FlowSpec

A declarative specification language for intra-procedural flow-Sensitive data-flow analysis

Smits, Jeff; Wachsmuth, Guido; Visser, Eelco

DOI
10.1016/j.cola.2019.100924

Publication date
2020

Document Version
Final published version

Published in
Journal of Computer Languages

Journal of Computer Languages 57 (2020) 100924

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.editorialmanager.com/cola/default.aspx

FLOWSPEC: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis

Jeff Smits*[,a], Guido Wachsmuth[b], Eelco Visser[a]

[a] Programming Languages Research Group, Delft University of Technology, Van Mourik Broekmanweg 6, XE Delft 2628, the Netherlands
[b] Oracle Labs, Prime Tower, Floor 17, Hardstrasse 201, Zürich 8005, Switzerland

HIGHLIGHTS

- Data-flow analysis is the static analysis of programs to estimate their approximate run-time behavior or approximate intermediate run-time values. It is an integral part of modern language specifications and compilers. In the specification of static semantics of programming languages, the concept of data-flow allows the description of well-formedness such as definite assignment of a local variable before its first use. In the implementation of compiler back-ends, data-flow analyses inform optimizations.
- Data-flow analysis has an established theoretical foundation. What lags behind is implementations of data-flow analysis in compilers, which are usually ad-hoc. This makes such implementations difficult to extend and maintain. In previous work researchers have proposed higher-level formalisms suitable for whole-program analysis in a separate tool, incremental analysis within editors, or bound to a specific intermediate representation.
- In this paper, we present FlowSpec, an executable formalism for specification of data-flow analysis. FlowSpec is a domain-specific language that enables direct and concise specification of data-flow analysis for programming languages, designed to express flow-sensitive, intra-procedural analyses.
- We define the formal semantics of FlowSpec in terms of monotone frameworks. We describe the design of FlowSpec using examples of standard analyses. We also include a description of our implementation of FlowSpec.
- In a case study we evaluate FlowSpec with the static analyses for GreenMarl, a domain-specific programming language for graph analytics.

ARTICLE INFO

MSC:
68N15

ABSTRACT

Data-flow analysis is the static analysis of programs to estimate their approximate run-time behavior or approximate intermediate run-time values. It is an integral part of modern language specifications and compilers. In the specification of static semantics of programming languages, the concept of data-flow allows the description of well-formedness such as definite assignment of a local variable before its first use. In the implementation of compiler back-ends, data-flow analyses inform optimizations.

Data-flow analysis has an established theoretical foundation. What lags behind is implementations of data-flow analysis in compilers, which are usually ad-hoc. This makes such implementations difficult to extend and maintain. In previous work researchers have proposed higher-level formalisms suitable for whole-program analysis in a separate tool, incremental analysis within editors, or bound to a specific intermediate representation.

In this paper, we present FLOWSPEC, an executable formalism for specification of data-flow analysis. FLOWSPEC is a domain-specific language that enables direct and concise specification of data-flow analysis for programming languages, designed to express flow-sensitive, intra-procedural analyses. We define the formal semantics of FLOWSPEC in terms of monotone frameworks. We describe the design of FLOWSPEC using examples of standard analyses. We also include a description of our implementation of FLOWSPEC.

In a case study we evaluate FLOWSPEC with the static analyses for GREEN-MARL, a domain-specific programming language for graph analytics.

* Corresponding author.
E-mail addresses: j.smits-1@tudelft.nl (J. Smits), guido.wachsmuth@oracle.com (G. Wachsmuth), e.visser@tudelft.nl (E. Visser).

https://doi.org/10.1016/j.cola.2019.100924
Received 11 August 2019; Accepted 20 September 2019
Available online 23 November 2019

1. Introduction

Nielson et al. define program analysis as follows: Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer [1, p. 1]. Data-flow analysis can answer questions such as if and when data in a variable is accessed, or if certain invariants hold on the data. Data-flow analyses are used to provide static guarantees in the form of compiler warnings and errors, to inform optimizations, to identify security problems, or problematic code style.

1.1. Uses of data-flow analysis

Data-flow analyses may be part of the static semantics of a language. For example, in Java a final field in a class must be initialized for an object of that class by the end of its construction [2, ch. 16]. Since constructor code can have conditional control-flow, a data-flow analysis is necessary to check that all possible execution paths through constructors actually assign a value to the final field. For another example, the compiler for Rust gives warnings on code paths that are unreachable [3].

Data-flow analyses are commonly used to inform optimizations in compilers. Live variables analysis provides information on which variables will be used with their current value, which can be used by a form of dead code elimination called *dead store elimination* [4, p. 24]. This optimization removes assignments to variables which are not observed. Available expressions analysis identifies expressions that have already been computed, which can be used for *common subexpression elimination*.

In some compilers, and in separate tools, data-flow is used to identify security problems. A common approach is taint analysis, which can analyze where data from relevant sources flow. For example, a source of data could be untrusted data from user input. User input should not be used directly in the text of an SQL query, as this opens the possibility of SQL injections.

Data-flow analysis is also applied in code style tools that check for code patterns which are hazards to maintenance or likely to be a logic error. Examples are analyses such as a switch case in Java which has some code, but then falls through to the next case. Although a case that directly falls through is likely intentional, one that has some code may be missing a break statement. Another style lint, as these analyses are often called, is the definition of a non-final variable that is only assigned once. Both of these lints are part of the CheckStyle [5] tool for Java.

1.2. Implementation of data-flow analysis

Data-flow analyses are important for the specification and implementation of programming languages and domain-specific languages (DSLs). However, they are expensive to implement, especially in a general purpose programming language. The compiler for GREEN-MARL [6], a graph analytics DSL from industry, requires more than 2000 lines of C++ code for a data-dependence analysis that takes the domain concepts of the language into account. Since DSLs typically have a relatively small audience, this is reflected in their development team size. The implementation cost of even the most common data-flow analyses can become prohibitive in such a situation.

Language workbenches aim to facilitate high-level language definition and generation of implementations, thereby improving the situation for DSL development. For example, the Spoofax language workbench [7] provides declarative meta-languages for the concise specification of a programming language. An SDF3 [8] specification is used by Spoofax to generate a parser. An NaBL2 [9] specification of the static semantics of the language is used to generate a type checker.

The goal of this work is to provide the same benefits of concise, executable specification for data-flow analysis. In this paper, we present FLOWSPEC, a specification language for intra-procedural, flow-sensitive data-flow analysis. FLOWSPEC is integrated in the Spoofax language workbench and makes use of the provided ecosystem. The analysis that is generated from FLOWSPEC consumes analyzed abstract syntax trees, and turns these trees into a control-flow graph using the control-flow rules. The control-flow graph is then used as input for data-flow analysis, based on the data-flow rules in a FLOWSPEC specification. When the data-flow analysis requires names or types, these can be referenced directly in the specification.

We evaluate FLOWSPEC with specifications of analyses, and we present case studies in static analysis definitions for GREEN-MARL, an industrial DSL for high performance, concurrent graph analytics, and STRATEGO, a term transformation language.

In summary, the contributions of this paper are:

- The language design of FLOWSPEC, a language parametric, domain-specific language for the definition of intra-procedural, flow-sensitive data-flow analysis.
- The formal semantics of FLOWSPEC in terms of Monotone Frameworks, a solid mathematical foundation that has been used for decades for sound approximation of data-flow information beyond sets.
- The implementation of FLOWSPEC, including the integration into the Spoofax language workbench, a fixed-point solving algorithm, and an adapted Strongly Connected Component (SCC) algorithm with extra ordering guarantees within the SCCs. The use of SCCs and their ordering is not novel, but we are not aware of a published algorithm that gives this directly.
- The evaluation of FLOWSPEC on the GREEN-MARL graph analytics DSL, which shows that the language can concisely and cleanly express analyses separately from the definition of transformations.
- The evaluation of FLOWSPEC on the STRATEGO term transformation language, which shows that the language can express interesting non-standard analyses on more languages than a typical imperative language.
- The performance evaluation of FLOWSPEC on different size STRATEGO strategies, which show that the speed of FLOWSPEC is reasonable for use within an optimizing compiler.

This paper extends the initial SLE 2017 paper on FlowSpec [10]. We describe the FlowSpec design and implementation in more depth and provide evidence of its expressiveness by means of a significantly extended set of examples. We give a more complete definition of the syntax and semantics of the FlowSpec core language including its connection to name analysis using NaBL2. We describe the implementation of FlowSpec, including an adapted SCC algorithm and worklist algorithm, and discuss its integration in the Spoofax language workbench. We extend the case study of the application of FlowSpec to the specification of data-flow analyses for the Green-Marl data analytics DSL with more and more complete specfications of analyses, demonstrating that FlowSpec can be used to concisely define data-flow analyses that can be used to replace ad hoc implementations of these analyses in the Green-Marl compiler. We present a new case study, applying FlowSpec to the specification of reaching definitions analysis for the Stratego rewriting language. We evaluate the performance of FlowSpec analyses for GreenMarl and Stratego.

Outline: In the next section we discuss background on data-flow analyses and monotone frameworks. In Section 3 we introduce FLOWSPEC by example. We present the semantics of FLOWSPEC in Section 4. In Section 5 we describe the implementation of FLOWSPEC, both its integration into Spoofax and the independent solver algorithm. In Section 6 we present the first part of our evaluation of FLOWSPEC through data-flow analyses specified for the GREEN-MARL programming language. We present some benchmarks that show that these analyses are practically usable. In Section 7 we present the second part of our evaluation of FLOWSPEC through a data-flow analysis for the STRATEGO term transformation language. This section includes a comparative performance evaluation with the same data-flow analysis as currently implemented in the STRATEGO compiler. In Section 8 we compare against related work, in Section 9 we discuss future research directions, and finally Section 10 concludes the paper.

2. Background: data-flow analysis and monotone frameworks

In this section we introduce data-flow analysis in general and monotone frameworks as a mathematical framework for sound, terminating data-flow analysis.

2.1. Data-flow analysis by example

We start this introduction to data-flow analysis with two examples. Consider the *live variables* analysis in Fig. 1. Live variables analysis provides the set of variable names, where the value currently bound to that variable *may* be read further along in the program. The figure shows an example program, the results of live variables analysis for each statement, both before and after the effect of the statement, and the control-flow graph of the program. The control-flow graph shows how the program will execute either statement 5 or statement 6 based on whether condition 4 holds. Note how the variable `x` is only read in one branch of the `if` statement. Before the `if` statement, in the LV$_\circ$ set, `x` is still in the set as the analysis approximates the behavior of both branches.

When a variable is not in the set of live variables after the statement that assigns a value to that variable, that means that the value assigned is not actually read. This information can be used by an optimization to safely remove that assignment from the program. In the example, the assignment to `x` in statement 1 is such a redundant assignment, which can be recognized by the absence of `x` in the LV$_\bullet$ set of that first statement. This is consistent with the program, which does not read `x` until statement 6, and yet the variable is unconditionally reassigned in statement 3.

For comparison, we now discuss another data-flow analysis, *available expressions*, shown in Fig. 2. Available expressions analysis provides the set of expressions that have already been computed. Expressions become unavailable again when a variable used in the expression is assigned a new value. Note that for an expression to be available, it needs to be available in *all* paths. At the start of the `while` loop, we can only consider expressions available that are available right before the loop *and* at the end of the body of the loop. Therefore `a*b` is not available in AE$_\circ$ of condition 3, whereas `a+b` is.

The information from available expressions analysis can be used to remove redundant recomputations of expressions. We can save a repeated expression in a separate variable and use that variable instead of the expression, which is known as *common subexpression elimination*. In our example this would be expression `a+b` which can be replaced by `x` in condition 3.

```
[x = 2]^1;
[y = 4]^2;
[x = 1]^3;
if([y > 0]^4) {
  [z = y]^5;
} else {
  [z = x * y]^6;
}
[x = z]^7;
```

	LV$_\circ$	LV$_\bullet$
1	∅	∅
2	∅	{y}
3	{y}	{x, y}
4	{x, y}	{x, y}
5	{y}	{z}
6	{x, y}	{z}
7	{z}	∅

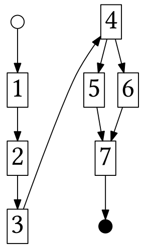


Fig. 1. An illustration of *live variables* (LV) analysis. On the left is an example program in the WHILE language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the center is the analysis result. The LV$_\circ$ and LV$_\bullet$ are respectively before and after the variable accesses of the CFG node.

```
[x = a + b]^1;
[y = a * b]^2;
while([y > a+b]^3) {
  [a = a + 1]^4;
  [x = a + b]^5
}
```

	AE$_\circ$	AE$_\bullet$
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}

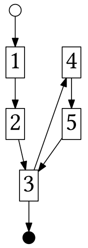


Fig. 2. An illustration of *available expressions* (AE) analysis. On the left is an example program in the WHILE language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the center is the analysis result. The open and closed dots on the analysis abbreviation are before and after a CFG node's effect, respectively.

In general, we note that for live variables analysis we need to know the behavior in the next part of the program, whereas for available expressions we need to know the expressions computed earlier. Therefore the computation of an analysis may need to propagate information either forward or backward. We also need to approximate the behavior of the program when there are multiple paths to a program point. In live variables analysis we consider variables read in any path, whereas in available expressions we consider only expressions computed in all paths. It is useful to model these paths of control-flow with a control-flow graph, to abstract from concrete language constructs.

2.2. Taxonomy of data-flow analysis

The example analyses are both *flow-sensitive* analyses. These analyses take the control-flow of the program into account, i.e. the order in which effects occur. *Flow-insensitive* analysis is less accurate, but also computationally cheaper. This can be a useful trade-off for whole-program analysis, where procedure calls are taken into account. A refined form of flow sensitivity is the *path-sensitive* data-flow analysis, which derives information from conditionals as it takes one path or another. Information from conditionals is also used in the type systems of some programming languages [11–13], where the terminology is flow-sensitive types. In programming languages these flow-sensitive types are primarily used for conveniences such as tracking null-safety of point types, as well as more general structural sub-typing support.

The data-flow analyses we just presented are *intra-procedural*, i.e. they only consider code within procedures, and not procedure calls. By contrast, *inter-procedural* analysis takes calls into account. Since procedures can be called from multiple places, a sound analysis must either approximate over all contexts in which a procedure can be called, or the analysis must be *context-sensitive*. Different forms of context sensitivity exist. For example, *call-site sensitivity* is a form of context sensitivity that keeps a string of calls through which the current procedure was reached. A well-known *control-flow analysis*, is *k*-CFA [14].

When a programming language allows dynamic dispatch, e.g. through function pointers, higher-order functions, or inheritance, the control-flow from a call site can depend on run-time values. At this point inter-procedural data-flow analysis becomes interdependent with a dynamic control-flow analysis. This interaction between control-flow and data-flow is difficult to handle, as more approximation in one analysis, which speeds up that analysis, will result in more work for the other analysis. In object-oriented languages, the context sensitivity that shows potential for these analyses is *object-sensitivity*, which tracks the allocation sites of objects [15]. Generally, finding the right contexts with a good trade-off in efficiency and accuracy are a topic of active research.

We aim to support flow-sensitive, intra-procedural data-flow analysis in FLOWSPEC as a start, which provides language designers with the tools to accurately analyze local properties.

2.3. Monotone frameworks

Monotone frameworks [16] is a formal method for describing data-flow analyses. We give a short introduction to the framework here. Throughout this paper we use the notation from Nielson et al. [1], which is the dual notation of the original publication (e.g. ⊔ instead of ∧).

In short, monotone frameworks is a general lattice theoretic framework for the definition of data-flow analyses. It captures the commonalities of intra-procedural, flow-sensitive data-flow analyses, and requires a number of components to be plugged in for any specific analysis. Given the correct components, this framework not only gives a clear, terminating semantics to a data-flow analysis, but also a simple worklist algorithm that can perform the analysis. The components required to instantiate a monotone framework are:

- The control-flow graph of a program in the form of a label set, an edge list of label pairs, and the starting labels.
- The type of data gathered by the data-flow analysis, along with its complete lattice instance of finite height. The framework uses lattice theory to guarantee a sound and terminating semantics.
- Transfer functions for every label in the control-flow graph, where the functions are monotone increasing with respect to the lattice.
- An initial value for the data-flow analysis at the starting labels.

2.4. Control-flow graphs

In order to make a data-flow analysis flow-sensitive, we need the control-flow of a program. In monotone frameworks program fragments are labelled ($\ell \in \mathbf{Lab}$) to distinguish different parts of the program. A control-flow graph F is a set of edges (a subset of $\mathbf{Lab} \times \mathbf{Lab}$) between different labels.

For a forward propagating data-flow analysis, this graph can be used as is. For a backward propagating analysis the edges of the graph are simply flipped. The framework also takes the ‘extremal labels’ $E \in \mathcal{P}(\mathbf{Lab})$, which are the start nodes of the analysis.

2.5. Data-flow type and transfer functions

Each control-flow graph node has an effect. An analysis specifies how this effect influences the analysis through a transfer function $f_\ell : L - > L$, where L is the type of the information the data-flow analysis propagates. At the extremal labels, this information is initialized with the extremal value ι.

An established factorization of these transfer functions is the *kill* and *gen* sets approach. For set based analyses, a kill set is defined separately from a gen set for each control-flow node of interest. The transfer function is then generic: first remove the kill set, then add the gen set.

