the transformation cannot be shown to produce a single concrete term.

Based on such term representation, a term abstraction for Stratego must implement the 10 functions from the IsTerm interface. We show the implementation of four of these functions in Listing 2 that also appeared in Listing 1.

Function matchString in Listing 2 defines a computation that takes a string value `s` and a matchee `t` of type `Term` as input. If `t` denotes a single concrete term, matchString delegates to the concrete string matching semantics using `liftConcrete`. However, if the matchee is `Any`, we cannot statically determine if the pattern match should succeed or fail. Thus, we join ⊔ the two potential outcomes: Either pattern matching succeeds and we return `t` unchanged, or pattern matching fails and we abort the matching by throwing an exception. Function matchNum is analogous to matchString.

Function matchCons distinguishes three cases. The first case checks if matchee `t` denotes a single concrete term with constructor `c` and right number of subterms. If so, we recursively match the subpatterns against the subterms, converted to singletons. Then, if all submatches yielded singleton terms again, we refine the matchee accordingly. The second case occurs when `t` denotes a singleton term but does not match the constructor pattern. In this case, we simply abort. Finally, if `t` is `Any`, we combine the two cases using a list of `Any` terms as subterms. Note that the recursive match on the subpatterns `ps` is necessary to bind pattern variables that may occur.

2.3 Soundness

Our approach drastically simplifies the soundness proof of the abstract interpreter. In particular, by factoring the concrete and abstract interpreter into a generic interpreter, we do not have to worry about soundness of the generic interpreter. Instead, its soundness proof follows by composing the proof of smaller soundness lemmas about its instances [13]. Furthermore, because we instantiate the generic interpreter with sound analysis components for environments, stores and exceptions, we do not have to worry about soundness of these analysis concerns either [12]. All that is left to prove, is the soundness of the term operations.

2.4 Summary

Our approach to developing static analyses for program transformations consists of two steps. First, develop a generic interpreter based on standard semantic components and a parametric term abstraction. Second, define a term abstraction and instantiate the generic interpreter. While the term abstraction is language-specific and analysis-specific, the generic interpreter can be reused across analyses and only needs to be implemented once per transformation language. In the subsequent section, we explain how to develop and instantiate generic interpreters for transformation languages using standard semantic components. Sections 4 and 5.1 demonstrate the development of sophisticated term abstractions.

3 Generic Interpreters for Program Transformations

Creating sound static analyses is a laborious and error-prone process. While there is a rich body of literature on analyzing functional and imperative programming languages, static analysis of program transformation languages is under-explored. Most work in the area of program transformations so far focused on type checking, which considers each rewriting separately and is limited to intra-procedural analysis.

The key enabler of our approach are generic interpreters that can be instantiated with different term abstractions to obtain different analyses. In this section, we demonstrate our approach at the example of Stratego and show how to develop generic interpreters for Stratego. In particular, we show that the features of program transformation languages do *not* require specific analysis code but can be mapped to existing language concepts whose analysis is already well-understood.

3.1 The Program Transformation Language Stratego

Stratego is a program transformation language featuring rich pattern matching, backtracking, and generic traversals [24]. For example, consider the following

```
desugar-type: PairType(t1,t2) → ⟦Pair<~t1,~t2>⟧
desugar-expr: PairExpr(e1,e2) → ⟦new Pair<>(~e1,~e2)⟧

topdown(s) = s; all(topdown(s))        try(s) = s <+ id
main = topdown(try(desugar-type + desugar-expr))
```

Listing 3: A generic traversal for desugaring pair notation.

desugaring of Java extended with pairs [9] in Listing 3. The two rewrite rules of the form 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 a node unchanged if neither rule applies. We also added the definitions of the higher-order functions `topdown` and `try` from the standard library. The built-in primitive `all` takes a transformation and applies it to each direct subterm of the current term. Function `topdown` uses `all` to realize a generic top-down traversal over a term, applying `s` to every node. Function `try` uses left-biased choice `<+` to catch any failure in `s` and to resume with the identity function `id` instead. Furthermore, the Stratego compiler translates the rewrite rules of the form `r : p → t` to transformations `r = ?p; !t`:

```
desugar-type = ?PairType(t1,t2); !ClassType("Pair",[t1,t2])
desugar-expr = ?PairExpr(e1,e2); !NewInstance("Pair",[e1,e2])
```

The translated rule first matches the pattern `p`, binding all pattern variables to the respective subterms and then builds the term `t` using the abstract syntax of Java.

3.2 A Generic Interpreter for Stratego

We demonstrate how to map these language features to standard language concepts and how this enables static analysis of program transformations. To this end, we developed a generic interpreter for Stratego.[2] The generic interpreter is based on a previous Sturdy case study [13] that was never described in detail.