Fig. 3 shows the monotone framework instance for available expressions analysis. The transfer function takes the set of available expressions, first removes any expressions containing the variable that is assigned (or nothing if it is not an assignment), then it adds any new expressions that do not contain the variable that is assigned. Note how the gen set has to repeat the conditions of the kill set. The reason for this repetition is that the right-hand side of the expression, that generates available expressions, happens before the assignment effect of the left-hand side. In a backward analysis this would not be the case, therefore on first glance independent kill and gen sets are *sometimes* dependent. We argue that this subtlety can be a source of analysis bugs.

We aim to support flow-sensitive

L	$\mathcal{P}(\mathbf{AExp})$
$\sqsubseteq$	$\supseteq$
$\bigsqcup$	$\bigcap$
$\perp$	$\mathbf{AExp}(\mathbf{Prog})$
ι	$\emptyset$
E	$\{init(\mathbf{Prog})\}$
F	$flow(\mathbf{Prog})$

$$f_\ell(l) = (l \setminus kill([B]^\ell)) \cup gen([B]^\ell)$$

$$\text{where } [B]^\ell \in blocks(\mathbf{Prog})$$

$$kill([x := a]^\ell) = \{a' \in \mathbf{AExp}(\mathbf{Prog}) \mid x \in FV(a')\}$$

$$gen([x := a]^\ell) = \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\}$$

$$gen([b]^\ell) = \mathbf{AExp}(b)$$

Fig. 3. The monotone framework instance for available expressions. An $l \in L$ is an element of the lattice. Two of those elements can be compared with ⊑, and joined with ⊔. ⊥ is the bottom of the lattice. The framework operates on the forward control-flow F, from the set of labels E, where the initial analysis information is ι. **Prog** is the entire program, *blocks* collects all labelled blocks, *FV* collects all free variables, *init* gives the initial label, and *flow* gives the control flow of the argument. **AExp** gives all arithmetic expressions.

2.6. Control-flow and lattices

If a control-flow graph node ℓ has the information $\text{Analysis}_\circ(\ell)$ before the effect of ℓ then we can use its transfer function f_ℓ to compute $\text{Analysis}_\bullet(\ell)$. However, when multiple control-flow paths join at a certain node, we need to merge the data from those different paths. We use the ⊔ operator for this to reach these equations:

$$\begin{aligned} \text{Analysis}_\circ(\ell) = &\ \bigsqcup\{\text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell \\ &\text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \perp & \text{if } \ell \notin E \end{cases} \\ \text{Analysis}_\bullet(\ell) = &\ f_\ell(\text{Analysis}_\circ(\ell)) \end{aligned}$$

The open dot is the analysis result before the effect of ℓ, and the closed dot is for after the effect of ℓ. The transfer function f_ℓ is used to compute the effect of ℓ. The ⊔ operator is used to combine the analysis data after the previous nodes ℓ' as the analysis data right before the current node ℓ. We use the initial value ι for the initial labels E and a ⊥ value elsewhere, where $\perp \sqcup d = d = d \sqcup \perp$.

Finding the fixed point to these equations may not be possible though, as loops in the control-flow graph make the equations recursive. Therefore we need stronger guarantees, for which lattices are used.

Monotone frameworks require a complete lattice instance (⊤, ⊥, ⊑, ⊔, ⊓) for the type L of the data-flow property. The intuition is that ⊤ is the value of L that reads as “could be anything”, the coarsest approximation available. By using the least upper-bound operator (⊔) we combine the information from two paths in the control-flow so it soundly approximates both (upper bound), while keeping as much information as possible (*least* upper bound).

In Fig. 3, the monotone frameworks instance of available expressions uses a powerset lattice. Available expressions analysis is a must analysis, which only keeps information that *must* be true for *all* paths. Therefore the analysis applies set intersection at join-points.

Now that we have a clearer definition of the ⊔ operator, we can resolve the issue of finding a fixed point to the equations. Monotone frameworks have two particular requirements. First, the transfer functions f_ℓ need to be monotone increasing with respect to the lattice. This means that in a loop either the information becomes more approximate, or it stays the same, in which case we have a fixed point. Secondly, the lattice must adhere to the ascending chain condition. In other words, the lattice must have a finite height. This way when the information on a loop keeps increasing, it takes a finite number of steps to reach $\top$, which is a fixed point for monotone increasing transfer functions.

Of course $\top$ is the coarsest approximation available. Although some approximation is necessary to keep the analysis computable, we can usually do better than $\top$ everywhere. The fixed point of the Analysis that we want is the *least fixed point*. This fixed point has enough information to be valid, with as little approximation as necessary. The accuracy of this fixed point is still dependent on the choice of lattice L and transfer functions f.

In the original work on monotone frameworks [16] the dual notion with meets (greatest lower bounds) and greatest fixed points was used. There, the authors give the Meet Over all Paths (MOP) as the desired solution, but show that this solution can be undecidable to calculate. In cases where it *can* be calculated, the greatest fixed point coincides with it, in cases where it is undecidable, the greatest fixed point safely approximates the MOP solution [1, sect. 2.4.2]. Therefore a correct instantiation of a monotone framework gives a computable, safe approximation of the run-time behavior of a program.

2.7. Worklist algorithm

Given an instance of a monotone framework, we can compute the fixed point of the recursive equations iteratively with a worklist algorithm, such as the one in Fig. 4. This algorithm works in three steps. First it initializes the analysis result $\mathtt{Analysis}_\circ$ to what comes down to ι^{ℓ}_{E}, and the worklist to all nodes in the control-flow graph. Second, it loops over the worklist, taking out one node at a time, and propagates transferred analysis information to successors in the control-flow graph. If that information is new ($\not\sqsubseteq$) the $\sqcup$ operator is used to add the information to the analysis information of the successor, and that successor is added to the worklist again. Once no more new information is discovered, the worklist becomes empty. The third step computes $\mathtt{Analysis}_\bullet$ as defined in its formula.

```
for $\ell \in F$:
    if $\ell \in E$:
        Analysis$_\circ$($\ell$) := $\iota$
    else:
        Analysis$_\circ$($\ell$) := $\bot$

$W$ := list of $\mathbf{Lab}$

while $W$ is not empty:
    $\ell$ := $W$.pop()
    for ($\ell$, $\ell'$) $\in F$:
        if $f_\ell$(Analysis$_\circ$($\ell$)) $\not\sqsubseteq$ Analysis$_\circ$($\ell'$):
            Analysis$_\circ$($\ell'$) := ($f_\ell$(Analysis$_\circ$($\ell$))
                                     $\sqcup$ Analysis$_\circ$($\ell'$))
            $W$.push($\ell'$)

for $\ell$ in $F$:
    Analysis$_\bullet$($\ell$) := $f_\ell$(Analysis$_\circ$($\ell$))
```

Fig. 4. A worklist algorithm to iteratively solve the equations of a monotone framework instance. W is the worklist. **Lab** is the set of labels used in the control-flow graph F.

2.8. Monotone frameworks recap

To summarise, to specify a data-flow analysis with monotone frameworks, we need the following ingredients:

1. A finite flow, $F \in \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$.
2. Labels $\ell \in \mathbf{Lab}$, which reference program fragments.
3. A set of extremal labels, $E \in \mathcal{P}(\mathbf{Lab})$, typically the initial label(s) of the flow.
4. A type L of the data-flow property, which is a complete lattice of finite height.
5. Monotone transfer functions f_ℓ for every label ℓ in the control-flow graph.
6. An extremal value, $\iota \in L$, for the extremal labels.

Monotone frameworks give a design pattern for correct data-flow analysis, and an implementation for such an analysis. However, direct instantiations of worklists for different analyses, especially analyses that use the results of other analyses can result in a complex implementation that is difficult to update or adapt.

In language workbenches we want to specify a data-flow analysis and get the implementation 'for free', i.e. we abstract from the implementation method. The iterative algorithm can still be used under the hood, but is no longer directly seen. The specification should be easy to understand and avoid pitfalls such as we saw in the gen and kill set definition of available expressions. In short, we need a domain-specific language for data-flow analysis specification.

3. FLOWSPEC by example

FLOWSPEC is a domain-specific language for specifying data-flow analysis, that builds on the theory of monotone frameworks. A FLOWSPEC specification only includes the analysis-specific elements, and from this specification we generate an implementation for that analysis. In this section we introduce FlowSpec by a number of examples.

3.1. Requirements

In language workbenches we want to specify a data-flow analysis and get the implementation 'for free', i.e. we abstract from the implementation method. The specification should be easy to understand and avoid pitfalls such as we saw in the gen and kill set definition of available expressions for monotone frameworks.

Within the context of a language workbench, we need a language that reuses information that is already available within a language specification of the workbench. We do not need to define a data-flow analysis directly on source text of a program, as we can obtain the abstract syntax of that program within the workbench. We can also reuse name and type analysis that is available. Our domain-specific language does not need to support the specification of such analyses, it should only support the use of the analysis results.

What we need then is a language that uses the concepts of abstract syntax, names and types, and provides features to define what is distinctly part of the domain of data-flow analysis. FLOWSPEC provides the features to define the relation between the control-flow and the abstract syntax of a programming language, and what effects control-flow nodes of the programming language have for different data-flow analyses.

3.2. Concrete and abstract syntax

In Spoofax the concrete and abstract syntax of a programming language are defined in SDF3. As an example, we provide the SDF3 definition of the syntax of our running example language in Fig. 5.

```
module running-example

context-free syntax

  Statement.Seq =
    [[Statement]
     [Statement]] {right}

  Statement.Assign = [[ID] = [Expr];]

  Statement.IfThenElse =
    [if([Expr]) [Statement]
     else [Statement]]

  Statement.While =
    [while([Expr])
       [Statement]]

context-free syntax

  Expr.IntLit = INT
  Expr.True   = [true]
  Expr.False  = [false]
  Expr.VarRef = ID

  Expr      = [([Expr])] {bracket}

  Expr.UMin  = [-[Expr]]
  Expr.Mul   = [[Expr] * [Expr]] {left}
  Expr.Div   = [[Expr] / [Expr]] {left}
  Expr.Add   = [[Expr] + [Expr]] {left}
  Expr.Sub   = [[Expr] - [Expr]] {left}

  Expr.Not = [![Expr]]
  Expr.And = [[Expr] && [Expr]] {left}
  Expr.Or  = [[Expr] || [Expr]] {left}
  Expr.Eq  = [[Expr] == [Expr]] {left}
  Expr.Gt  = [[Expr] > [Expr]]  {left}
  Expr.Lt  = [[Expr] < [Expr]]  {left}
  Expr.Geq = [[Expr] >= [Expr]] {left}
  Expr.Leq = [[Expr] <= [Expr]] {left}
  Expr.Neq = [[Expr] != [Expr]] {left}

context-free priorities

  { Expr.UMin  Expr.Not } >
  { left: Expr.Mul  Expr.Div  Expr.Mod } >
  { left: Expr.Add  Expr.Sub } >
  { left: Expr.Lt  Expr.Leq
          Expr.Gt  Expr.Geq } >
  { left: Expr.Eq  Expr.Neq } >
  Expr.And > Expr.Or

lexical syntax

  ID  = [a-zA-Z] [a-zA-Z0-9\_]*
  INT = "-"? [0-9]+
```

Fig. 5. The SDF3 grammar for the running example language WHILE.

This SDF3 grammar uses templates to specify grammar rules along with some basic formatting hints. Within the outer brackets (either square or angled), are terminals, within another pair of brackets are non-terminals. The first rule defines that a statement can be a sequence of two statements. The annotation `right` disambiguates this rule by making the rule right-associative. In other words, a sequence of three statements $S_1\, S_2\, S_3$ is parsed as $S_1\, (S_2\, S_3)$. In this example grammar we define statements as sequences of statements, assignment statements, if statements and while loop statements.

We define expressions within conditions and assignment right-hand sides. Expressions include a number of binary and unary operations which have associativity and priority to disambiguate. The lexical syntax for identifiers and integer literals is defined at the end with some regular expressions.

```
signature
  constructors // generated from SDF3 spec
    Seq : Statement * Statement → Statement
    Assign : ID * Expr → Statement
    IfThenElse :
      Expr * Statement * Statement → Statement
    While : Expr * Statement → Statement
  constructors
    VarRef : ID → Expr
  constructors // manually defined to desugar into
    BinOp : BinOp * Expr * Expr → Expr
    UnOp  : UnOp * Expr → Expr
constructors                       constructors
  Abs  : UnOp                        Mod  : BinOp
  UMin : UnOp                        Add  : BinOp
  Mul  : BinOp                       Sub  : BinOp
  Div  : BinOp                       Not  : UnOp
constructors                       constructors
  And  : BinOp                       Lt   : BinOp
  Or   : BinOp                       Geq  : BinOp
  Eq   : BinOp                       Leq  : BinOp
  Gt   : BinOp                       Neq  : BinOp
```

Fig. 6. The abstract syntax definitions of our running example language WHILE.

The abstract syntax of our running example is already written as part of the SDF3 grammar, in the form of constructor names on the rules. A sequence of statements uses `Seq`, an addition uses `Add`, et cetera. From the grammar we can generate the signatures of the abstract syntax, as shown in Fig. 6. The first two sections of signatures define the shape of the abstract syntax tree (AST) as defined in the grammar. However, we have *desugared* unary and binary operations to common constructors that have a separate operator field. These constructor signatures are hand-written, and some simple transformation rules can translate the original AST to this desugared version that we will operate on throughout the examples.

In Fig. 7 we give an example program along with its abstract syntax tree. The different Spoofax meta-languages, including FLOWSPEC, work with these ASTs by pattern matching against parts of the tree.

```
x = a + b;
y = a * b;
while(y > a + b) {
  a = a + 1;
  x = a + b;
}
```

```
Seq(
  Assign("x",
    BinOp(Add(), VarRef("a"), VarRef("b"))),
  Seq(
    Assign("y",
      BinOp(Mul(), VarRef("a"), VarRef("b"))),
    While(
      BinOp(Gt(), VarRef("y"),
        BinOp(Add(), VarRef("a"), VarRef("b"))),
      Seq(
        Assign("a",
          BinOp(Add(), VarRef("a"), IntLit("1"))),
        Assign("x",
          BinOp(Add(), VarRef("a"), VarRef("b")
                                          ))))))
```

Fig. 7. An example program (top) with its desugared abstract syntax tree (bottom).

3.3. Name and type analysis

Name and type analysis is extracted from an NaBL2 specification. This analysis annotates the entire tree with unique numbers, so different *occurrences* of a name can be distinguished from each other. All information of names and types is then attached to these occurrences.

The reason we care about name and type analysis, is that data-flow analysis commonly gathers information about names. Many programming languages allow shadowing of names, i.e. definition of a name in an inner scope when the same name is present already in an outer scope. Therefore the name of a variable is not unique enough when data-flow analysis collects information on that name.

In FLOWSPEC we operate on analyzed ASTs, in which name occurrences have unique numbers. Within FLOWSPEC we borrow the NaBL2 notation `Namespace{name}` for name occurrences. In FLOWSPEC, such an occurrence denotes names *after* name analysis. That is, names represented by the same occurrence, correspond to the same declaration. Thus, name capture is not a concern in FLOWSPEC.

3.4. Control-flow graphs

For flow-sensitive data-flow analysis we require a control-flow graph. Control-flow graphs are a finite representation of a possible infinite set of paths through a program. For example, as statements of a program are executed, control flows from one statement to the next. However, as soon as a program has a loop, control can flow around the loop or at some point exit the loop. A control-flow graph is a finite model that show where control *could* flow. Thereby a loop in a program become in a loop in a control-flow graph. Examples of control-flow graphs can be found back in Figs. 1 and 2.

3.5. Mapping from abstract syntax to control flow

In FLOWSPEC we build control-flow graphs between occurrences in the AST. Not only a string occurrence as is usually used for names, but also entire subtrees of the AST can be control-flow graph nodes. The FLOWSPEC specification defines which AST nodes should be considered control-flow graph nodes, and how control flows within and between different AST nodes.

Consider Fig. 8 where we have defined some example mapping rules. The rules are defined case-by-case using patterns to match the signature from Fig. 6. Each rule uses the contextual `entry` and `exit` nodes to connect the sub-graph of the matched AST node to the outer graph. These nodes do not show up in the final control-flow graph. When the AST node matched by the pattern should be included as a control-flow node, we use the `this` keyword to denote that. The direct use of a pattern variable from the AST pattern is substituted with the subgraph of that AST node in accordance with other control-flow rules.

```
module control

control-flow rules
  Assign(_, e) = entry → e → this → exit

  Seq(s1, s2) = entry → s1 → s2 → exit

  IfThenElse(c, t, e) =
    entry → c → t → exit,
            c → e → exit

  While(c, b) = entry → c → b → c → exit

  BinOp(_, l, r) = entry → l → r → this → exit

  UnOp(_, e) = entry → e → this → exit

  node VarRef(_)
  node IntLit(_)
  node True()
  node False()
```

Fig. 8. Control-flow graph rules for the WHILE language. Each rule can have one or more chains of edges, where `entry` and `exit` represent the connection between the local control flow the rest of the graph.

Fig. 9 shows how control-flow rules can be applied to a program in a number of steps. First the sequence rule is used to create an edge between the two statements. Then the assignment rule is used to create nodes of the assignments, with the expression preceding it. Then the binary operation rule creates a node for the operation and its operands, and finally the operands are turned into nodes themselves.