We consider fully desugared Stratego code in our interpreter, ignoring Stratego's dynamic rules. This core Stratego language [23] only contains 12 constructs as defined by the data type `Strat` in Listing 4. We explain these constructs together with their generic semantics, shown in the same listing. The semantics is defined by a function `eval` that accepts a Stratego program and yields a computation of type `(c term term)`, meaning that a Stratego program takes a term as input and yields another term as output. That is, Stratego programs are term transformations as expected. The arrow `c` captures the side-effects of the computation, as explained in Section 2.1.

[2] `https://gitlab.rlp.net/plmz/sturdy/blob/master/stratego/src/GenericInterpreter.hs`

```
data Strat = Match Pat | Build Pat | Id | Seq Strat Strat
  | Fail | GuardedChoice Strat Strat Strat | Scope [String] Strat
  | Call String [Strat] [String] | Let [(String,Strategy)] Strat
  | One Strat | Some Strat | All Strat

eval :: (IsTerm term c, ArrowEnv String term c, ArrowExcept () c,
         ArrowFix c, ...) ⇒ Strat → c term term
eval = fix $ λev strat → case strat of
  Match pat → proc t → match ≺ (pat,t)
  Build pat → proc _ → build ≺ pat

  Id → proc x → returnA ≺ x
  Seq s1 s2 → proc t1 →
    do t2 ← ev s1 ≺ t1; t3 ← ev s2 ≺ t2; returnA ≺ t3
  Fail → proc _ → throw ≺ ()
  GuardedChoice s1 s2 s3 → try (ev s1) (ev s2) (ev s3)

  Scope vars s → scoped vars (ev s)
  Call f ss ts → proc t → do
    senv ← readStratEnv ≺ ()
    case Map.lookup f senv of
      Just (Closure s@(Strat _ ps _) senv') → do
        args ← mapA lookupOrFail ≺ ts
        scoped ps (invoke ev) ≪ (s, senv', ss, args, t)
      Nothing → failString ≺ "Cannot␣find␣strat"
  Let bnds body → let_ bnds body eval'

  One s → mapSubterms (one (ev s))
  Some s → mapSubterms (some (ev s))
  All s → mapSubterms (all (ev s))

scoped vars f = proc t → do
  oldEnv ← getEnv ≺ ()
  deleteEnvVars ≺ vars
  finally (proc (t,_) → f ≺ t)
          (proc (_,oldE) → restoreEnvVars vars ≺ oldE)
    ≺ (t, oldEnv)
```

Listing 4: Generic interpreter for Stratego.

The first two core Stratego constructs deconstruct and construct terms. A `(Match pat)` transformation is based on a term pattern `pat`, which it matches against the input term `t`. Function `match` from Listing 1 implements the actual pattern matching, as we have discussed in Section 2. Recall that `match` binds pattern variables in the environment as a side-effect and throws an exception if the pattern match fails. We will see shortly how these side-effects are supported by the generic interpreter. A `(Build pat)` transformation is the dual of match: it constructs a new term according to the pattern, filling in information from the environment in place of pattern variables.

The next four core Stratego constructs handle control-flow. The identity transformation `Id` returns the input term unchanged. A sequence `(Seq s1 s2)`

of transformations `s1` and `s2` pipes the output of `s1` into `s2`. The `Fail` transformation never succeeds and always throws an exception using `throw`, which we also used to indicate failed pattern matches. To catch such exceptions, core Stratego programs can use guarded choice, written `(s1 < s2 + s3)` in Stratego notation. Guarded choice runs `s3` if `s1` fails (throws an exception) and `s2` otherwise. We implemented guarded choice using the `try` function. Like `throw`, `try` is declared in the `ArrowExcept` interface and allows us to catch exceptions triggered by `throw`. There are two things to note here:

- The implementation of `throw` and `try` are not specific to Stratego and are provided as sound reusable analysis components [12] by the standard library of Sturdy. We are effectively mapping Stratego features to these pre-defined features of Sturdy.
- We can choose how exceptions affect the variables bound during pattern matching. For Stratego, we need exceptions to undo variable bindings in order to correctly implement backtracking. However, in other languages we may want to retain the state of a computation even after an exception was thrown.

The next three constructs handle scoping, strategy calls, and local strategy definitions. We discuss the first two of these in some detail. Stratego's scoping is somewhat unconventional, because Stratego has explicit scope declarations and environments follow store-passing style. Variables listed in a scope declaration are lexically scoped as usual, but other variables can occur in the environment and must be preserved. We use function `scoped` (at the bottom of Listing 4) to implement this scoping. First, we unbind the scoped variables from the current environment to allow pattern matching to bind them afresh. Second, after the scoped code finishes, we restore the bindings of scoped variables from the old environment while retaining other bindings from the current environment unchanged. Scoping also occurs when calling a strategy. To evaluate a call, we first find the strategy definition, then lookup the term arguments `ts` in the current environment, and then invoke the strategy using `scoped` for the term parameters `ps`.

The final three constructs are generic traversals that use `mapSubterms` to call `one`, `some`, or `all` on the subterms of the current input term. Function `mapSubterms` is part of the `IsTerm` interface and thus analysis-specific because depends on the term representation. Functions `one`, `some`, or `all` are part of the generic interpreter and ensure that, respectively, exactly one, at least one, or all of subterms are transformed by the given strategy `s`. This way our generic interpreter separates term-specific operations from operations that can be defined generically.

3.3 The Term Abstraction

At this point, all that it takes to define a Stratego analysis is to implement the `IsTerm` interface for a new term abstraction. The rest of the analysis is given by the generic interpreter and reusable functionality from the Sturdy library.

```
class Arrow c ⇒ IsTerm term c where
  matchString     :: c (String,term) term
  matchNum        :: c (Int,term) term
  matchCons       :: c ([p],[term]) [term] →
                       c (String,[p],term) term
  matchExplode    :: c term term → c term term → c term term

  buildString     :: c String term
  buildNum        :: c Int term
  buildCons       :: c (String,[term]) term
  buildExplode    :: c (term,term) term

  equal           :: c (term,term) term
  mapSubterms     :: c [term] [term] → c term term
```

Listing 5: An interface for operations on terms.

The generic interpreter described in the previous section crucially relies on the term abstraction. In particular, pattern matching, term construction, and generic traversals must inspect or manipulate terms. In Section 2 we have seen how `match` used term operations and how we could implement these for the singleton term abstraction. Here we show the complete interface for term abstractions.

Stratego terms are strings, numbers, or constructor terms:

```
data Term = Cons String [Term] | StringLit String | NumLit Int
```

Our interface must at least provide operations to match and construct such terms. In addition, we must support Stratego's generic traversals and explode patterns. Note that Stratego represents lists using constructors `Cons` and `Nil`:

```
Cons "Cons" [NumLit 1, Cons "Cons" [NumLit 2, Cons "Nil" []]]
```

We designed an interface for term abstractions of Stratego terms that requires only 10 operations. Listing 5 shows the corresponding type class. The interface contains four functions for pattern matching, four functions for term construction, one equality function, and one function to map subterms.

We have discussed the functions for pattern matching Section 2 already. Function `matchCons` takes a function for matching subterms against subpatterns. Function `matchExplode` takes functions for matching the constructor name and the subterms. The functions for term construction are straightforward. While function `buildCons` takes a String and a list of terms, function `buildExplode` takes two terms. The first of these terms must be a string term, the second one must represent a list of terms. Finally, we require functions for checking the equality of two terms and for mapping a function over a term's subterms. This last function enables generic traversals as shown in Listing 4.

Our interface for term abstractions can be instantiated in various ways by defining instances of the type class. We have shown an instance for the singleton term abstraction in Listing 2 and will describe further term abstractions in

the upcoming sections. But it is worth noting that the interface can also be instantiated for concrete Stratego terms:

```
instance ... ⇒ IsTerm Term c where ...
```

This concrete term instance allows us to run the generic interpreter as a concrete Stratego semantics. This is not only great for testing the generic interpreter against a reference implementation of Stratego, but also crucial for proving the soundness of term abstractions against the concrete semantics.

To summarize, we implemented the Stratego language semantics as a generic interpreter based on a few term operations only. The generic interpreter maps many aspects of Stratego language to standard language concepts such as environments and exceptions. For these language concepts, we reuse the abstract semantics found in the Sturdy standard library. In the end, to design and implement a new analysis for Stratego, all it takes is a new term abstraction. We exploit this reduction of effort in the next two sections, where we develop two novel static analyses for Stratego by defining term abstractions.

4 Sort Analysis

In this section, we define an inter-procedural sort analysis for Stratego. The analysis checks if a program transformation generates well-formed programs and to which sort the program belongs. That is, we implement a term abstraction where we choose to represent terms through their sort.

4.1 Sorts and Sort Contexts

We describe the sorts of Stratego terms by the following Haskell datatype:

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

Sort `Lexical` represents string values, `Numerical` represents numeric values. We use `(Sort s)` to represent named sorts such as `(Sort "Exp")`. We further include sorts for representing Stratego's lists, tuples, and option terms. Finally, `Bottom` represents the empty set of terms and `Top` represents all terms (also ill-formed ones). This means, we can guarantee a term is well-formed if its sort is not `Top`.