The rules of `IfThenElse` shows that multiple chains of edges in the control-flow graph may be defined. This allows us to express conditions. The sub-graph of condition `c` is followed by both the subgraph of the then-branch and subgraph of the else-branch. Each of the branches connect to the `exit` of the construct. Multiple uses of the `c` sub-graph

```
[[Seq(
  Assign("a",
    BinOp(Add(), VarRef("a"), IntLit("1"))),
  Assign("x",
    BinOp(Add(), VarRef("a"), VarRef("b")))
)]]

entry →
[[Assign("a",
  BinOp(Add(), VarRef("a"), IntLit("1")))]] →
[[Assign("x",
  BinOp(Add(), VarRef("a"), VarRef("b")))]] →
exit

entry →
  [[BinOp(Add(), VarRef("a"), IntLit("1"))]] →
  node Assign("a", ...) →
  [[BinOp(Add(), VarRef("a"), VarRef("b"))]] →
  node Assign("x", ...) →
exit

entry →
    [[VarRef("a")]] → [[IntLit("1")]] →
    node BinOp(Add(), ..., ...)) →
  node Assign("a", ...) →
    [[VarRef("a")]] → [[VarRef("b")]] →
    node BinOp(Add(), ..., ...)) →
  node Assign("x", ...) →
exit

entry →
      node VarRef("a") → node IntLit("1") →
    node BinOp(Add(), ..., ...)) →
  node Assign("a", ...) →
      node VarRef("a") → node VarRef("b") →
    node BinOp(Add(), ..., ...)) →
  node Assign("x", ...) →
exit
```

Fig. 9. Control-flow graph rules applied to a piece of abstract syntax, where double square brackets show parts of the AST that are not processed yet.

```
module liveness

properties
  live : MaySet(name)

property rules
  live(Assign(n, _) → s) = live(s) \ { Var{n} }

  live(VarRef(n) → s) = live(s) ∪ { Var{n} }

  live(_ → s) = live(s)
```

Fig. 10. Live Variables specification in FLOWSPEC. Note how the rule for assignments does not inspect the right-hand side expression. Instead the control-flow is defined within expressions (not in this figure), and a separate rule for the variable reference expressions adds live variables. Names are added to the live variables as names within a namespace `Var`. External name information is used to handle name issues such as shadowing.

refer to the same subgraph. This is also used in the `While` rule, where multiple uses of condition `c` construct the loop.

In these rules binary and unary operations, i.e. expressions, are also considered part of the control-flow graph. This is not a restriction in FLOWSPEC, we define the control-flow this way to the benefit of our data-flow analysis definitions later. One could also use `node c` within the chain of edges to make condition expression `c` a node in the control-flow graph[1]

The rules for variable references and integer literals are a shorthand to define that this is a node in the control-flow graph and it has no further control-flow inside. The following would be equivalent to the variable reference rule:

```
VarRef(_) = entry -> this -> exit
```

3.6. Data-flow type and transfer functions

FLOWSPEC defines data-flow analyses as properties on the control-flow graph. During analysis, the data of this property is propagated along the control-flow graph. Every node in the control-flow graph has an associated effect on this data.

In Fig. 10 we show FLOWSPEC's analogue of transfer functions for live variables analysis: property rules. Property rules show both the direction of the data-flow analysis, in this case backward, and define the data-flow property in terms of itself. We have a rule for assignments, which only applies the effect of the assignment itself and disregards the right-hand side expression. A separate rule for the variable reference expression handles the effect of reading a variable. We are able to split these two effects because earlier we defined control-flow in expressions too.

The FLOWSPEC specification of available expressions is given in Fig. 11. We use an external property `refs` from NaBL2 to extract references from expressions. The effects of the assignment and its right-hand side expression are split over multiple rules again. The assignment filters out those expressions that use the variable that is being assigned to. We can express this as a direct filter instead of relying on global program information of all expressions, as was the case in the monotone frameworks definition in Fig. 3. Since our control-flow graph includes the expressions in an assignment separately, it models the ordering of effects directly. Therefore the FLOWSPEC specification does not suffer from the subtle interdependence that the traditional kill-gen definitions have.

```
module availability

properties
  available: MustSet(term)
  external
    refs: Set(name)
property rules
  available(prev → Assign(n, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr) }

  available(prev → e@BinOp(_,_,_)) =
    available(prev) ∪ {e}

  available(prev → e@UnOp(_,_,_)) =
    available(prev) ∪ {e}

  available(prev → _) = available(prev)
```

Fig. 11. Available Expressions specification in FLOWSPEC. We consider references in expressions a separate concern based on names, not flow and therefore out of scope for our language. Note how the assignment rule only handles the assignment effect, expressions are visited separately.

3.7. Lattices and termination

The control-flow can split and join because of conditional control-flow such as an `if` statement. We can propagate data along both edges of a split, but need to merge the data coming from multiple directions at a join. The data is merged before the property rule of the join-point node is applied. We require a lattice instance ($\top$, $\bot$, $\sqsubseteq$, $\sqcup$, $\sqcap$) for the type of the data-flow property, and use the least-upper bound $\sqcup$ at join points in the control-flow. In our examples the `MaySet` and `MustSet` are lattice instances that use the `Set` type:

```
properties
  live : MaySet(name)
  available : MustSet(term)
```

A `MaySet` performs set unions at control-flow join points and compares with non-strict subset comparison. A `MustSet` uses intersection and non-strict superset comparison. It uses a symbolic bottom element to represent the full set of possible values in the analysis.

3.8. Very busy expressions

Very busy expression analysis provides the set of expressions which

```
module busyness

properties
  external
    refs: Set(name)

properties
  veryBusy: MustSet(term)

property rules
  veryBusy(Assign(n, e) → next) =
    { expr |
      expr ← veryBusy(next),
      !(n in refs(expr)) }

  veryBusy(e@BinOp(_,_,_) → next) =
    veryBusy(next) ∪ {e}

  veryBusy(e@UnOp(_,_,_) → next) =
    veryBusy(next) ∪ {e}

  veryBusy(_ → next) = veryBusy(next)
```

Fig. 12. Very Busy Expressions specification in FLOWSPEC.

[1] In fact ***this*** is syntactic sugar for **node**`t` where `t` is the whole AST node matched by the pattern of the rule.

```
module reach

properties
  definition: MaySet(name * occurrence)

property rules

  definition(prev → this@Assign(n, e)) =
    { (Var{n}, occurrence(this)) } ∪
    { (m, l) |
      (m, l) ← definition(prev),
      m != Var{n} }

  definition(prev → _) = definition(prev)
```

Fig. 13. Reaching Definitions specification in FLOWSPEC.

will definitely be calculated in the future. This information can be used to hoist an expression out of an if statement if it is calculated in both branches. In Fig. 12 we provide the definition of very busy expressions analysis in FLOWSPEC. Note how similar this analysis is to the available expressions analysis.

3.9. Reaching definitions

Reaching definition analysis is an analysis that provides the positions in the program where a variable was last assigned a value. This can be multiple positions since a variable may be assigned in different conditional paths in the control-flow. See Fig. 13 for the FLOWSPEC description of reaching definitions. To preserve the information from all branches, we use a `MaySet`. The `occurrence` is used to denote the position in the program where the assignment occurred.

The sole rule for reaching definitions analysis of our example language is that of the assignment. There we remove any previously reaching definitions of the currently assigned variable. We add the pair of the name and the occurrence of the assignment.

3.10. Constant propagation and folding

Constant propagation is the name of both an analysis and the corresponding optimization. The optimization replaces uses of a variable with its value if that value is guaranteed to be constant. Constant folding is the optimization that computes constant expressions and replaces those expressions with the computed result. We combine these two optimizations and make them part of our constant propagation, to improve the accuracy of the analysis results. Because of constant folding, more constants can be found. Because more constants can be found, and filled into expressions, more constant expressions can be folded.

```
module constants

prop constProp: CP

constProp(prev → Assign(n, e)) =
  match constProp(prev) with
    | M1(m, v) ⇒ M(m ∪ {Var{n} ↦ v})
    | _ ⇒ CP.top

constProp(prev → Add(e1,e2)) =
  match constProp(prev) with
   | M2(m, l, r) ⇒ M1(m, constAdd(l,r))
   | _ ⇒ CP.top

constProp(prev → VarRef(n)) =
  addResult(
    constProp(prev),
    getMap(constProp(prev))[Var{n}])

constProp(prev → _) = constProp(prev)
```

Fig. 14. Constant propagation specification in FLOWSPEC. `CP` holds the map of names to constants and 0, 1 or 2 constants from previous computations.

In Figs. 14 and 15 we give the definition of this combined constant propagation analysis in FLOWSPEC. The constant propagation property had a `Map` type. A FLOWSPEC `Map` forms a lattice if the value type forms a lattice. Any key not bound in the map, instead maps to the top of the

```
types
  CPType =
    | M(Map(name, Const))
    | M1(Map(name, Const), ConstProp)
    | M2(Map(name, Const), ConstProp, ConstProp)

  ConstProp =
    | Top()
    | Const(int)
    | Bottom()

functions
  getMap(cpt: CPType) =
    match cpt with
      | M(m) ⇒ m
      | M1(m,_) ⇒ m
      | M2(m,_,_) ⇒ m

  addResult(cpt: CPType, v: Const) =
    match cpt with
      | M1(m, v1) ⇒ M2(m, v1, v)
      | _ ⇒ M1(getMap(cpt), v)

  constAdd(l: Const, r: Const) =
    match(l,r) with
      | (Const(i), Const(j)) ⇒ Const(i+j)
      | _ ⇒ Const.top

lattices
  CP where
    type CPType

    lub(l, r) = match (l,r) with
      | (M(l), M(r)) ⇒ M(Map.lub(l,r))
      | (M1(l, cl), M1(r, cr)) ⇒
          M1(Map.lub(l,r), Const.lub(cl,cr))
      | (M2(l, cl1, cl2), M2(r, cr1, cr2)) ⇒
          M2(
            Map.lub(l,r),
            Const.lub(cl1, cr1),
            Const.lub(cl2, cr2))
      | _ ⇒ CP.top

    bottom = M(Map.bottom)

    top = M(Map.top)

  Const where
    type = ConstProp

    lub(l, r) = match (l,r) with
      | (Top(), _) ⇒ Top()
      | (_, Top()) ⇒ Top()
      | (Const(i), Const(j)) ⇒ if i == j
          then Const(i) else Top()
      | (_, Bottom()) ⇒ l
      | (Bottom(), _) ⇒ r

    bottom = Bottom()
```

Fig. 15. The type, function and lattice definitions for *constant propagation* specification in FLOWSPEC.

lattice of the value type. The ⊔ and ⊑ operators are defined point-wise. This means that if a variable is only constant in one condition branch, when it joins with another branch the variable will no longer be considered constant.

The constant value lattice has a symbolic top and bottom, and constants which are not ordered. Therefore when two branches in the control-flow join, and different constant values for the same variable are found, that variable is no longer constant at the join point. The constant propagation property rule takes assignment into account and applies the `foldConst` function, which computes constant expressions.

3.11. Sign analysis

Sign analysis is a data-flow analysis that computes the possible sign of integers typed variables. This can be used to detect if a comparison condition will always evaluate to a constant, which makes further analysis more accurate as a branch of control-flow is eliminated. Sign analysis is similar in definition to constant propagation as illustrated in Fig. 16.

3.12. Definite assignment

Definite assignment analysis keeps a set of variables which have definitely been assigned a value. This information can be used to give warnings or error upon the use of a possibly uninitialized variable. The FLOWSPEC specification of definite assignment is in Fig. 17.

```
module sign

properties
  sign : MaySet(name * Sign)

property rules
  sign(prev → Assign(n, e)) =
    (Var{n}, interpSet(sign(prev), e)) ∪
    { (m, s) | (m, s) ← sign(prev), m != Var{n} }

  sign(prev → _) = sign(prev)

functions
  interpSet(set, e) = match e with
    | VarRef(n) ⇒ { s | (m, s) ←
set, m == Var{n} }
    | BinOp(op,e1,e2) ⇒
      { s | l ← interpSet(set, e1),
            r ← interpSet(set, e2),
            s ← interpBin(op,l,r) }
    // etc.

  interpBin(op, l, r) = match op with
    | Add() ⇒ (if l == r
      then { l }
      else match (l, r) with
        | (_, Zero()) ⇒ { l }
        | (Zero(), _) ⇒ { r }
        | _ ⇒ { Pos(), Neg(), Zero() })
    // etc.

types
  Sign =
    | Zero()
    | Neg()
    | Pos()
```

Fig. 16. Sign Analysis specification in FLOWSPEC.

```
module initialization

properties
  definite: MustSet(name)

property rules

  definite(prev → this@Assign(n, e)) =
    definite(prev) ∪ { Var{n} }

  definite(prev → _) = definite(prev)
```

Fig. 17. Definite Assignment specification in FLOWSPEC.

4. The semantics of FLOWSPEC

In this section we present the semantics of FLOWSPEC. For brevity we only show rules for the novel parts of the language, and use monotone frameworks as the semantic model for the language. We will discuss the language in roughly the same order as in the last section. Please refer to Fig. 18 for a small syntax definition of the language, from which we will use non-terminals to introduce judgements of the semantics.

Note that the `this` construct in FLOWSPEC is syntactic sugar for a `node t` where `t` refers to the entire AST that was matched with pattern *p*.

$S ::=$	$\texttt{control-flow rules}\ G^*$	control-flow section
	$\mid\ \texttt{properties}\ D^*$	dataflow prop def section
	$\mid\ \texttt{property rules}\ R^*$	dataflow prop rules section
$G ::=$	$p = C\ \{,\ C\}^*$	control-flow rules
$C ::=$	$E \rightarrow E\ \{\rightarrow E\}^*$	edge chains
$E ::=$	$\texttt{entry} \mid \texttt{exit} \mid \texttt{start} \mid \texttt{end}$	chain elements
	$\mid\ n \mid \texttt{node}\ n$	
$D ::=$	$n : t$	property definitions
$R ::=$	$n\ P = e$	property rules
$P ::=$	$p \rightarrow n$	match ahead
	$\mid\ n \rightarrow p$	match behind
$p ::=$	$n(p^*)$	term pattern
	$\mid\ (p^*)$	tuple pattern
	$\mid\ \_$	wildcard pattern
	$\mid\ n$	pattern variable
	$\mid\ n\texttt{@}p$	as pattern
$e ::=$	n	variable reference
	$\mid\ n(n)$	property lookup
	$\mid\ n(e^*)$	function application
	$\mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	if else
	$\mid\ \texttt{match}\ e\ \texttt{with}\ \{\texttt{\|}\ p \Rightarrow e\}^*$	pattern match
	$\mid\ \texttt{type}(n)$	type lookup
	$\mid\ n\ \{n\}$	name lookup
	$\mid\ \texttt{occurrence}(n)$	occurrence lookup
	$\mid\ e == e \mid e\ \texttt{!=}\ e \mid \texttt{!}e$	equality, inequality and negation
	$\mid\ (e^*) \mid n(e^*)$	tuple and term literals
	$\mid\ s \mid i$	string and number literals
	$\mid\ \{e^*\}$	set literal
	$\mid\ \{e\ \texttt{\|}\ p \leftarrow e\{,p \leftarrow e\}^*\{,e\}^*\}$	set comprehension
	$\mid\ \{\{e \mapsto e\}^*\} \mid e[e]$	map literal and lookup
	$\mid\ e \cup e \mid e \setminus e \mid e\ \texttt{in}\ e \mid e \cap e$	set operations
n		names
t		types
s		string
i		number

Fig. 18. The core grammar of FLOWSPEC.

4.1. Control-Flow rules

The control-flow rules, that map the abstract syntax of a language to its control-flow, are defined case-wise with AST patterns. To model the behavior of the virtual entry and exit nodes in these rules, we employ a constraint based semantics, given in Fig. 19. The smallest set that satisfies these constraints is the control-flow graph that the rules define. We use $\llbracket p \rrbracket^{a^\ell} = \Gamma$ to abstract over pattern matching, where p is the pattern, a^ℓ is the labeled AST node, and Γ is the environment with bindings that come from the match. The extremal labels are all possible, valid bindings of $\ell_\circ$ and $\ell_\bullet$ for [*rule$_i$*] where a^ℓ is the whole program.

In general the four labels left of the turnstile are the virtual entry and exit labels, and the start and end labels. The entry and exit labels are mostly left to be inferred by the rules. The chain rule [*noedge*] connects the labels in a chain by using an inference variable as a label to connect the two judgements. The chain rule [*edge*] connects the labels by using two inference variables and adding an edge between these variables to the graph.

For the chain element rules [*en*] and [*ex*] we assume that entry nodes are only on the left-most end of a chain, and exit nodes are only on the right-most end of a chain. The entry and exit rules simply equate the two labels left of the turnstile, without putting any constraints on the two labels. The [*end*] and [*start*] rules are similar, except these use the downward propagated **start**$^\ell$ and **end**$^\ell$ nodes that are created by the `root` rule. The [*lab*] rule looks up the label of the AST node, and requires that both labels left of the turnstile are equal to this label. This rule is the one that adds an actual label to the system of rules, instead of an inference variable. This forces the [*edge*] rule to be used between two AST nodes, resulting in actual edges in the constraints. Lastly the [*cfg*] rule handles the recursive call of `cfg`, where it will use any cfg rule from the program which matches the AST node that the variable refers to.

Cfg root rule constraints $\boxed{\vdash \llbracket G \rrbracket^{a^\ell} \supseteq \mathbf{Lab} \times \mathbf{Lab}}$

$$\frac{\llbracket p \rrbracket^{a^\ell} = \Gamma \wedge 1 \leq i \leq m \quad \wedge\ \mathbf{start}^\ell, \mathbf{end}^\ell, \mathbf{start}^\ell, \mathbf{end}^\ell; \Gamma \vdash C_i \supseteq g_i}{\vdash \llbracket \mathbf{root}\ p = C_1, \ldots, C_m \rrbracket^{a^\ell} \supseteq g_i}\ [\mathrm{rule}_i]$$

Cfg rule constraints $\boxed{\ell, \ell, \ell, \ell \vdash \llbracket G \rrbracket^{a^\ell} \supseteq \mathbf{Lab} \times \mathbf{Lab}}$

$$\frac{\llbracket p \rrbracket^{a^\ell} = \Gamma \wedge \ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash C_i \supseteq g_i \wedge 1 \leq i \leq m}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e} \vdash \llbracket p = C_1, \ldots, C_m \rrbracket^{a^\ell} \supseteq g_i}\ [\mathrm{rule}_i]$$

Cfg chain constraints $\boxed{\ell, \ell, \ell, \ell; \Gamma \vdash C \supseteq \mathbf{Lab} \times \mathbf{Lab}}$

$$\frac{\ell_\circ, \ell, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \supseteq g_1 \quad \wedge\ \ell, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_2 \rightarrow \ldots \rightarrow E_m \supseteq g_2}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_m \supseteq g_1 \cup g_2}\ [\mathrm{noedge}]$$

$$\frac{\ell_\circ, \ell_1, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \supseteq g_1 \wedge \ell_1 \neq \ell_2 \quad \wedge\ \ell_2, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_2 \rightarrow \ldots \rightarrow E_m \supseteq g_2 \quad \wedge\ g_3 = \{(\ell_1, \ell_2)\}}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_m \supseteq g_1 \cup g_2 \cup g_3}\ [\mathrm{edge}]$$

Cfg element constraints $\boxed{\ell, \ell, \ell, \ell; \Gamma \vdash E \supseteq \mathbf{Lab} \times \mathbf{Lab}}$

$$\frac{}{\ell_\circ, \ell_\circ, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \mathbf{entry} \supseteq \emptyset}\ [\mathrm{en}]$$

$$\frac{}{\ell_\bullet, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \mathbf{exit} \supseteq \emptyset}\ [\mathrm{ex}]$$

$$\frac{}{\ell_\mathbf{e}, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \mathbf{end} \supseteq \emptyset}\ [\mathrm{end}]$$

$$\frac{}{\ell_\circ, \ell_\mathbf{s}, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \mathbf{start} \supseteq \emptyset}\ [\mathrm{st}]$$

$$\frac{\Gamma(n) = a^\ell}{\ell, \ell, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \mathbf{node}\ n \supseteq \emptyset}\ [\mathrm{lab}]$$

$$\frac{\Gamma(n) = a^\ell \wedge \ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e} \vdash \llbracket p = c_1, \ldots, c_m \rrbracket^{a^\ell} \supseteq g}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash n \supseteq g}\ [\mathrm{cfg}]$$

Fig. 19. Semantic constraints of the `control-flow rules` in FLOWSPEC.

4.2. Transfer functions

Transfer functions for properties come from the property rules in FLOWSPEC. These rules define Analysis$_\bullet(\ell)$ in terms of Analysis$_\bullet(\ell')$. However, there can be multiple matching edges, multiple ℓ'. Therefore, we use Analysis$_\circ(\ell) = \bigsqcup_{(\ell,\ell') \in F}$Analysis$_\bullet(\ell')$ for recursive calls instead. This means that we can map our property rules onto mathematical

Transfer function mapping $\boxed{\Gamma \vdash R \Rightarrow \mathcal{F}}$

$$\frac{\Gamma \vdash \llbracket p \rrbracket^{a^\ell} = \Gamma' \wedge \Gamma' \vdash \llbracket e[n\ n_{adj} := l] \rrbracket \Rightarrow e_\lambda}{\Gamma \vdash \llbracket n\ n_{adj} \rightarrow p = e \rrbracket^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\mathrm{transfw}]$$

$$\frac{\Gamma \vdash \llbracket p \rrbracket^{a^\ell} = \Gamma' \wedge \Gamma' \vdash \llbracket e[n\ n_{adj} := l] \rrbracket \Rightarrow e_\lambda}{\Gamma \vdash \llbracket n\ p \rightarrow n_{adj} = e \rrbracket^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\mathrm{transbw}]$$

$$\frac{\Gamma \vdash \llbracket p \rrbracket^{a^\ell} = \Gamma' \wedge \Gamma' \vdash \llbracket e \rrbracket \Rightarrow e_\lambda}{\Gamma \vdash \llbracket n\ p = e \rrbracket^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\mathrm{trans}]$$

Fig. 20. Mapping of transfer functions in FLOWSPEC to Monotone Frameworks.

Expression semantics $\boxed{\Gamma \vdash e \Rightarrow \mathcal{V}}$

$$\frac{\Gamma \vdash e \Rightarrow v_i}{\Gamma \vdash \mathbf{occurrence}(e) \Rightarrow i}\ [\mathrm{occurrence}]$$

$$\frac{\Gamma \vdash \mathbf{occurrence}(e) \Rightarrow i}{\Gamma \vdash \mathbf{type}(e) \Rightarrow \mathcal{T}_i}\ [\mathrm{type}]$$

$$\frac{\Gamma \vdash \mathbf{occurrence}(n_2) \Rightarrow i \quad \wedge \vdash n_1\,\{n_2\}_i \in \mathcal{R} \quad \wedge\ \mathcal{S} \vdash n_1\,\{n_2\}_i \longmapsto n_1\,\{n_2\}_j}{\Gamma \vdash n_1\{\ n_2\ \} \Rightarrow n_1\,\{n_2\}_j}\ [\mathrm{reference}]$$

$$\frac{\Gamma \vdash \mathbf{occurrence}(n_2) \Rightarrow i \quad \wedge \vdash n_1\,\{n_2\}_i \in \mathcal{D}}{\Gamma \vdash n_1\{\ n_2\ \} \Rightarrow n_1\,\{n_2\}_i}\ [\mathrm{declaration}]$$

$$\frac{}{\Gamma \vdash n_1(n_2) \Rightarrow n_{1,\bullet}(n_2)}\ [\mathrm{property}]$$

Fig. 21. Big-step semantics of a subset of expressions in FLOWSPEC. An occurrence can only be found on expressions that evaluate to terms from the program, which have an occurrence number *i*. The static components used are: the set of references $\mathcal{R}$ and declarations $\mathcal{D}$, the scope graph $\mathcal{S}$, and the type relation $\mathcal{T}$.

$$\begin{array}{lr}
S ::= \ldots & \text{Sections}\\
\quad \mid\ \texttt{lattices}\ T_L^* & \text{Lattice definition section}\\
\quad \mid\ \texttt{types}\ T_T^* & \text{Type definition section}\\
\quad \mid\ \texttt{functions}\ T_F^* & \text{Function definition section}\\
T_L ::= n\ t^*\ \texttt{where}\ L^* & \text{Lattice definitions}\\
T_T ::= n = n\ t^*\ \{\mid\ n\ t^*\}^* & \text{(ADT) Type definitions}\\
T_F ::= n\ \{(n\texttt{:}\ t)\}^* = e & \text{Function definitions}\\
L ::= \texttt{type} = t \mid \texttt{lub}(n,\ n) = e \mid \texttt{leq}(n,\ n) = e & \text{Lattice components}\\
\quad \mid\ \texttt{bottom} = e \mid \texttt{top} = e \mid \texttt{glb}(n,\ n) = e &
\end{array}$$

Fig. 22. The types and function part of FLOWSPEC's grammar.

transfer functions, which is what we do in Fig. 20. Again, we abstract over pattern matching, and we translate expressions into lambda terms to fit the mathematical framework.

We use $\mathcal{F}$ for the transfer function space. The property rules are translated by pattern matching on the AST, then substituting all recursive calls with l, the argument name of the transfer function, and finally translating the functional code into a single mathematical expression.

A mapping from expressions to lambda terms would be a tedious exercise, therefore we separately define the big-step semantics of the interesting part of the expressions in Fig. 21. In particular we have describe to lookup of occurrences, types and names. The occurrence lookup extracts the occurrence index from a term that originated from the program. Type lookup uses the occurrence index to uniquely identify a term in the program and looks up the type in globally available type relation $\mathcal{T}$. Next to type information, we also have access to name information from static analysis, such as scope graph $\mathcal{S}$, set of reference $\mathcal{R}$, and set of declarations $\mathcal{D}$. This provides the information necessary for the two name lookup rules, which together normalize names to their declaration. A declaration is directly found in the $\mathcal{D}$, while a reference in $\mathcal{R}$ is resolved using the scope graph. These normalized names give an intuitive equality semantics for names in FLOWSPEC: names that resolve to the same declaration are the same.

In the rules for transfer functions we saw that a property rule can use the property information from the neighboring node. A property can also make use of other properties that have already been calculated. Note that this means that properties cannot depend on each other cyclically. As long as no cyclic dependency exists, we can define a property lookup that uses the property information after the effect of the node ([*property*]).

4.3. Lattices

Users of FLOWSPEC can define their own algebraic data types and lattice definitions on these types. Of the 5-tuple (⊤, ⊥, ⊑, ⊔, ⊓), ⊓ and ⊤ are not actually used by the implementation and may be left out of the lattice definition. The other three elements are called `bottom`, `leq` and `lub`. The grammar for this part of the language can be found in Fig. 22. The lattice definition contains an associated type to that it can be used in any place where a type can be used. We provide an example of a constant propagation lattice in Fig. 23. Lattices are required in the type position of a data-flow property definition. External property definitions, which may give access to other analysis information such as name sets and type are not required to hold lattices.

```
type ConstProp =
  | Top()
  | Const(int)
  | Bottom()

lattice Const where
  type = ConstProp

  lub(l, r) = match (l,r) with
    | (Top(), _) ⇒ Top()
    | (_, Top()) ⇒ Top()
    | (Const(i), Const(j)) ⇒ if i == j
      then Const(i) else Top()
    | (_, Bottom()) ⇒ l
    | (Bottom(), _) ⇒ r

  bottom = Bottom()
```

Fig. 23. A constant propagation type and lattice in FLOWSPEC. The ⊑ operation is derived from the ⊔ operation by default, although we allow both to be defined.

4.4. Built-in types and functions

FLOWSPEC has the built-in types `Set`, `Map` and `List`, and a number of built-in functions on these types. The `MaySet` and `MustSet` can technically be defined within FLOWSPEC. However, the `MustSet` needs a symbolic bottom value to represents the largest possible set. For ease of use we make `MustSet` built-in so values from the lattice can be considered sets instead of a union type of sets and the symbolic bottom value.

5. Implementation

We integrated our implementation of FLOWSPEC in the Spoofax [7] language workbench. Spoofax provides domain-specific meta-languages to declaratively specify a programming language. In this section we provide an overview of how FLOWSPEC is integrated into Spoofax and what the different parts of the FLOWSPEC implementation are.

5.1. Architecture

Consider Fig. 24. SDF3 is used for the specification of the grammar and abstract syntax, from which a parse table and different editor services are extracted [17]. The parse table is used by the parser in Spoofax to parse program text into an abstract syntax tree (AST). NaBL2 [9] is used from specifying name and type rules, based on the scope graphs [18] model that can handle many different binding patterns. With an NaBL2 specification, Spoofax can extract constraints from a program AST, which are fed to a custom constraint solving engine that builds the scope graph.

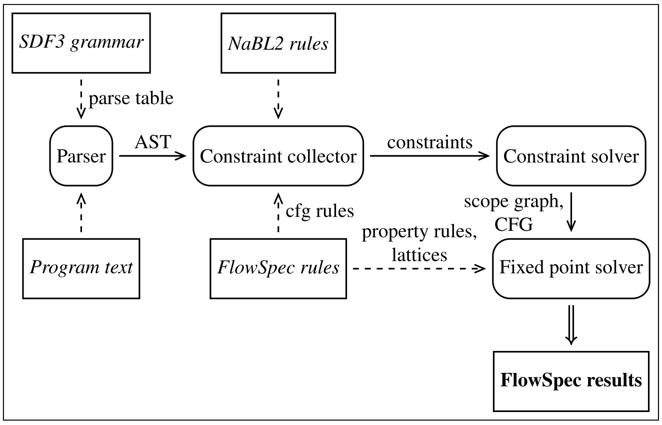


Fig. 24. Architecture diagram of FlowSpec within the context of the Spoofax language workbench. A program from an object language is first parsed using a grammar in SDF3. Then we use an NaBL2 static semantics definition to analyze the names and types in the program. The same machinery is used to build the control flow graph, based on the FlowSpec control flow graph rules. A separate fixed point solver is the new addition that computes data-flow information based on the FlowSpec specification.

FLOWSPEC is active in this same stage. Based on a FLOWSPEC specification, in particular the control-flow rules, we can automatically extract more constraints from the program AST to build the control-flow graph (CFG). The same constraint solving engine is used, which we adapted to be able to build a control-flow graph. At this point each CFG node is also associated with a transfer function.

The transfer functions, derived from the property rules, are passed to a separate fixed-point solver that we built for FLOWSPEC, along with the CFG and scope graph. Remember that name information from the scope graph is also used by FLOWSPEC. The end result is the computed data-flow properties, which can be queried in a later stage. These properties are connected to CFG nodes, which are in turn AST nodes, therefore you need only the AST node and the name of the property to request the information.

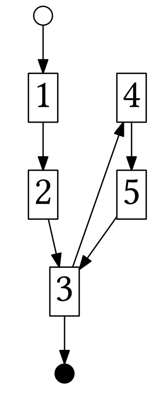
5.2. Control flow graph construction

The control-flow graph is built in two steps. First the CFG rules from a FLOWSPEC specification are used to extract edges from the program AST. The edge list is used to create the control-flow graph. At this point the control-flow graph still uses explicit artificial nodes for every `entry` and `exit`.

5.3. Data-flow solver

The data-flow solver takes in the CFG, the scope graph and the transfer functions. We apply the transfer functions through an interpreter written in the Truffle [19] framework.

The simplest version of a solving algorithm for monotone frameworks is a worklist algorithm. All nodes of the CFG are added to the worklist algorithm. When the algorithm computes a new value for a node from the worklist, all out-neighbors of that node in the CFG get re-added to the worklist. Although this is a correct algorithm, it may compute information in an inefficient order when the graph has loops. See Fig. 25 for a visual example of efficient and inefficient order.



Inefficient	Efficient
3	3
2	
1	
5	5
4	4
3	3
2	
1	
5	5
4	4
3	3
2	2
1	1

Fig. 25. An illustration of efficient and inefficient order in a backward analysis that requires two round through loop 3,4,5 before reaching a fixed point. The inefficient order always propagates from 3 to 2,1 first, whereas the efficient order first propagates from 3 to 5,4.

5.3.1. Strongly connected components

We first compute a topological ordering of strongly connected components (SCCs) in the control-flow graph [20,21]. Within each SCC we use a reverse post-order of the depth first spanning forest [22]. This ordering is more efficient in that we can compute fixed points per SCC and only propagate information to other SCCs in the graph afterwards.

It is also designed so the initial ⊥ value of the lattice is not given to the user-defined transfer functions, it only occurs in lattice operations ¬⊑ and ⊔. This can be important as in general a *must* analysis has the set of all possible values as the bottom of the lattice. This can of course be restricted to a set with all possible values from the program, but such a set would then have to be provided by the analysis author. Instead we can make sure this is not a concern by not exposing ⊥ to user-defined transfer functions, which allows us to describe ⊥ symbolically for *must* analysis. The lattice operations have clearly defined laws around ⊥

```
index := 0
S := empty stack
Q := empty stack
for ℓ ∈ E: // from the extremal labels of the CFG
  if ℓ.index is undefined:
    strongconnect(ℓ)

function strongconnect(ℓ):
  ℓ.index := index
  ℓ.lowlink := index
  index := index + 1
  ℓ.onStack := true
  // Note we don't add ℓ to the SCC stack here,
  //  but the onStack marker is still there so the
  //  algorithm works unchanged for the next loop

  for (ℓ, ℓ′) ∈ F:
    if ℓ′.index is undefined:
      strongconnect(ℓ′)
      ℓ.lowlink := min(ℓ.lowlink, ℓ′.lowlink)
    else if ℓ′.onStack:
      ℓ.lowlink := min(ℓ.lowlink, ℓ′.index)

  // Note that we add ℓ to the SCC stack here
  //  instead, in *postorder*
  S.push(ℓ)

  if ℓ.lowlink = ℓ.index:
    scc := empty array
    do:
      ℓ′ := S.pop()
      ℓ′.onStack := false
      scc.push(ℓ′)
    while(ℓ′ ≠ ℓ)
    Q.push(scc)

output: Q, the stack of SCCs
```

Fig. 26. The adapted version of Tarjan's strongly connected components, which gives topologically ordered strongly connected components (SCCs) where the SCCs have reverse postorder in their depth-first spanning tree.

without needing to look into the set, so we can implement the operation's cases with ⊥ symbolically.

The computation of the ordering uses a slightly adapted version of Tarjan's strongly connected components (SCCs) algorithm [23] in Fig. 26. Tarjan's algorithm already gives the strongly connected components in *reverse* topological order. To get the topological order out, we use a stack instead of an array to add the SCCs to when they are discovered. We also keep an extra stack where nodes of an SCC are added in *postorder*, in contrast to the set or boolean flag which is added in *preorder*. Since the algorithm already does a depth-first search, this gives us a reverse (because of the stack) postorder over the depth-first spanning forest of the SCC.