To associate terms to sorts, we parse the declaration of constructor signatures that are part of any Stratego program. Typically, these declarations are automatically derived from the grammar of the source and target language.

```
Num : Int → ArithExp
Add : ArithExp * ArithExp → ArithExp
    : ArithExp → PythonExp
```

Each line declares a constructor, the sorts of its arguments and the generated sort. We allow overloaded constructor signatures as long as they generate terms

of the same sort. That is, if $c : s_1 \dots s_m \to s \in \Gamma$ and $c : s'_1 \dots s'_n \to s' \in \Gamma$, then $s = s'$.

The third signature declares that any term of sort `ArithExp` should also be considered a term of sort `PythonExp`. This is the result of injection production in the grammar and effectively declares a subtype relation `ArithExp <: PythonExp`. Dealing with subtyping correctly is one of the major challenges of developing a sort analysis. Thanks to our separation of concerns, we can fully focus on that challenge here.

We collect all constructor signatures and the subtyping relation in a sort context:

```
type Sig = ([Sort], Sort)
data Context = Context { sorts :: Map Sort [(String,Sig)],
                         subtypes :: SubtypeRelation }
```

Since we require the context when operating on sorts, we actually represent terms abstractly as a pair `(Sort,Context)`. However, all terms refer to the same context and the context never changes. To simplify the presentation in this paper, we assume the context is globally known and terms are represented by `Sort` alone.

4.2 Abstract Term Operations

In the remainder of this section, we explain how to implement the term abstraction for our sort analysis. To this end, we have to provide an instance of type class `IsTerm` as shown in Listing 6. We only show the code for lists and user-defined constructor and omit the other cases for tuples and optionals for brevity.

As a warm-up, consider operation `buildString` that yields sort `Lexical` independent of the string literal. When matching a string against sort `s` in `matchString`, the match can only succeed if `Lexical` terms may be part of `s` terms. Otherwise the match must fail.

Arguably the most interesting part of the term abstraction is building and matching constructor terms. Let's start with operation `buildCons`, which obtains the constructor name `c` and the list of subsorts `ss`. In Stratego, list, tuple, and optional terms use reserved constructor names. We include one case for each reserved constructor to generate the appropriate sort. For example, constructor `Nil` can be applied to an empty argument list to generate an empty list. This list has sort `(List Bottom)`. Constructor `Cons` generates a compound term that has sort list if the second argument was a list. The sort of the resulting list is the least super-sort ($\sqcup$) of the new head list and the tail. The empty constructor `""` generates tuples; `None` and `Some` generate optional terms.

The last case of `buildCons` handles user-defined constructor symbols `c`. We use `(constrSigs c)` to look up the signatures `(ss',t)` of `c` from the sort context. We only retain those signatures that can accept the constructor arguments `ss`. Finally, we collect all result sorts `t` and compute their greatest lower bound.

```
instance (ArrowExcept () c, ArrowJoin c, ...)⇒IsTerm Sort c where
  buildString = proc _ → returnA ≺ Lexical

  matchString = proc (_,s) → if subtype Lexical s
    then (returnA ≺ s) ⊔ (throw ≺ ()) else throw ≺ ()

  buildCons = proc (c, ss) → returnA ≺ case (c, ss) of
    ("Nil",[]) → List Bottom
    ("Cons",[a,s]) | subtype (List Bottom) s → List a ⊔ s
    _ → ⨆(Top : [t | (ss',t) ← constrSigs c, ss ⊑ ss'])

  matchCons matchSubs = proc (c,ps,s)→ case (c,ps)
    ("Nil",[]) → if subtype (List Bottom) s
      then (buildCons≺("Nil",[])) ⊔ (throw≺()) else throw≺()
    ("Cons",[hd,tl]) → if subtype (List Bottom) s
      then do let subterms = [getListElem s, s]
              ss ← matchSubs ≺ ([hd,tl],subterms)
              (buildCons ≺ ("Cons",ss)) ⊔ (throw ≺ ())
      else throw ≺ ()
    _ → ⨆(proc (c',ss) → if c == c' && length ss == length ps
      then do ss' ← matchSubs ≺ (ps,ss); cons ≺ (c,ss')
      else throw ≺ ()) ≪ constructorsOfSort s

  mapSubterms f = proc s → do ⨆(proc (c,ts) →
     do ts' ← f ≺ ts buildCons ≺ (c,ts'))
        ≺ constructorsOfSort s
```

Listing 6: Abstract term operations for the sort analysis.

If none of the signatures matches, we return sort `Top`. For example, consider the call:

```
buildCons ≺ ("While",[Sort "Exp",Sort "Block"])
```

If the signature of `While` is `(Exp * Block → Stmt)`, we obtain `Sort "Stmt''` as result. If the signature is instead declared as `(Exp * Exp → Stmt)`, we obtain `Top` because the constructed term is ill-formed (unless `Block` is a sub-sort of `Exp`).