5.3.2. Solving algorithm

As our data-flow properties may (non-cyclically) depend on each other, we order the properties topologically and then solve each one in turn. The algorithm is given in pseudo-code in Fig. 27.

The first inner loop initializes the property analysis. The extremal labels `starts` of the control-flow graph *F′* are initialized with the extremal value `Prop.initial`, everything else with ⊥.

The main loop traverses the topologically ordered strongly connected components (SCCs), and uses a `while` loop to recompute the SCC if the previous iteration changed something. No worklist is necessary as any node can influence any other node.

The SCC itself is traversed in its reverse post-order. For each node ℓ in the SCC we traverse the outgoing edges (ℓ, ℓ') and use the transferred version of the property at ℓ to see if it would contribute to ℓ'. If so, the transferred property of ℓ is added to the property for ℓ' with the least-upper-bound operator.

After the main loop, the final loop uses the transfer function one more time to calculate the property just after the effect of each control-flow graph node.

```
for Prop in topologically ordered Properties:
  for $\ell$ in F:
    $\text{Prop}_\circ(\ell)$ := $\bot$

  if Prop.direction = forward:
    F' := F
  else:
    F' := F.flipped

  for $\ell$ in F'.starts:
    $\text{Prop}_\circ(\ell)$ := Prop.initial

  for scc in F'.sccs:
    done := false
    while not done:
      done := true
      for $\ell$ in scc:
        for $(\ell, \ell')$ in F'.edges:
          if $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell)) \not\sqsubseteq \text{Prop}_\circ(\ell')$:
            $\text{Prop}_\circ(\ell')$ :=
              $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell)) \sqcup \text{Prop}_\circ(\ell')$
            if $\ell'$ in scc:
              done := false

  for $\ell$ in $F'$:
    $\text{Prop}_\bullet(\ell)$ := $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell))$
```

Fig. 27. Worklist algorithm used in the implementation of FLOWSPEC. `Properties` is the list of dataflow properties. `F` is the control flow graph. f_ℓ^{Prop} is the transfer function of property `Prop` for control flow graph node ℓ. Lattice operations and values are those corresponding to the lattice of `Prop`.

5.3.3. Filtering the control-flow graph

For every data-flow property, the control-flow nodes have an associated transfer function. Most nodes in the graphs have the identity transfer function, especially `entry` and `exit` nodes. Before the solving algorithm runs, we traverse the graph once to reduce it to only the nodes that actually contribute to the solution. This is especially cheaper when we can remove nodes from a cycle in the graph. Values computed for the previous control-flow graph are propagated to those nodes which have an identity transfer function at the end of the solving phase.

6. Case study of GREEN-MARL

We evaluate the expressiveness and conciseness of FLOWSPEC with a number of case studies. So far we have presented our example analyses on a simple imperative language WHILE [1, p. 3–4]. In our first case studies we expand this toy imperative language to the full language of GREEN-MARL [24], a domain specific language for graph processing. First we introduce the domain concepts and the GREEN-MARL language.

6.1. The domain of graph analysis

In principle any relational data can be considered a graph, although binary relations are easiest to map onto nodes and edges of a graph. For higher arity relations one may employ a property graph representation, where nodes and edges can be labeled with extra information. The benefit of considering data as a graph is that standard graph algorithms can be applied to extract useful information from the data.

Large datasets from the big data world can be seen and processed as property graphs. But this requires high-performance processing, to handle the large amount of data within a reasonable amount of time. Here the issues that crop up is that a straight-forward implementation of a classic graph algorithm in a general purpose programming language usually is not able to fully exploit modern hardware for computation on large data. Both multi-core processor parallelism and multi-machine parallelism that is usually used for larger data processing requires that algorithms are mixed with bookkeeping and interoperation code, or the algorithm has to be manipulated to fit a framework.

6.2. An introduction to green-Marl

GREEN-MARL is a domain-specific language for efficient graph analysis [24]. To support its efficiency goal it provides domain-specific and non-domain specific language features so the user can expose opportunities of data-parallelism to the compiler. The style of the language is imperative so graph analysis algorithms can be written in their natural form using graph specific features and imperative loops. The compiler then applies static analysis and outputs highly optimized code for the specified graph analysis.

GREEN-MARL operates on property graphs, by accepting graphs, node-properties and edge-properties as inputs to its programs, as well as primitive data such as integers, strings and floating point numbers and collections such lists, sets and maps. While the input graph cannot be mutated in GREEN-MARL, properties can, and new properties can be created on the graph.

The graph can be iterated over using domain-specific ranges, such as the nodes or edges of the graph, or the neighbors of a node. It can also be queried for neighborhood information. Besides a standard for loop over such ranges, the language provides the parallel foreach loop, and depth- and breadth-first search traversals over graph ranges.

6.3. An example green-Marl program

Consider the GREEN-MARL program in Fig. 28. This program computes

Copyright © 2013–2015 Oracle and/or its affiliates. All rights reserved. [25]

```
1   procedure ccOne(g: graph;
2       cc: nodeProperty<double>) : bool {
3     if(g.numNodes() == 0) { // corner case empty
4       return true;
5     }
6
7     // Kosaraju (simplified)
8     nodeProperty<bool> checked;
9     g.checked = false;
10    node t = g.pickRandom();
11    inDFS(n: g.nodes from t) {
12      n.checked = true;
13    }
14    if(any(v: g.nodes) {!v.checked}) {
15      return false; // not strongly connected
16    }
17    g.checked = false;
18    inDFS(n: g^.nodes from t) {
19      n.checked = true;
20    }
21    if(any(v: g.nodes) {!v.checked}) {
22      return false; // not strongly connected
23    }
24
25    // Closeness Centrality
26    foreach(n: g.nodes) {
27      long levelSum = 0;
28      inBFS(v: g.nodes from n) {
29        levelSum += currentBFSLevel();
30      }
31      n.cc = 1.0 / (double) levelSum;
32    }
33    return true;
34  }
```

Fig. 28. Closeness Centrality (Unit Length) – Simplified.

the Closeness Centrality [26] measure on a graph, assuming all edges are the same length. Closeness Centrality of a node in a graph is the reciprocal (Line 31) of the sum of the shortest paths to every other node in the graph. For unit length edges this can be found by using breadth-first search (28–30) to visit all nodes, using the “level” of the breadth-first search as the shortest path length. Before the centrality measure is computed, a simplified version of Kosaraju’s algorithm for strongly connected components [27] is used to check that the input graph is strongly connected. This check initializes a boolean flag for each node. Then using that flag it checks that every node can be reached from a randomly picked node using a depth-first search. The flag is reset and used again, but now the depth-first search is done on the reverse graph.

Within parallel regions, such as the foreach loop and the breadth-first search, the operations are statically checked not to contain data races. The add-and-assign operator is called a reduce-assignment and is explicitly safe to perform in parallel. The language does require that no other writes or reductions with different operators are done within the same parallel section (in this case the breadth-first search).

Next to reduce-assignments, there are reduction expressions. In the example program these are used in the strongly connected check, the `any` expression used in the `if` conditions is a parallel combinator of boolean values.

6.4. The current green-Marl compiler

The current implementation of GREEN-MARL already uses the Spoofax language workbench. The GREEN-MARL compiler uses SDF3 for its grammar definition, and the older NABL name binding and TS type system languages for its static semantics implementation. The compiler uses the Stratego transformation language to analyze, optimize and generate code.

```
control-flow rules
  Block(statements) =
    entry → statements → exit

  DeferAssign(lhs, rhs, _) =
    entry → rhs → lhs → exit

  InReverse(filter, statement) =
    entry → filter → statement → exit

  InPost(filter, statement) =
    entry → filter → statement → exit
```

Fig. 29. A sample of the control flow graph rules for GREEN-MARL.

The current implementation of optimizations often have the enabling analysis embedded in that code. This makes it hard to find out what analysis is necessary, whether some analysis can be reused by other optimizations, and whether the optimization and analysis are correct. In our search for analyses and optimizations that benefit most from FLOWSPEC, we have found dead code elimination, constant propagation and loop unswitching[2]. Constant propagation is not implemented in the GREEN-MARL compiler yet because of time constraints on the compiler development team.

6.5. Control-flow graph

The whole of GREEN-MARL requires 77 control-flow rules. The rules span 165 lines of code, including comments and empty lines. Fig. 29 shows a sample of the control-flow graph code. The full list of rules can be found in Appendix A.

By comparison, back in Fig. 8 we saw 10 control-flow rules for the WHILE language, which took 18 lines of code (again including empty lines). This makes sense, given that on average most language structures have very simple control-flow expressed on a single line, followed by a blank line for readability.

In GREEN-MARL expressions are not desugared to binary and unary operations that share the same abstract syntax. We chose not to do this desugaring ourselves, which would lead to changes throughout the rest of the compiler. However, if such a change were made, 15 control-flow rules would be reduced to two, and expression related analyses would also shrink in size.

6.6. Live variables

Live variables analysis can be used to perform dead code elimination in the GREEN-MARL compiler in a more principled way. Currently dead code is discovered in an ad-hoc manner where only variables that are completely unused are removed. These variables are discovered by performing multiple tree traversals over the abstract syntax of a procedure, one to collect all local variables, and then one per variable to check that the variable is not referenced.

The FLOWSPEC implementation of live variables analysis for GREEN-MARL in Fig. 30 is the full analysis. The analysis tracks variables and property variables. A `VarAssign` is a variable reference on the left-hand side of an assignment.

Note that the control-flow rules for GREEN-MARL grew to 77 rules, compared to the 10 of WHILE, but the rules for live variables analysis only grew from 3 rules to 7 rules.

[2] Loop unswitching is an optimization that pulls a conditional program fragment out of a loop when the condition is loop-independent. After the optimization the conditional wraps two modified versions of the loop, one to be executed if the condition is true, the other if the condition is false.

```
property rules

  live(VarAssign(n) → next) =
    live(next) \ { Var{n} }

  live(PropAssign(_,p) → next) =
    live(next) \ { Prop{p} }

  live(IterBounds(n, _, _) → next) =
    live(next) \ { Var{n} }

  live(XFSIterBounds(n, _, _) → next) =
    live(next) \ { Var{n} }

  live(this@PropRef(_,p) → next) =
    live(next) ∪ { Prop{p} }

  live(this@VarRef(n) → next) =
    live(next) ∪ { Var{n} }

  live(_ → next) =
    live(next)
```

Fig. 30. Live variables analysis for GREEN-MARL.

6.7. Constant propagation

The definition of constant propagation follows the approach from Section 3.10. The full analysis implementation is available in Appendix F. Our implementation is 21 rules, each of which takes up 4 lines of code and 1 blank line for readability. Although the implementation is adequate, we find the repetition of similar `match` clauses less concise than ideal. This is an area where we believe we can still improve on the design of FLOWSPEC.

6.8. Reaching definitions

Reaching definitions analysis can be used for many applications where some form of data dependence is required. In this case we define this analysis for GREEN-MARL for the use case of loop unswitching [28]. This optimization pulls an if statement out of a loop when the condition of the if statement does not depend on the loop, something that can be discovered with reaching definitions analysis. By interchanging the if statement and loop, the loop does need to be duplicated. One version of the loop for when the if condition is true, and one for when the if condition is false.

This saves the overhead of conditional branching inside the loop, and enables the recognition of other optimization patterns for the loop. For GREEN-MARL a particular pattern, that is important when the program is compiled to a distributed setting, is the transfer of data from every node to every neighbor node. This pattern is a loop over all nodes in the graph, and within it only a loop over all neighbors of the node. Therefore if the inner loop is nested in an if statement, loop unswitching can help.

In Fig. 31 we show the rules of an enhanced Reaching Definitions analysis that explicitly tracks uninitialized variables too. This enhanced analysis would be three rules in an extended version of WHILE with variable declaration. For GREEN-MARL we have 9 rules, one of which tracks the writing to an output channel instead of a variable to keep track of data dependencies induced by the effect of printing messages.

6.9. Definite assignment

We can use definite assignment analysis in GREEN-MARL for the code generation task of initialization. When a variable is defined, it is not necessarily initialized. This is particularly of interest for variables that have a collection type, such as a set. Within GREEN-MARL the semantics is that a defined variable of type set holds an empty set. However, if the variable is later definitely assigned a set, the variable does not need to be initialized.

```
properties

  // Reaching definitions
  reaching: MaySet(term * Option(index))

property rules

  reaching(prev → this@Decl(n, _, InArg())) =
    { (n, Some(indexOf(this))) }
    ∪ reaching(prev)

  reaching(prev → this@Decl(n, _, OutArg())) =
    { (n, Some(indexOf(this))) }
    ∪ reaching(prev)

  reaching(prev → this@Decl(n, _, Local())) =
    { (n, None()) } ∪ reaching(prev)

  reaching(prev → this@VarAssign(n)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  reaching(prev → this@PropAssign(_,p)) =
    { (p, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != p }

  reaching(prev → this@IterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  reaching(prev → this@XFSIterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  // note that we model output effects like
  //  printing as writing to an artificial
  //  variable, which allows reasoning about
  //  output dependences
  reaching(prev → this@Print(_,_)) =
    { (Print(), Some(indexOf(this))) } ∪
    { (m, l) |
      (m, l) ← reaching(prev),
      m != Print() }

  reaching(prev → _) =
    reaching(prev)
```

Fig. 31. Reaching definitions analysis for GREEN-MARL.

We can use the results of reaching definitions analysis for this definite assignment analysis. The reaching definitions analysis we defined previously tracks variables from definition, marking these as uninitialized. Wherever a `(n, None())` pair is in the set, *n* is not definitely assigned there.

6.10. Available and very busy expressions

Available expressions and very busy expressions are very similar in definition, as we observed previously. We present available expressions analysis for GREEN-MARL in Fig. 32. The definition of the analysis is not particularly short, since all expressions have distinct abstract syntax that needs to be handled in a separate rule. Compared to the definition for WHILE, which needed only 3 rules, this is a rather steep increase to 24 rules. However, as we have noted before, this is due to the shape of the AST that the GREEN-MARL compiler works with, which was outside of our control.

```
properties
  available: MustSet(term)
  external refs: Set(name)

property rules
  available(prev → _) = available(prev)

  available(prev → VarAssign(n)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }

  available(prev → PropAssign(_,p)) =
    { expr |
      expr ← available(prev),
      !(Prop{p} in refs(expr)) }

  available(prev → this@IterBounds(n, _, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }

  available(prev → this@XFSIterBounds(n, _, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }

  available(prev → this@Abs(_)) =
    available(prev) ∪ { this }
  // Eliding similar lines for UMin, Mul, Div,
  //  Mod, Add, Sub, Not, Or, Eq, Gt, Lt, Geq,
  //  Leq, Neq, Cast, TerIf, FuncCall, ProcCall
```

Fig. 32. Available expressions analysis for GREEN-MARL.

6.11. *Performance measurement*

Although we do not have analyses to compare against, we can measure the current performance of the analyses we presented on typical GREEN-MARL. FLOWSPEC was designed to be a concise, *executable* specification language, where we would like the execution to be of practical use. Therefore we are not after best-in-class performance, but FLOWSPEC should have a reasonable performance for typical programs.

6.11.1. *Setup*

Our test machine has a 2.8 GHz Intel Core i7 processor, with 16 GB for main memory. It runs MacOS 10.14.2. The Java version is 1.8.0_152-b16, run on the HotSpot VM 25.152-b16. We use JMH, the OpenJDK benchmark harness library, version 1.21. Each benchmark is run with JVM arguments `-Xms512m -Xmx2g -Xss16m`, meaning the initial JVM heap size is 512 mebibytes, the maximum JVM heap size is 2 gibibytes and the JVM thread stack size is 16 mebibytes.

We run 5 warmup iterations, 10 seconds each, after which we run 5 measurement iterations. The benchmark sets up all required dependencies beforehand.

6.11.2. *Inputs*

We gathered three typical size GREEN-MARL programs: Closeness Centrality[3], Closeness Centrality with edge weights, and Betweenness Centrality[4]. The characteristics of these inputs are in Fig. 33, where lines of code (LOC) are measured without blank lines and comments, and the bytes of input are in the intermediate representation on disk from which they are read for input to the benchmarks.

Input name	LOC	Bytes of input
CC	22	1892
BC	17	2689
CC weighted	41	6963

Fig. 33. Benchmark inputs and their characteristics.

Input name	Live Variables	Reaching Definitions
CC	0.9	1.4
BC	1.3	2.8
CC weighted	2.7	10.7

Fig. 34. Analysis time in milliseconds for implementations of Live Variable and Reaching Definitions in FLOWSPEC on typical GREEN-MARL programs.

6.11.3. *Results*

We present the measurement results in Fig. 34, where each number is the arithmetic mean of the five measurements for that benchmark. Given the low analysis time for these typical size inputs, we are confident that FLOWSPEC analyses can be effectively used within the GREEN-MARL compiler. In Section 7.4 we mention some more optimization options we have to improve the performance of FLOWSPEC even further.

7. Case study of STRATEGO

We evaluate the expressiveness and conciseness of FLOWSPEC with a number of case studies. So far we have only presented example analyses for typical imperative programming languages. In this section we present a case study of an analysis written in FLOWSPEC for a programming language with a different paradigm: the STRATEGO [30] term rewriting language. We show how we specified a reaching definitions analysis in FLOWSPEC and compare it to the existing implementation of reaching definitions in the STRATEGO compiler, which is itself written in STRATEGO. We compare not only the implementation from a source code perspective but also give a performance comparison between the FLOWSPEC implementation and the implementation in STRATEGO.