Operation `matchCons` is quite complex, although all cases for reserved constructors follow the same pattern:

1. We check if the sort of the current term `s` is compatible with the matched constructor. For example, a match against `Nil` can only succeed if the sort is a list.
2. We retrieve the subterm sorts if any. For example, for `Cons` we have two subterms: the head element and the tail list. Auxiliary function `getListElem` carefully finds all possible list elements, taking subtyping into account.
3. We match the subterms against the subpatterns, yielding refined subterms `ss`.
4. We refine the current term by calling `buildCons` on the refined subterms and the matched constructor. Since matching may always fail, we join the result with a call to `throw`.

The last case of `matchCons` again handles user-defined constructor symbols `c`. We use `constructorsOfsort s` to obtain all constructors `c'` and their argument types `ss`. If the constructor has the required name and the right number of arguments, then the corresponding match might succeed. We match the subterms and refine the current term as in the other cases, but the we compute the least upper bound over all possible results. For example, when we match a constructor `Add` against sort `Exp`, we would lookup all constructors that generate sort `Exp`. For `(Add : Exp * Exp → Exp)` the match can succeed, but for `(Var : Lexical → Exp)` the match must fail. The join operator merges the results to compute a sound approximation.

Lastly, we show the code of `mapSubterms`, which needs to retrieve the current subterms as a list and pass them to `f`. However, sorts do not directly point out their subterms. Again we use `constructorsOfsort s` to retrieve the sorts of subterms indirectly by finding all constructors of the current sort and taking their parameter lists. For example, if we call `mapSubterms` with sort `"Exp"`, then computation f will be called on `[Sort "Exp", Sort "Exp"]` for constructor `Add` and on `[Lexical]` for constructor `Var`.

To summarize, in this section we defined a sort analysis for Stratego, simply by designing a sort term abstraction which implements the `IsTerm` interface. The rest of the analysis we get for free from the generic interpreter and reusable analysis code. As the reader probably noticed, the term abstraction for sorts is fairly complex in its own right. Being able to focus on the term abstraction without considering other analysis aspects was crucial.

4.3 Sort Analysis and Generic Traversals

In this subsection, we showcase the inter-procedurality of our sort analysis by analyzing generic traversals. A generic traversal traverses a syntax tree independent of its shape and transforms the visited nodes. Statically assigning types to a generic traversal is notoriously difficult, because the type needs to summarize all changes the traversal does to the entire tree. In this subsection, we will illustrate how our inter-procedural sort analysis can support some generic traversals, before refining our analysis further in the subsequent section.

Consider the trace of the sort analysis (Figure 1) of the pair desugaring from Section 3.1. The trace starts in the `main` function with an input term of sort `Expr`. The `main` function calls `topdown`, which calls `try(D)`, which calls the desugaring rules `desugar-type + desugar-expr`. The rule `desugar-expr` either yields a term of sort `Expr` or fails because the pattern `PairExpr(...)` matches some but not all terms of sort `Expr`. Furthermore, the rule `desugar-type` definitely fails because no terms of sort `Expr` match the pattern `PairType(...)`. Even though one of the rules failed, the call `try(D)` produces a successful result by applying the input term to the identity transformation. The function `topdown` then passes the resulting term of sort `Expr` to the generic traversal `all(...)`. Since we know the sort of the current term, we enumerate all relevant constructors and the sorts of their direct subterms and recursively analyze the desugaring for them. In the example trace of Figure 1, we consider three subterm sorts of `Expr`. The second

```
desugar-type: PairType(t1,t2) → ⟦Pair<~t1,~t2>⟧
desugar-expr: PairExpr(e1,e2) → ⟦new Pair<>(~e1,~e2)⟧

topdown(s) = s; all(topdown(s))    try(s) = s <+ id
main = topdown(try(desugar-type + desugar-expr))
```

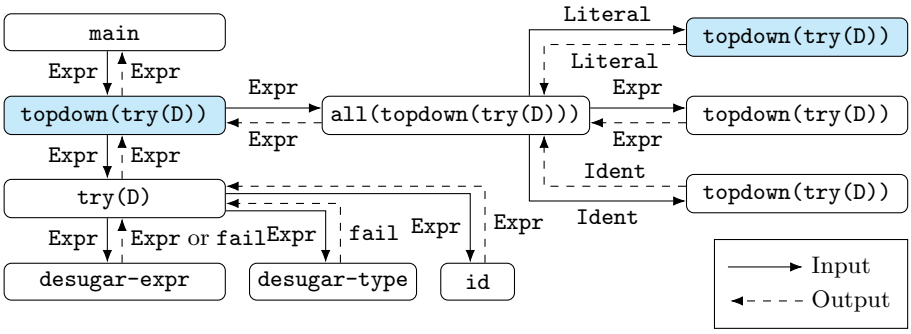

Fig. 1: A simplified trace of the sort analysis of the pair desugaring, where we abbreviate `desugar-type + desugar-expr` with `D`.

and third recursive call to `topdown(try(D))` resolve easily, whereas the first recursive call would end up in a cycle (shaded nodes in Figure 1). To this end, we use a fixpoint algorithm with widening to ensure that the analysis terminates.

The example shows why it is hard to analyze the type of a generic traversal: For different input sorts, a generic traversal might produce different output sorts. Therefore, our sort analysis reanalyzes a generic traversal for each input sort, instead of assigning a fixed type like a type checker would do.

The example we considered here is a special case of generic traversals, known as type-preserving. A generic traversal is type-preserving if the sort of the input and output term are the same at every node. However, some generic traversals change the sort of the input term. The sort analysis of this section is not capable of analyzing such type-changing generic traversals. To this end, we require a more precise analysis, which we develop in the following section.

5 Locally Ill-Sorted Sort Analysis

Many program transformations, like a compiler, translate terms from one sort to terms of another sort. When these program transformations use generic traversals, they produce mixed intermediate terms, which contain subterms of the input sort *and* subterms of the output sort. Because mixed intermediate terms are not well-sorted, these program transformations are challenging to type check.

For example, consider the traversal in Figure 2 that translates Boolean expressions into numeric expressions in a bottom-up fashion. The boolean expression `And(True(),False())` is transformed in two steps:

```
And(True(),False()) ⇝ And(1,0) ⇝ Min(1,0)
```

```
encode: True        → 1                    : Int → NExp
encode: False       → 0              Max : NExp * NExp → NExp
encode: And(e1,e2) → Min(e1,e2)      Min : NExp * NExp → NExp
encode: Or(e1,e2)  → Max(e1,e2)
                                          : Bool → BExp
bottomup(s) = all(bottomup(s)); s    And : BExp * BExp → BExp
main = bottomup(encode)              Or  : BExp * BExp → BExp
```

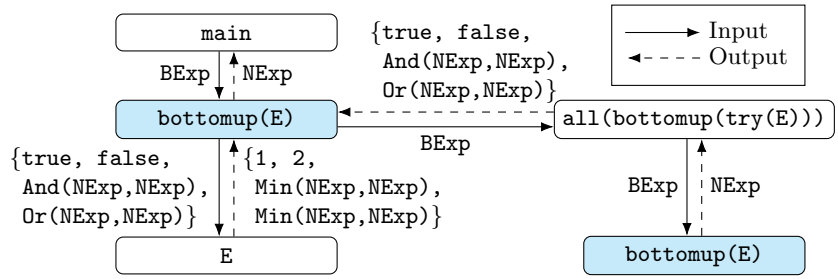

Fig. 2: The top of the figure contains a type-changing generic traversal that translates boolean to numeric expressions. The bottom contains the analysis trace of the transformation, where we abbreviate `encode` with `E`.

Even though the input term `And(True(),False())` is a valid boolean expression and the output term `Min(1,0)` a valid numeric expression, the transformation creates an intermediate term `And(1,0)`, which is ill-sorted. The sort analysis of the previous section is only able to check transformations which produce well-sorted terms and therefore cannot handle this example. To analyze this example, we need a more precise sort analysis that can represent ill-sorted terms, which we develop in the remainder of this section

5.1 Term Abstraction for Ill-Sorted Terms

The key idea is to use a term abstraction which can represent terms with well-sorted leafs and an possible ill-sorted prefix, such as `And(NumExp,NumExp)`. This abstract term represents all terms with `"And"` as top-level constructor and two numeric expressions as subterms. We implement this term abstraction with the following Haskell type:

```
data Term = Sorted Sort | MaybeSorted (Set (String,[Term]))
```

The case `Sorted s` represents well-sorted terms that belong to sort `s`, and the case `MaybeSorted` represents terms with an possibly ill-sorted prefix. For example, this datatype allows us to represent the ill-sorted term `And(1,0)` with the abstract term

```
MaybeSorted [("And",[Sorted "NExp",Sorted "NExp"])].
```