First we introduce the domain concepts and the STRATEGO language.

7.1. *Term rewriting and* STRATEGO

STRATEGO is a language for program transformation. The language has features for defining *rewrite rules*, and *strategies* for the application of those rewrite rules. Given an Abstract Syntax Tree that presents a program, a Stratego program can transform terms from the tree with rewrite rules and apply them in the right places in the right order with strategies.

Any rule or strategy can fail to apply. Special strategy combinators allow recovery from failure, which looks similar to an if-else but based on failure or success instead of a boolean value. Rewrite rules can be expressed as strategies, and in fact the STRATEGO compiler desugars all features down to a core language that consists only of strategies.

STRATEGO Core

STRATEGO core consists of a list of strategy definitions. Each strategy definition has zero or more strategy arguments (functions, making the strategy higher-order), zero or more term arguments (data), and an implicit argument: the current term. The body of a strategy definition is a strategy expression, which can consist of:

1. `fail`, the primitive strategy that fails
2. `id`, the primitive strategy that succeeds and does nothing to the current term
3. pattern-match, matches a pattern against the current term, possibly failing or binding variables when it succeeds

[3] as previously shown in Fig. 28.

[4] Gathered from [29].

4. pattern-build, replaces the current term with another term, possibly failing when using an unbound variable
5. scope, defines a scope with a list of fresh (unbound) variables
6. sequence, apply one strategy expression after the other
7. guarded choice, the if-then-else lookalike based on failure instead of boolean values
8. `one`, `some`, `all`, three language constructs that take a strategy argument and apply that strategy on one, some or all of the children of the current term.
9. strategy call, to call another named strategy
10. let, to define local strategies

7.2. Control-Flow

With the list of constructs, we are able to define the control-flow of STRATEGO core, as seen in Fig. 35.

Note how the guarded choice rule specifies two paths, one for the condition and then branch, one for the else branch *without* the condition. This is something particular to Stratego, where variable bindings from the condition are backtracked when the condition fails at some point.

All call-like constructs have control-flow that optionally goes into the strategy arguments. This models the uncertainty of whether those strategy expressions are executed or not.

```
control-flow rules
  root SDefT(_, sargs, targs, body) =
    start → sargs → targs → body → end

  node VarDec(_, _)

  Let(defs, body) =
    entry → defs → body → exit

  SDefT(_, sargs, targs, body) =
    entry → exit,
    entry → sargs → targs → body → exit

  CallT(_, sargs, targs) =
    entry → targs → this → exit,
                    this → sargs → exit

  node Fail()
  node Id()
  node Match(_)
  node Build(_)

  Scope(_, body) =
    entry → this → body → exit

  Seq(first, second) =
    entry → first → second → exit

  GuardedLChoice(c, t, e) =
    entry → c → t → exit,
    entry → e → exit

  Some(s) =
    entry → this → exit,
            this → s → exit

  One(s) =
    entry → this → exit,
            this → s → exit

  All(s) =
    entry → this → exit,
            this → s → exit
```

Fig. 35. Most of the control flow graph rules for STRATEGO.

```
properties

  reachingDefinitions:
    MaySet(name * Option(position))

property rules

  reachingDefinitions(_.start) = {}

  reachingDefinitions(prev → v@VarDec(n, _)) =
    reachingDefinitions(prev)
    ∪ {(Var{n}, Some(position(v)))}

  reachingDefinitions(prev → Scope(names, _)) =
    reachingDefinitions(prev)
    ∪ { (Var{n}, None()) |
          n ← Set.fromList(names) }

  reachingDefinitions(prev → m@Match(t)) =
    { (n, p) |
      (n, p) ← rdprev,
      !(n in pv && p == None()) }
    ∪ { (nm, Some(position(m))) |
          nm ← pv,
          (nm, None()) in rdprev }
  where rdprev = reachingDefinitions(prev)
        pv = patternVars(t)
```

Fig. 36. Reaching definitions analysis for STRATEGO.

7.3. Reaching definitions

Reaching definitions is used in a number of places in the current STRATEGO compiler, which is written in STRATEGO. Whether a variable is guaranteed to be bound or unbound at some point in a STRATEGO program is valuable to given errors messages (e.g. on build a pattern with an unbound variable) and to generate efficient code (e.g. elide a check and variable binding code when pattern matching against an always bound variable).

Our reaching definitions analysis for STRATEGO, written in FLOWSPEC is given in Fig. 36. We start without bindings, add reaching definitions for arguments to strategies, add uninitialized variables for scopes, and replace uninitialized variables with their initialization when they are first matched.

For comparison, the original analysis consists of 232 lines of STRATEGO code (provided in Appendix G) that implement reaching definitions analysis under the name `bound-unbound-vars` in the current STRATEGO compiler. Our implementation in FLOWSPEC is only 19 lines.

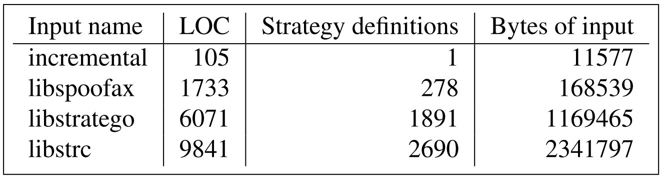
The STRATEGO implementation uses a feature called dynamic rules [31] to implement both the data-flow analysis and specify the name and scope rules for STRATEGO in an ad-hoc fashion. Other analyses in the STRATEGO compiler repeat this name and scope structure in a similar way. Notably, this code analyzes a subset of STRATEGO but slightly more than STRATEGO core. This is most likely a legacy code issue. Currently this analysis is called within the compiler at a point where the AST has been reduced to STRATEGO core already.

7.4. Performance comparison

Since we have a STRATEGO implementation and FLOWSPEC implementation of the same analysis, we can do a performance comparison.

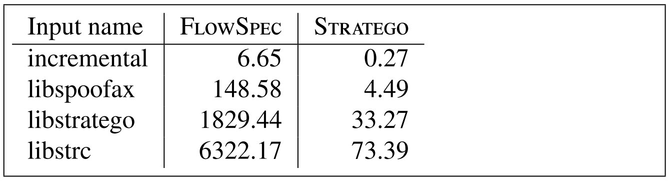
7.4.1. Setup

Our test machine has a 2.8 GHz Intel Core i7 processor, with 16 GB for main memory. It runs MacOS 10.14.2. The Java version is 1.8.0_152-b16, run on the HotSpot VM 25.152-b16. We use JMH, the OpenJDK benchmark harness library, version 1.21. Each benchmark is



Input name	LOC	Strategy definitions	Bytes of input
incremental	105	1	11577
libspoofax	1733	278	168539
libstratego	6071	1891	1169465
libstrc	9841	2690	2341797

Fig. 37. Benchmark inputs and their characteristics.



Input name	FLOWSPEC	STRATEGO
incremental	6.65	0.27
libspoofax	148.58	4.49
libstratego	1829.44	33.27
libstrc	6322.17	73.39

Fig. 38. Analysis time in milliseconds for our implementation in FLOWSPEC and the optimized STRATEGO implementation in the current STRATEGO compiler.

run with JVM arguments `-Xms512m -Xmx2g -Xss16m`, meaning the initial JVM heap size is 512 mebibytes, the maximum JVM heap size is 2 gibibytes and the JVM thread stack size is 16 mebibytes.

We run 5 warmup iterations, 10 seconds each, after which we run 5 measurement iterations. The benchmark sets up all required dependencies beforehand. The actual measured code is the STRATEGO strategy in the STRATEGO case and the FLOWSPEC analysis in the FLOWSPEC case.

7.4.2. Inputs

We gathered input program of different size, from the typical size for the incremental STRATEGO compiler (a single strategy), up to the largest STRATEGO library we know of. The characteristics of these inputs are in Fig. 37, where lines of code (LOC) are measured without blank lines and comments, and the bytes of input are in the intermediate representation on disk from which they are read for input to the benchmarks.

7.4.3. Results

We present the benchmark results in Fig. 38, where each number is the arithmetic mean of the five measurements for that benchmark.

FLOWSPEC has reasonable execution times for typical workloads. We do see the execution time grow rather quickly of very large workloads. We see that from the libspoofax input to the libstratego input is a 7x growth in raw bytes of input, the STRATEGO implementation of the analysis spends 8x the time on that input, but the FLOWSPEC implementation spends 26x milliseconds.

We are aware of some of the causes for this behavior. FLOWSPEC builds up and saves all control-flow graphs to save these as part of the analysis results, as well as the data-flow analysis information. In contrast, the STRATEGO implementation is manually written in such a way that the analysis information is used and forgotten as soon as possible. FLOWSPEC also currently runs an AST interpreter for the data-flow rules, which takes a majority of time. The STRATEGO implementation is compiled entirely.

Correctness: We compared the outcomes of the two analyses to each other. The STRATEGO implementation immediately transforms the AST by annotating each variable use with its estimated state: bound, unbound, or maybe bound. Once the FLOWSPEC analysis is finished, the code to add such annotations based on the FLOWSPEC analysis is trivial[5].

7.4.4. Threats to validity

Although this is a small benchmark, more for the sake of curiosity than validation, we still address the threats to validity of the measurements.

External Validity: A threat to the generalizability of these measurements is how we compared only a single analysis (Reaching Definitions) with a single language (STRATEGO). In fact, intra-procedural analysis of STRATEGO gives rise to acyclic control-flow graphs. This is not at all representative of typical control-flow graphs. However, this was the analysis and language that were easily available for comparison against an older implementation.

Internal Validity: The comparison we make here is that of end-goal usage, not exactly the same analysis. The STRATEGO analysis is both analysis and transformation, combined by hand. This combination is a specialization that does well in performance, although we argue that it is not good for maintainability. On the other hand, FLOWSPEC computes and returns a control-flow graph, and computes and returns all Reaching Definitions information for the entire program. This is more work, more information that can be used for multiple purposes. And yet the result is not the transformed program.

Construct Validity: Finally, we measure performance on the JVM and need to consider JIT compilation and garbage collection. Thankfully the JMH framework takes care of warmup for the JIT and garbage collection between iterations. We could not eliminate background noise entirely, but all measurement iterations looked to be close to each other. The biggest open question is that of the three phases benchmarked separately which do not sum up to the whole benchmark.

8. Related work

We will shortly discuss the history of monotone frameworks which underlies our work, and some other systems and formalisms for implementing data-flow analysis. Some of the aspects we discuss are summarized in Fig. 39.

8.1. Monotone frameworks

Monotone data-flow analysis frameworks [16] were first introduced as a generalization over Killdall's lattice theoretic approach to data-flow analysis [32]. By replacing the distributivity requirement with a monotonicity requirement for the transfer function, Kam and Ullman found a way to describe more flow problems in a framework with a clear solution by maximal fixed point. This maximal fixed point can be iteratively computed with a simple worklist algorithm. As mentioned in Section 2, FLOWSPEC is based on this analysis framework.

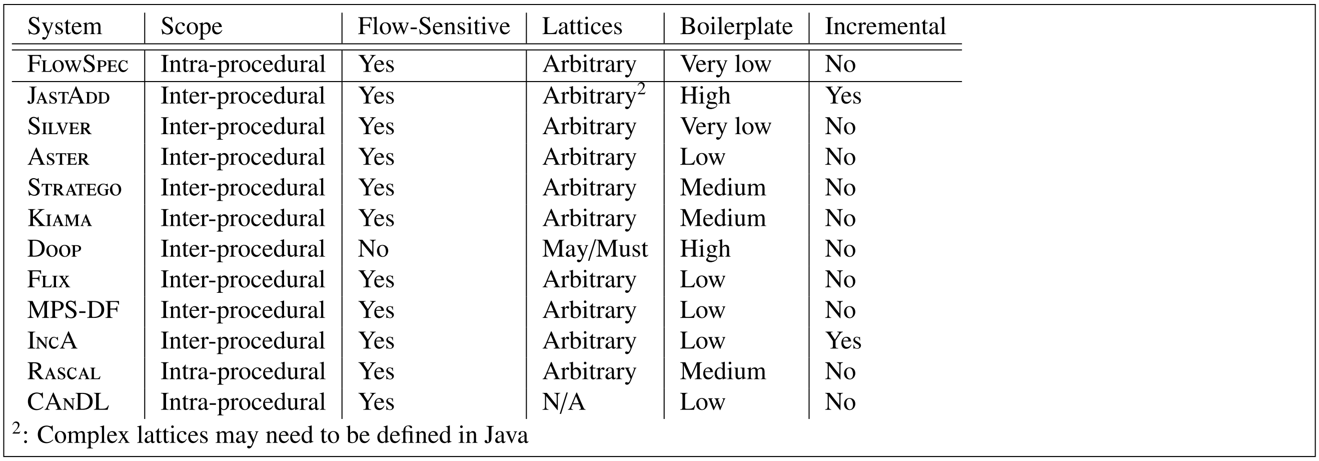
8.2. Attribute grammars

JastAdd: The JASTADD system [33] supports attribute grammars [34] extended with a number of special attributes which allows a declarative intra-procedural control- and data-flow analysis specification [35]. In particular, these are reference attributes [36] for control-flow graph (CFG) edges, higher-order attributes [37] for virtual CFG nodes, used for entry/exit of methods, circular attributes [38] for fixed points of data-flow equations, and collection attributes [39] e.g. for the CFG where there are multiple successors.

Note that each feature can be used for data-flow analysis but is not specifically designed for it. Therefore JASTADD is a much more general computation system that has much more expressive power than FLOWSPEC. The downside of this generality, versus the domain-specific nature of FLOWSPEC, is the verbosity. Whereas in FLOWSPEC our specifications are small and the language provides domain terms for each of the features, JASTADD requires an encoding in the different attributes. It is also not clear to us whether higher-order attributes are enough to encode arbitrary lattices. If not, JastAdd's Java integration would be required to implement the lattice.

Silver: SILVER [40] is another attribute grammar system and specification language that supports similar features to the JASTADD system. However, for control- and data-flow analysis, it provides dedicated

[5] We used 23 lines of code of STRATEGO code.



System	Scope	Flow-Sensitive	Lattices	Boilerplate	Incremental
FlowSpec	Intra-procedural	Yes	Arbitrary	Very low	No
JastAdd	Inter-procedural	Yes	Arbitrary[2]	High	Yes
Silver	Inter-procedural	Yes	Arbitrary	Very low	No
Aster	Inter-procedural	Yes	Arbitrary	Low	No
Stratego	Inter-procedural	Yes	Arbitrary	Medium	No
Kiama	Inter-procedural	Yes	Arbitrary	Medium	No
Doop	Inter-procedural	No	May/Must	High	No
Flix	Inter-procedural	Yes	Arbitrary	Low	No
MPS-DF	Inter-procedural	Yes	Arbitrary	Low	No
IncA	Inter-procedural	Yes	Arbitrary	Low	Yes
Rascal	Intra-procedural	Yes	Arbitrary	Medium	No
CAnDL	Intra-procedural	Yes	N/A	Low	No

[2]: Complex lattices may need to be defined in Java

Fig. 39. Related work summary table.

syntax which translates to a control-flow graph and temporal logic formulae (CTL-FV) that are offloaded to a model checker (NuSMW). Temporal logic can express reasoning in terms of time, which can be used to express data-flow properties in a declarative manner.

Temporal logic is a very terse notation for data-flow specification, and is subjectively not very easy to read. Our language design for flowspec is a very different approach which is not rooted in logic formulae. Instead we use domain names for keywords and provide a declarative approach to specification which borrows from functional programming.

The Silver publication did not report on the performance of their data-flow analysis approach. Model checkers have a sweet spot where their heuristics perform well, but ultimately cannot cover the entire NP-Hard problem space. As such, it may suddenly perform poorly for the problems that Silver generates for it based on the translation Silver uses, the temporal logic formula in the Sivler specification, or the particular input program.

Aster: The stratego strategic programming language was extended with attribute grammars in aster [41]. aster allows for attribute decorators that allow the user to program different attribute grammar extensions, which allows it to support declarative flow analysis similar to jastadd.

Stratego: The stratego programming language was also directly applied to data-flow analysis by leveraging its dynamic rewrite rules [31]. In this paper the authors apply a combination of rewrite rules and dynamic rules for dynamic propagation of information. Dynamic rules can use either union or intersection to follow control-flow that splits and merges. At the splitting point the dynamic rule is copied to both branches. In all other places dynamic rules are mutated, which is not an issue as the rewrite based on the dynamic information is done immediately. Fixed point calculation can also be done with a similar choice of union or intersection.

In flowspec we treat data-flow analysis as a separate concern. In contrast an analysis in stratego is usually interspersed in the transformation code, which makes the code more difficult to read and understand. This code style also brings frustrations when an analysis already interleaved in a transformation turns out to be more generally useful in other transformations. Extracting and reusing such an analysis is not well supported by dynamic rules. flowspec is built around the idea of simple, separate specifications of data-flow analysis. The analysis results can be used to inform an arbitrary amount of transformations.

Kiama: Kiama [42] is a language processing library in Scala, based on attribute grammars and strategic programming. The interesting property Kiama has over aster is the provisions for updating analyses after transformation, a concern we currently do not address in flowspec. The tree transformations done with strategic programming can invalidate the values of certain attributes that are dependent on the parents of a tree node (e.g. inherited attributes), or some other context. To easily combine attribute grammars with strategic programming, Kiama provides tree-indexed attribute families. The root of the particular tree is used for indexing whenever an attribute is context-dependent.