5.2 Abstract Term Operations

We develop an analysis for Stratego by implementing the term operations with the term abstraction from above. We only discuss the `matchCons` and `buildCons`

```
matchCons matchSub = proc (c,ps,t) → case t of
  MaybeSorted cs → matchCons' ≺(c,ps,cs)
  Sorted s → Sort.matchCons matchSub≺(c,ps,lookupSort' ctx s)
 where
  matchCons' = proc (c,ps,cs) → ⨆(proc (c',ss) →
    if c == c' && length ss == length ps
      then do ss' ← mapSub ≺ (ps,ss); cons ≺ (c,ss')
      else throw ≺ ()) ≪ cs

buildCons = proc (c,ts) → returnA ≺ MaybeSorted [(c,ts)]

widening :: Context → Int → Term → Term → Term
widening ctx k cs1 cs2
  | k == 0 = Sorted (typecheck ctx (cs1 ⊔ cs2))
  | otherwise =
     MaybeSorted (zipSubterms (termWidening ctx (k-1)) cs1
    cs2)
  where typecheck :: Context → Term → Sort
```

Listing 7: Abstract term operations for the locally ill-sorted sort analysis.

operations (Listing 7), because the remaining functions are similar to the operations of the sort analysis.

The `matchCons` operation first matches on the term representation and in both cases calls the `matchCons'` helper function, which compares the constructors, arity and subterms. The `lookupSort'` function, similar to Listing 6, looks up all constructor signature for a sort, but additionally converts the signatures to abstract terms. This `matchCons` operation is more than the `matchCons` of the sort analysis, because we may know the top-level constructor of the term. This improved precision results in more pattern matches which *unconditionally* succeed or fail.

In contrast to the sort analysis, the `buildCons` operation in Listing 7 does not check if the constructor and its subterms belong to a valid sort. Instead, it constructs a new abstract term, which may or may not be well-sorted. The type checking of this term is then delayed until a later point.

With these definitions, the analysis would be able to check some type-changing generic traversals, however, it might not terminate because the abstract terms might grow arbitrarily large. To avoid this problem, we reduce the size of abstract terms by type checking their subterms. For example, we can type check the immediate subterms of `Or(And(1,0),1)` to obtain the abstract term `Or(`$\top$`,NumExp)`. In the new term, the sort $\top$ indicates the type checking of `And(1,0)` failed and the term is ill-sorted. We use this technique in a widening operator [7] that ensures that the analysis terminates. The operator simply type checks all subterms deeper than a certain limit k, such that the resulting terms are not deeper than k.

5.3 Analyzing Type-Changing Generic Traversals

In the remainder of this section, we discuss how the analysis of this section checks type-changing generic traversals. To this end, we discuss an analysis trace of the example at the beginning of this of this section (Figure 2).

The trace in Figure 2 shows only the final fixpoint iteration (earlier iterations produce subsets of the sets shown in the trace). It starts with the analysis of the `main` function with the boolean expression sort `BExp`, which is then passed to `bottomup(E)`. In contrast to the top-down traversal, the bottom-up traversal first traverses with `all(bottomup(E))` over the subterms of boolean expressions and replaces them by numeric expressions, e.g., `And(NExp,NExp)`. The resulting set of ill-sorted terms is then passed to the rewrite rule `E`. The rule `E` then replaces each top-level boolean constructor with a numeric constructor without touching the subterms. All terms in the resulting set are now well-typed and `bottomup(E)` applies the widening operator to reduce this set to `NExp`.

In summary, we defined an advanced sort analysis, which can represent ill-sorted terms. This analysis is able to check type-changing generic traversals, which produces ill-sorted intermediate terms.

6 Related Work

Transformation languages like Stratego [10] and PLT Redex [17] have a dynamic type checker for syntactic well-formedness. While dynamic type checking supports generic traversals, it does not help developers of transformations to understand the code. In contrast, we developed a static analysis such that program transformations can be checked before running them.

Other program transformation languages like Ott [22], Maude [5], Tom [2] and Rascal [14] use static type checking to ensure syntactic well-formedness. However, these languages do not support or struggle to statically check arbitrary generic traversals. Ott is a language for specifying rewrite systems and exporting them to proof assistants such as Coq or Isabelle. However, it does not support generic traversals. Maude is a language for specifying rewrite systems in membership equational logic. However, it implements generic traversals with reflection and hence cannot statically check their type. Tom and Rascal are statically typed transformation languages with support for type-preserving generic traversals. However, they do not support type-changing generic traversals. We explained in Section 5 why conventional static type checkers cannot analyze type-changing generic traversals: these traversals produce intermediate terms which are ill-sorted. In this work, we aim to analyze type-preserving as well as type-changing generic traversals. We solve this problem by defining a static analysis which can represent terms with a finite ill-sorted prefix. In contrast to a conventional type checking, this term abstraction is more precise than regular types, but requires computing a fixed point.

Lämmel distinguishes "type-preserving" from "type-unifying" generic traversals [15], as realized in Scrap-Your-Boilerplate [19]. A unifying generic traversal

is a fold over the term that yields a value of the same "unified" type at each node. These kinds of generic traversals are easier to type statically, however, not all generic traversals fit in one of these two typing schemes. For example, a generic traversal that translates code from one language to another is neither type-preserving nor type-unifying. Rather than developing additional specialized traversal styles, our paper aims to support static analysis for arbitrary generic traversals.

Most closely related to our work, Al-Sibahi et al. present an abstract interpreter of a subset of Rascal, including generic traversals [1]. Al-Sibahi et al. use inductive refinement types as abstract domain. The main difference of our work is that we separated analysis-independent concerns (the generic interpreter) from analysis-specific concerns (the instances). This way we can develop different analyses for program transformations with relatively little effort. Furthermore, it also simplifies the analysis definition, because most of the language complexity is captured in the generic interpreter. Lastly, our work is based on the well-founded theory of compositional soundness proofs [13] provided by the Sturdy framework. This allows us to verify that soundness of analyses more easily, as we only need to prove that the instances are sound.

CompCert [16] is a formally verified C compiler. The compiler guarantees that the compiled program has the same semantics as the input program. To this end, each program transformation in the compiler passes has to preserve the semantics of the transformed program. While CompCert focuses on the semantics of the transformed program, the static analyses for program transformations in this work have to satisfy a different correctness property. Soundness of these static analyses guarantees that the analyses results overapproximate which programs can be generated by a program transformation. However, soundness does not give any guarantees about the semantics of the transformed program. In the future, we aim to develop more precise analyses for program transformation languages that allow us to draw conclusion about the semantics of transformed programs.

7 Conclusion

To summarize, in this work, we presented a systematic approach to designing static analyses for program transformations. Key of our approach is to capture the core semantics of the program transformations with a *generic interpreter* that does not refer to any analysis-specific details. This lets the analysis developer focus on designing a good abstraction for programs. We demonstrated the usefulness of our approach by designing three analyses for the program transformation language Stratego. Our sort analyses are able to check the well-sortedness of type-preserving and even type-changing generic traversals.

Acknowledgements

This research was supported by DFG grant "Evolute". We thank André Pacak and Tamás Szabó who provided helpful feedback on the introduction.

References

1. Al-Sibahi, A.S., Jensen, T.P., Dimovski, A.S., Wasowski, A.: Verification of high-level transformations with inductive refinement types. CoRR (2018)
2. Balland, E., Brauner, P., Kopetz, R., Moreau, P., Reilles, A.: Tom: Piggybacking rewriting on java. In: Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings. pp. 36–47 (2007)
3. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986. pp. 152–161 (1986)
4. Chen, Y., Gansner, E.R., Koutsofios, E.: A C++ data model supporting reachability analysis and dead code detection. In: Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings. pp. 414–431 (1997)
5. 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)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252 (1977)
7. 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)
8. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: 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 (1995)
9. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: 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 (2011)
10. Erdweg, S., Vergu, V., Mezini, M., Visser, E.: Modular specification and dynamic enforcement of syntactic language constraints. In: Proceedings of International Conference on Modularity (AOSD). pp. 241–252. ACM (2014)
11. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000)
12. Keidel, S., Erdweg, S.: Compositional soundness proofs of abstract interpreters. PACMPL **3**(OOPSLA) (Oct 2019)
13. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. PACMPL **2**(ICFP), 72:1–72:26 (Jul 2018)
14. 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 (2009)

15. Lämmel, R.: Typed generic traversal with term rewriting strategies. Logic and Algebraic Programming **54**(1-2), 1–64 (2003)
16. Leroy, X., et al.: The compcert verified compiler. Documentation and user's manual. INRIA Paris-Rocquencourt **53** (2012)
17. Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A visual environment for developing context-sensitive term rewriting systems. In: Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings. pp. 301–311 (2004)
18. Paterson, R.: A new notation for arrows. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001. pp. 229–240 (2001)
19. Peyton Jones, S.L., Lämmel, R.: Scrap your boilerplate. In: Asian Symposium on Programming Languages and Systems (APLAS). LNCS, vol. 2895, p. 357. Springer (2003)
20. Pierce, B.C.: Types and programming languages. MIT Press (2002)
21. Salcianu, A., Rinard, M.C.: Purity and side effect analysis for java programs. In: Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings. pp. 199–215 (2005)
22. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. Functional Programming **20**(1), 71–122 (2010)
23. Visser, E., Benaissa, Z.: A core language for rewriting. Electr. Notes Theor. Comput. Sci. **15**, 422–441 (1998)
24. Visser, E., Benaissa, Z., Tolmach, A.P.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998. pp. 13–26 (1998)
25. Xie, Y., Chou, A., Engler, D.R.: ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003. pp. 327–336 (2003)