8.3. Relational programming

Doop: The doop framework [43] uses a datalog dialect for a declarative specification of static analyses such as context-sensitive pointer analysis. In a recent tutorial [44, p. 46], Smaragdakis and Balatsouras explain different techniques specific to pointer analysis with datalog examples. These mostly focus on whole-program, flow-insensitive may-analyses. Flow-sensitive analyses and must-analyses are significantly more complex and harder to ensure soundness of.

flowspec focusses on a complementary set of data-flow analysis. Instead of whole-program (inter-procedural) flow-insensitive may-analyses, flowspec provides support for local (intra-procedural) flow-sensitive analyses with arbitrary lattices.

Flix: The flix programming language [45] is a new contender that extends datalog to a language with user-defined lattices, and monotonic transfer and filter functions on these lattices. These allow Flix to express data-flow analysis with infinite value domains while keeping guaranteed termination with a unique minimal model; under the assumption that the user-defined lattices and functions are defined correctly.

User-defined types and lattices in flix and flowspec are very similar. flowspec benefits from the larger Spoofax ecosystem, to develop features such as the (experimental) automatic name abstraction. One may be able to provide name and scope information along with an input program in flix, and use explicit filtering, but to our knowledge there is no way to automatically filter names that go out of scope.

8.4. Other analysis approaches

MPS-DF: The MPS language workbench[6] has MPS-DF, an extensible framework for definition of data-flow analyses [46]. MPS-DF has support for building data-flow graphs (control-flow graphs with *read* and *write* primitives), and a syntax for writing transfer and confluence operators. These operators form the ingredients that allows MPS-DF to apply a classical monotone frameworks solution. The analysis can be done in an intra-procedural fashion by correctly implementing the operators to abstract over the possible effects of a procedure call, or inter-procedurally by inlining method calls. To support this variability,

[6] https://www.jetbrains.com/mps/

two different data-flow graph builders need to be implemented for a procedure call element in the AST.

IncA: Another MPS related language is INCA [47], a DSL for incremental program analysis. This DSL is built upon the InQuery engine which supports incremental computations using first order logic extended with the least fixed point operator. The language originally only worked for analyses that can be modeled with relations (i.e. may- and must-analysis). It did not support the generation of data that is not directly from the program, such as building intervals in an interval analysis. This was remediated by extending the incremental algorithm for lattice based values [48]. The language design of the INCA DSL evokes a procedural style, whereas FLOWSPEC uses a declarative style with domain terms.

FLOWSPEC makes a different trade-off than INCA. We do not support incremental analysis, thereby also avoiding the prohibitive memory overhead of INCA, which the authors mention as a concern for future work. The benefit of INCA is that data-flow analysis can be used for rapid feedback to a user in an IDE setting. This fits well with inter-procedural analysis. With FLOWSPEC our focus has been on analyses that inform optimization, which are done in a compiler backend.

Rascal: RASCAL [49] provides a facility for control-flow graph construction with DCFLOW [50], a domain-specific language. It simplifies the definition of simple control-flow constructions, but does not support abrupt termination such as exceptions. To implement these constructs the user needs to fall back on the DCFLOW library in RASCAL. Similarly, the actual implementation of data-flow algorithms on top of a CFG is still done in the RASCAL language, without a special library or framework for the use-case.

FLOWSPEC support both control-flow graph construction and data-flow analysis within the domain-specific language. RASCAL, as a general-purpose language, can support anything, but without extra support for the use case of data-flow analysis.

CAnDL: The domain specific language CANDL [51] provides a constraint based approach to compiler analysis for LLVM. It is specifically designed for the LLVM intermediate representation, which is in single static assignment (SSA) form. The focus is on programmer productivity, and in their evaluation the authors show several real world use cases where analyses were expressed more briefly in CANDL than in the original C + + of LLVM or in Polly, a polyhedral optimization framework.

FLOWSPEC makes no assumptions on the representation of the program or intermediate representation it analyses. We provide a language parametric analysis DSL, where we cannot make assumptions about a representation such as SSA form. Instead of the specific constraints of CANDL, we provide property rules where the user can propagate information of their choosing.

9. Future work

Currently we describe control-flow with purely local rules that can be solved before the start of data-flow analysis. To allow breaks from loops and jumps to labels we would like to extend the specification, so it may use properties and name resolution to gain access to non-local jump targets. This could also be used for static dispatched procedure calls, although this could result in rather large control-flow graphs.

In general the interaction between names, control- and data-flow, and types is of interest. We integrated FLOWSPEC in Spoofax, which has domain specific support for name binding [52]. The theoretical foundation for the newest name binding support [18] gives an interesting model of scope graphs. The combination of scope graphs and control-flow graphs may be enough to fully describe a program to the point that we no longer need the abstract syntax tree.

At the same time the constraint language for scope graphs [9] can also model types of a programming language. If we can fully integrate our control- and data-flow work in this framework we can extend the expressiveness of the system to have name resolution or types that depend on control- and data-flow.

Data dependencies can be discovered with data-flow analysis. We believe this data dependency information can be used to relax the strict ordering in the control-flow graph, and that this will improve discovery of optimizable patterns and reasoning about optimizations.

The safety of the user-defined lattices and property rules is an important issue. On lattices of infinite height or with non-monotone transfer functions, we cannot guarantee termination of our implementation. There may be opportunities to generate proof obligations to be proven by the user, or even pass it an automatic theorem prover. The proof obligations may also be usable for randomized testing.

Another opportunity for further research would be to verify the correctness of control- and data-flow specifications relative to a dynamic semantics specification.

10. Conclusion

We have presented FLOWSPEC, a declarative specification language for the domain of data-flow analysis. We have shown its static semantics, and its dynamic semantics as a mapping onto monotone frameworks. The implementation of FLOWSPEC is integrated into the Spoofax language workbench where it can access name information to take into account during analyses. We have demonstrated a number of example specifications in FLOWSPEC. We have also demonstrated FLOWSPEC in a case study of an industrial domain-specific language with domain-specific analyses and a case study of a term transformation language.

In short, FLOWSPEC provides domain-specific, integrated support for data-flow analysis in compilers. With it, we can remove ad-hoc analyses and provide more maintainable compilers in the future. [25]

Declaration of Competing Interest

The authors declare that they do not have any financial or non-financial conflict of interests.

Acknowledgements

We would like to thank Peter Mosses, the anonymous reviewers of SLE'17 and the anonymous reviewers of COMLAN'18 for their valuable feedback and suggestions. This research was partially funded by the NWO VICI Language Designer's Workbench project (639.023.206) and by a gift from the Oracle Corporation.

Appendix A. Control-flow graph rules for GREEN-MARL

The control-flow graph rules are roughly ordered by the corresponding SDF3 files that define the abstract syntax that we match. The file starts with a module definition and the import of the external signature definitions. Then we define general rules for `Cons` and `Nil`, control-flow in lists

```
module control-data

imports
  external
    signatures/-
    signatures/preprocess/Extra-Constructors-sig
    signatures/post-analysis/-
    signatures/frontend/syntax/core/-

control-flow rules // Library rules?

  Cons(head, tail) =
    entry → head → tail → exit

  Nil() = entry → exit

control-flow rules // gm_lang

  root Proc(_, params, _, _, body) =
    start → params → body → end

control-flow rules // Statements

  Block(statements) =
    entry → statements → exit

  ReduceAssign(lhs, _, rhs, _) =
    entry → rhs → lhs → exit

  ArgMinMax(lhs, lhss, _, rhs, rhss, _) =
    entry → rhs → rhss → lhs → lhss → exit

  ForEach(_, bounds, statement) =
    entry → bounds → exit,
             bounds → statement → bounds

  BFS(bounds, statement, rev) =
    entry → bounds → exit,
             bounds → statement → bounds,
             bounds → rev → bounds

  DFS(bounds, statement, post) =
    entry → bounds → exit,
             bounds → statement → bounds,
             bounds → post → bounds
```

Fig. A1. The control-flow graph rules for GREEN-MARL (1/4).

assume that each element of the list can control-flow and threads the control-flow through the list from left to right. The root of a control-flow graph in GREEN-MARL is at the procedure level. Blocks have lists of statements, so their control-flow is that of the list. In all kinds of assignments, such as the reduce assignment and the arg-min assignment, the right-hand side expressions are executed before the left-hand side. Loops and traversals show up as loops in the control-flow graph as well. The bounds stand in for the decision to go into the body or go on with the program that follows the loop/traversal.

Rules like the one for NoInReverse() do not contribute nodes to the control-flow graph. Procedure calls execute their expressions, then do the call itself by using the matched AST node as a control-flow graph node with the this keyword. Printing has side-effects and is therefore also itself put in the control-flow graph. Returns use the end keyword instead of the local exit to connect to the end of the procedure enclosing (since it declared itself a root). When an AST node is directly a control-flow graph node and has no further control-flow inside, we can use the shortcut rule `node` followed by the AST pattern.

```
  NoInReverse() = entry → exit

  InReverse(filter, statement) =
    entry → filter → statement → exit

  NoInPost() = entry → exit

  InPost(filter, statement) =
    entry → filter → statement → exit

  CallStm(call) = entry → call → exit

  ProcCallStm(_, exprs, outargs) =
    entry → exprs → this → outargs → exit

  Print(_, exprs) = entry → exprs → this → exit

  Error(expr) = entry → expr → this → end

  ReturnWith(_, expr) =
    entry → expr → end

  Return() = entry → end

control-flow rules // Declarations (post-analysis)

  node Decl(_, _, _)

control-flow rules // Statements (core)

  IfThenElse(cond, thenb, elseb) =
    entry → cond → thenb → exit,
            cond → elseb → exit

  While(cond, body) =
    entry → cond → exit,
    cond → body → cond

  DoWhile(body, cond) =
    entry → body → cond → exit,
    cond → body

  Assign(lhs, rhs) =
    entry → rhs → lhs → exit

  node VarAssign(_)

  PropAssign(r,_) = entry → r → this → exit

  ElementAssignWrapper(ea, _) =
    entry → ea → exit
```

Fig. A2. The control-flow graph rules for GREEN-MARL (2/4).

```
  ElementAssign(ea,expr) =
    entry → expr → ea → exit

control-flow rules // Iterators

  IterBounds(_, range, order) =
    entry → range → order → this → exit

  NoOrder() = entry → exit

  OrderBy(expr, _) =
    entry → expr → exit

  node GraphIterRange(_,_)
  node NodeIterRange(_,_)
  node CollectionIterRange(_,_)
  node MapIterRange(_,_)

  VectorIterRange(e) = entry → e → this → exit

  XFSIterBounds(_, range, nav) =
    entry → range → nav → this → exit

  node BFSRange(_,_,_)
  node DFSRange(_,_,_)

  NoNavigator() = entry → exit

  Navigator(expr) =
    entry → expr → exit

control-flow rules // Expressions

  node IntLit(_)
  node LongLit(_)
  node FloatLit(_)
  node DoubleLit(_)
  node StringLit(_)
  node NIL()
  node Inf(_, _)
```

Fig. A3. The control-flow graph rules for GREEN-MARL (3/4).

```
  node True()
  node False()

  Abs(e) = entry → e → this → exit

  node VarRef(_)
  Placeholder() = entry → exit
  node PropRef(_, _)
  ElementAccess(ea, e) = entry → e → ea → exit

  UMin(e) = entry → e → this → exit
  Mul(e1,e2) = entry → e1 → e2 → this → exit
  Div(e1,e2) = entry → e1 → e2 → this → exit
  Mod(e1,e2) = entry → e1 → e2 → this → exit
  Add(e1,e2) = entry → e1 → e2 → this → exit
  Sub(e1,e2) = entry → e1 → e2 → this → exit
  Not(e) = entry → e → this → exit
  And(e1,e2) = entry → e1 → e2 → this → exit
  Or(e1,e2) = entry → e1 → e2 → this → exit
  Eq(e1,e2) = entry → e1 → e2 → this → exit
  Gt(e1,e2) = entry → e1 → e2 → this → exit
  Lt(e1,e2) = entry → e1 → e2 → this → exit
  Geq(e1,e2) = entry → e1 → e2 → this → exit
  Leq(e1,e2) = entry → e1 → e2 → this → exit
  Neq(e1,e2) = entry → e1 → e2 → this → exit

  Cast(_,e) = entry → e → this → exit
  TerIf(e1,e2,e3) = entry → e1 → e2 → exit,
                              e1 → e3 → exit

control-flow rules // Common

  FuncCall(e1, _, e2) =
    entry → e1 → e2 → this → exit
  ProcCall(_, e, _) =
    entry → e → this → exit

  NoOutArgs() = entry → exit
  OutArgs(outargs) = entry → outargs → exit
  Ignore() = entry → exit
```

Fig. A4. The control-flow graph rules for GREEN-MARL (4/4).

Appendix B. Live variables analysis rules for GREEN-MARL

In the figure is a definition of a live variables that tells you which variables may be read before being re-assigned. The Set contains the term that is the name string from the AST. At any assignment the name is removed. At a reading point the name is added. Information is propagated backwards so that you can look into the future of the program when reading the analysis results.

```
properties

  // Live Variables
  live: MaySet(term)

property rules

  live(VarAssign(n) → next) =
    live(next) \ { Var{n} }

  live(PropAssign(_,p) → next) =
    live(next) \ { Prop{p} }

  live(IterBounds(n, _, _) → next) =
    live(next) \ { Var{n} }

  live(XFSIterBounds(n, _, _) → next) =
    live(next) \ { Var{n} }

  live(this@PropRef(_,p) → next) =
    live(next) ∪ { Prop{p} }

  live(this@VarRef(n) → next) =
    live(next) ∪ { Var{n} }

  live(_ → next) =
    live(next)
```

Fig. B1. A live variables analysis for GREEN-MARL.

Appendix C. Reaching definitions analysis rules for GREEN-MARL

Reaching Definitions is similar to the previous analysis but records writes to a variable and passes these forwards. Therefore you can use this analysis at a point where a variable is read to see where the value read there may have originated from. Because local variable declarations are given a “write” of None, you can use this information to track down places where a variable may be uninitialized as well. As a separate analysis this is usually known as Definite Assignment analysis.

```
properties

  // Reaching definitions
  reaching: MaySet(term * Option(index))

property rules

  reaching(prev → this@Decl(n, _, InArg())) =
    { (n, Some(indexOf(this))) } ∪
reaching(prev)

  reaching(prev → this@Decl(n, _, OutArg())) =
    { (n, Some(indexOf(this))) } ∪
reaching(prev)

  reaching(prev → this@Decl(n, _, Local())) =
    { (n, None()) } ∪ reaching(prev)

  reaching(prev → this@VarAssign(n)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  reaching(prev → this@PropAssign(_,p)) =
    { (p, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != p }

  reaching(prev → this@IterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  reaching(prev → this@XFSIterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } ∪
    { (m, l) | (m, l) ← reaching(prev), m != n }

  // note that we model output effects like
  //  printing as writing to an artificial
  //  variable, which allows reasoning about
  //  output dependences
  reaching(prev → this@Print(_,_)) =
    { (Print(), Some(indexOf(this))) } ∪
    { (m, l) |
      (m, l) ← reaching(prev),
      m != Print() }

  reaching(prev → _) =
    reaching(prev)
```

Fig. C1. A Reaching Definitions analysis for GREEN-MARL.

Appendix D. Available expressions analysis rules for green-marl

```
properties
  available: MustSet(term)
  external refs: Set(name)

property rules
  available(prev → _) = available(prev)

  available(prev → VarAssign(n)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }

  available(prev → PropAssign(_,p)) =
    { expr |
      expr ← available(prev),
      !(Prop{p} in refs(expr)) }

  available(prev → this@IterBounds(n, _, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }

  available(prev → this@XFSIterBounds(n, _, _)) =
    { expr |
      expr ← available(prev),
      !(Var{n} in refs(expr)) }
```

Fig. D1. Available expressions analysis for green-marl (1/2).

```
  available(prev → this@Abs(_)) =
    available(prev) ∪ { this }
  available(prev → this@UMin(_)) =
    available(prev) ∪ { this }
  available(prev → this@Mul(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Div(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Mod(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Add(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Sub(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Not(_)) =
    available(prev) ∪ { this }
  available(prev → this@Or(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Eq(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Gt(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Lt(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Geq(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Leq(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Neq(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@Cast(_,_)) =
    available(prev) ∪ { this }
  available(prev → this@TerIf(_,_,_)) =
    available(prev) ∪ { this }
  available(prev → this@FuncCall(_,_,_)) =
    available(prev) ∪ { this }
  available(prev → this@ProcCall(_,_,_)) =
    available(prev) ∪ { this }
```

Fig. D2. Available expressions analysis for green-marl (2/2).

Appendix E. Very Busy expressions analysis rules for green-marl

```
properties
  veryBusy: MustSet(term)

property rules
  veryBusy(_ → next) = veryBusy(next)

  veryBusy(VarAssign(n) → next) =
    { expr |
      expr ← veryBusy(next),
      !(Var{n} in refs(expr)) }

  veryBusy(PropAssign(_,p) → next) =
    { expr |
      expr ← veryBusy(next),
      !(Prop{p} in refs(expr)) }

  veryBusy(this@IterBounds(n, _, _) → next) =
    { expr |
      expr ← veryBusy(next),
      !(Var{n} in refs(expr)) }

  veryBusy(this@XFSIterBounds(n, _, _) → next) =
    { expr |
      expr ← veryBusy(next),
      !(Var{n} in refs(expr)) }
```

Fig. E1. Very Busy Expressions analysis for green-marl (1/2).

```
  veryBusy(this@Abs(_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@UMin(_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Mul(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Div(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Mod(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Add(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Sub(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Not(_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Or(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Eq(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Gt(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Lt(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Geq(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Leq(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Neq(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@Cast(_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@TerIf(_,_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@FuncCall(_,_,_) → next) =
    veryBusy(next) ∪ { this }
  veryBusy(this@ProcCall(_,_,_) → next) =
    veryBusy(next) ∪ { this }
```

Fig. E2. Very Busy Expressions analysis for green-marl (2/2).

Appendix F. Constant propagation analysis specification for green-marl

The full rules for constant propagation in green-marl. This showcases how much the analysis is really an abstract interpreter that is indeed just a lifted concrete interpreter.

```
properties
  constProp: CP

property rules

  constProp(prev → Assign(n, e)) =
    match constProp(prev) with
      | M1(m, v) ⇒ M(m ∪ {Var{n} ↦ v})
      | _ ⇒ CP.top

  constProp(prev → VarRef(n)) =
    addResult(
      constProp(prev),
      getMap(constProp(prev))[Var{n}])

  constProp(prev → Abs(_)) =
    match constProp(prev) with
     | M1(m, v) ⇒ M1(m, constAbs(v))
     | _ ⇒ CP.top

  constProp(prev → UMin(_)) =
    match constProp(prev) with
     | M1(m, v) ⇒ M1(m, constUMin(v))
     | _ ⇒ CP.top

  constProp(prev → Mul(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constMul(l,r))
     | _ ⇒ CP.top

  constProp(prev → Div(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constDiv(l,r))
     | _ ⇒ CP.top

  constProp(prev → Mod(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constMod(l,r))
     | _ ⇒ CP.top

  constProp(prev → Add(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constAdd(l,r))
     | _ ⇒ CP.top

  constProp(prev → Sub(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constSub(l,r))
     | _ ⇒ CP.top

  constProp(prev → Not(_)) =
    match constProp(prev) with
     | M1(m, v) ⇒ M1(m, constNot(v))
     | _ ⇒ CP.top
```

Fig. F1. Constant propagation property rules for green-marl (1/2).

```
  constProp(prev → Or(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constOr(l,r))
     | _ ⇒ CP.top

  constProp(prev → Eq(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constEq(l,r))
     | _ ⇒ CP.top

  constProp(prev → Gt(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constGt(l,r))
     | _ ⇒ CP.top

  constProp(prev → Lt(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constLt(l,r))
     | _ ⇒ CP.top

  constProp(prev → Geq(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constGeq(l,r))
     | _ ⇒ CP.top

  constProp(prev → Leq(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constLeq(l,r))
     | _ ⇒ CP.top

  constProp(prev → Neq(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, constNeq(l,r))
     | _ ⇒ CP.top

  constProp(prev → Cast(_,_)) =
    match constProp(prev) with
       // not modelling casts for now
     | M1(m, v) ⇒ M1(m, Const.top)
     | _ ⇒ CP.top

  constProp(prev → FuncCall(_,_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, Const.top)
     | _ ⇒ CP.top

  constProp(prev → ProcCall(_,_,_)) =
    match constProp(prev) with
     | M2(m, l, r) ⇒ M1(m, Const.top)
     | _ ⇒ CP.top

  constProp(prev → _) = constProp(prev)
```

Fig. F2. Constant propagation property rules for green-marl (2/2).

```
types
  CPType =
  | M(Map(name, Const))
  | M1(Map(name, Const), ConstProp)
  | M2(Map(name, Const), ConstProp, ConstProp)

  ConstProp =
  | Top()
  | Int(int)
  | String(string)
  | Float(float)
  | Bool(bool)
  | Bottom()
```

Fig. F3. Type definitions for constant propagation in green-marl.

```
functions
  getMap(cpt: CPType) = match cpt with
  | M(m) ⇒ m
  | M1(m,_) ⇒ m
  | M2(m,_,_) ⇒ m

  addResult(cpt: CPType, v: Const) =
    match cpt with
    | M1(m, v1) ⇒ M2(m, v1, v)
    | _ ⇒ M1(getMap(cpt), v)

  constAbs(v: Const) = match v with
    | Int(i) ⇒
      if i < 0
        then Int(-i)
        else Int(i)
    | Float(i) ⇒
      if i < 0
        then Float(-i)
        else Float(i)
    | _ ⇒ Const.top

  constUMin(v: Const) = match v with
    | Int(i) ⇒ Int(-i)
    | Float(i) ⇒ Float(-i)
    | _ ⇒ Const.top

  constMul(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Int(i*j)
    | (Float(i), Float(j)) ⇒ Int(i*j)
    | _ ⇒ Const.top

  constDiv(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Int(i/j)
    | (Float(i), Float(j)) ⇒ Int(i/j)
    | _ ⇒ Const.top

  constMod(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Int(i%j)
    | (Float(i), Float(j)) ⇒ Int(i%j)
    | _ ⇒ Const.top

  constAdd(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Int(i+j)
    | (Float(i), Float(j)) ⇒ Int(i+j)
    | _ ⇒ Const.top

  constSub(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Int(i-j)
    | (Float(i), Float(j)) ⇒ Int(i-j)
    | _ ⇒ Const.top

  constNot(v: Const) = match v with
    | Bool(b) ⇒ Bool(!b)
    | _ ⇒ Const.top

  constOr(l: Const, r: Const) = match (l,r) with
    | (Bool(i), Bool(j)) ⇒ Bool(i||j)
    | _ ⇒ Const.top
```

Fig. F4. Constant propagation functions for GREEN-MARL (1/2).

```
  constEq(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i==j)
    | (Float(i), Float(j)) ⇒ Bool(i==j)
    | (String(i), String(j)) ⇒ Bool(i==j)
    | (Bool(i), Bool(j)) ⇒ Bool(i==j)
    | _ ⇒ Const.top

  constGt(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i>j)
    | (Float(i), Float(j)) ⇒ Bool(i>j)
    | _ ⇒ Const.top

  constLt(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i<j)
    | (Float(i), Float(j)) ⇒ Bool(i<j)
    | _ ⇒ Const.top

  constGeq(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i>=j)
    | (Float(i), Float(j)) ⇒ Bool(i>=j)
    | _ ⇒ Const.top

  constLeq(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i<=j)
    | (Float(i), Float(j)) ⇒ Bool(i<=j)
    | _ ⇒ Const.top

  constNeq(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) ⇒ Bool(i!=j)
    | (Float(i), Float(j)) ⇒ Bool(i!=j)
    | _ ⇒ Const.top
```

Fig. F5. Constant propagation functions for GREEN-MARL (2/2).

```
lattices
  CP where
    type = CPType

    lub(l, r) = match (l,r) with
      | (M(l), M(r)) ⇒ M(Map.lub(l,r))
      | (M1(l, cl), M1(r, cr)) ⇒
          M1(Map.lub(l,r), Const.lub(cl,cr))
      | (M2(l, cl1, cl2), M2(r, cr1, cr2)) ⇒
          M2(
            Map.lub(l,r),
            Const.lub(cl1, cr1),
            Const.lub(cl2, cr2))
      | _ ⇒ CP.top

    bottom = M(Map.bottom)

    top = M(Map.top)

  Const where
    type = ConstProp

    lub(l, r) = match (l,r) with
      | (Top(), _) ⇒ Top()
      | (_, Top()) ⇒ Top()
      | (_, Bottom()) ⇒ l
      | (Bottom(), _) ⇒ r
      | (Int(i), Int(j)) ⇒ if i == j
          then Int(i) else Top()
      | (String(i), String(j)) ⇒ if i == j
          then String(i) else Top()
      | (Float(i), Float(j)) ⇒ if i == j
          then Float(i) else Top()
      | (Bool(i), Bool(j)) ⇒ if i == j
          then Bool(i) else Top()
      | _ ⇒ Top()

    bottom = Bottom()
```

Fig. F6. Lattice definitions for constant propagation in GREEN-MARL.

Appendix G. Reaching definitions analysis for STRATEGO

Named `bound-unbound-vars` in the STRATEGO compiler. Used on STRATEGO core but, due to legacy reasons, is defined on a larger subset of STRATEGO. Uses dynamic rewrite rules to encode name binding, mixed with data-flow analysis. Source: https://github.com/metaborg/strategoxt/blob/76689003f94bcd51c84712bf4509b706dd34d9ab/strategoxt/stratego-libraries/strc/lib/stratego/strc/opt/bound-unbound-vars.str.

```
module bound-unbound-vars
imports stratego/strc/lib/stratlib
strategies

  mark-bound-unbound-vars =
    mark-buv

  mark-bound-unbound-vars-old =
    if-verbose4(say(!"marking bound-unbound-vars"))
    ; Specification(
        {| MarkVar
         : at-last(Strategies(map(mark-buv)))
         |}
      )
    ; if-verbose4(say(!"marked bound-unbound-vars"))

  /**
   * Annotate variables with one of the annotations "bound",
   * "unbound", or "(un)bound".
   *
   * Variables are bound in matches, used in builds, and
   * refreshed in scopes. Choice operators may lead to
   * a variable being bound in one path, but not in the
   * other. Such variables are annotated with "(un)bound".
   */

  mark-buv = //debug(!"mark-buv in: "); dr-print-rule-set(|"MarkVar"); (
    mark-match
    <+ mark-build
    <+ mark-scope
    <+ mark-let
    <+ mark-traversal
    <+ mark-sdef
    <+ mark-rdef
    <+ mark-lrule
    <+ mark-srule
    <+ mark-overlay
    <+ mark-call
    <+ mark-prim
    <+ mark-rec
    <+ mark-choice(mark-buv)
    <+ mark-lchoice(mark-buv)
    <+ mark-guardedlchoice(mark-buv)
    <+ all(mark-buv)
  //); debug(!"mark-buv out: "); dr-print-rule-set(|"MarkVar")

strategies

  DeclareUnbound =
    where(!"unbound" ⇒ anno)
    ; ?x
    ; rules(MarkVar+x : Var(x) → Var(x){anno})

  IntroduceBound =
    where(!"bound" ⇒ anno)
    ; ?x
    ; rules(MarkVar+x : Var(x) → Var(x){anno})
```

Fig. G1. Reaching Definitions as implemented in the STRATEGO compiler, written in STRATEGO using the dynamic rules feature. (1/5).

```
  DeclareBound =
    where(!"bound" ⇒ anno)
    ; ?x
    ; rules(MarkVar.x : Var(x) → Var(x){anno})

  DeclareMaybeUnbound =
    where(!"(un)bound" ⇒ anno)
    ; ?x
    ; rules(MarkVar.x : Var(x) → Var(x){anno})

  undefine-unbound-MarkVar =
    where(map(
      where(<mark-var> Var(<id>) ⇒ Var(_){"unbound"})
      ; DeclareMaybeUnbound
    ))

  mark-var =
    bagof-MarkVar; select-mark

  select-mark =
    ?[] < fail + ?[<id>] <+ \ [Var(x) | _] → Var(x){"(un)bound"} \

  fork-MarkVar(s1, s2) =
    s1 \MarkVar/ s2

strategies

  mark-scope =
    Scope(?xs, {| MarkVar : where(!xs; map(DeclareUnbound)); mark-buv|})

  mark-match =
    Match(mark-match-vars)

  mark-match-vars =
      Var(id)     < MarkAndBind
    + App(id,id)  < App(mark-buv, mark-build-vars)
    + RootApp(id) < RootApp(mark-buv)
    + all(mark-match-vars)

  MarkAndBind =
    try(mark-var)
    ; Var(DeclareBound)

  mark-build =
    Build(mark-build-vars)

  mark-build-vars =
      Var(id)     < mark-var
    + App(id,id)  < App(mark-buv, mark-build-vars)
    + RootApp(id) < RootApp(mark-buv)
    + all(mark-build-vars)

  mark-traversal =
    (?All(_) + ?One(_) + ?Some(_))
    ; fork-MarkVar(id, one(mark-buv))
```

Fig. G2. Reaching Definitions as implemented in the STRATEGO compiler, written in STRATEGO using the dynamic rules feature. (2/5).

```
mark-call =
  Call(id,id)
  //; debug(!"call a: ")
  ///; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, Call(id, mark-buv))
  //; debug(!"call b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-call =
  CallT(id,id,id)
  //; debug(!"callt a: ")
  //; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, CallT(id, id, map(mark-build-vars)); CallT(id, mark-buv , id))
  //; debug(!"callt b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-call =
  CallDynamic(id,id,id)
  //; debug(!"callt a: ")
  //; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, CallDynamic(mark-build-vars , id, map(mark-build-vars))
  ; CallDynamic(id, mark-buv , id))
  //; debug(!"callt b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-prim =
  PrimT(id,id,id)
  ; fork-MarkVar(id, PrimT(id, id, map(mark-build-vars)); PrimT(id, mark-buv , id))

mark-let =
  Let(id, id)
  //; debug(!"let a: ")
  //; dr-print-rule-set(|"MarkVar")
  ; where(?Let(<tvars>,_); undefine-unbound-MarkVar)
  ; Let(map(fork-MarkVar(
                          mark-buv//; debug(!"let right: "); dr-print-rule-set(|"MarkVar")
                        , id//; debug(!"let left:"); dr-print-rule-set(|"MarkVar")
                         ))
        //; debug(!"let b: ")
        //; dr-print-rule-set(|"MarkVar")
       ,    fork-MarkVar(id, mark-buv))
  //; debug(!"let c: ")
  //; dr-print-rule-set(|"MarkVar")

mark-sdef :
  SDefT(f, as1, as2, s) → SDefT(f, as1, as2, s’)
  where <map(?VarDec(<id>,_) + ?DefaultVarDec(<id>))> as2 ⇒ as2’
      ; {| MarkVar :
           <map(IntroduceBound)> as2’
           ; <mark-buv> s ⇒ s’
        |}
```

Fig. G3. Reaching Definitions as implemented in the STRATEGO compiler, written in STRATEGO using the dynamic rules feature. (3/5).

```
mark-rdef :
  RDefT(f, as1, as2, r@Rule(t1, t2, s)) →
  RDefT(f, as1, as2, Rule(t1', t2', s'))
  where <map(?VarDec(<id>,_) + ?DefaultVarDec(<id>))> as2 ⇒ as2'
      ; <diff>(<tvars> r, as2') ⇒ xs
      ; {| MarkVar :
           <map(IntroduceBound)> as2'
           ; <map(DeclareUnbound)> xs
           ; <mark-match-vars> t1 ⇒ t1'
           ; <mark-buv> s ⇒ s'
           ; <mark-build-vars> t2 ⇒ t2'
        |}

mark-rdef :
  RDef(f, as1, r@Rule(t1, t2, s)) →
  RDef(f, as1, Rule(t1', t2', s'))
  where <tvars> r ⇒ xs
      ; {| MarkVar :
           <map(DeclareUnbound)> xs
           ; <mark-match-vars> t1 ⇒ t1'
           ; <mark-buv> s ⇒ s'
           ; <mark-build-vars> t2 ⇒ t2'
        |}

mark-lrule :
  LRule(Rule(t1, t2, s)) → LRule(Rule(t1', t2', s'))
  where {| MarkVar :
           <tvars> t1; map(DeclareUnbound)
           ; <mark-match-vars> t1 ⇒ t1'
           ; <mark-buv> s ⇒ s'
           ; <mark-build-vars> t2 ⇒ t2'
        |}

mark-srule :
  SRule(Rule(t1, t2, s)) → SRule(Rule(t1', t2', s'))
  where {| MarkVar :
           // <tvars> t1; map(DeclareUnbound)
           <mark-match-vars> t1 ⇒ t1'
           ; <mark-buv> s ⇒ s'
           ; <mark-build-vars> t2 ⇒ t2'
        |}

mark-overlay :
  Overlay(f, xs, t) → Overlay(f, xs, t')
  where {| MarkVar :
           <map(IntroduceBound)> xs
           ; <mark-build-vars> t ⇒ t'
        |}

mark-rec =
  ?Rec(_, _)
  ; fork-MarkVar(id, Rec(id, mark-buv))

mark-choice(uv) =
  Choice(id, id)
  ; fork-MarkVar(Choice(uv,id), Choice(id, uv))
```

Fig. G4. Reaching Definitions as implemented in the STRATEGO compiler, written in STRATEGO using the dynamic rules feature. (4/5).

```
  mark-lchoice(uv) =
    LChoice(id, id)
    ; fork-MarkVar(LChoice(uv,id), LChoice(id, uv))

  mark-guardedlchoice(uv) =
    GuardedLChoice(id, id, id)
    ; fork-MarkVar(
        GuardedLChoice(uv,id,id); GuardedLChoice(id,uv,id),
        GuardedLChoice(id,id,uv)
      )



/*
    Bound/Unbound variable analysis for Stratego programs

    Copyright (C) 2003 Eelco Visser <visser@acm.org>

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA

*/
```

Fig. G5. Reaching Definitions as implemented in the STRATEGO compiler, written in STRATEGO using the dynamic rules feature. (5/5).

References

[1] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis (2. corr. print), Springer, 2005. http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65410-0

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, third edition, Prentice Hall PTR, Boston, Mass., 2005.

[3] The Rust Project Developers, librustc – builtin lints, 2018. Accessed on 2018-04-10.

[4] M.A. Auslander, M. Hopkins, An overview of the pl.8 compiler, Proceedings of the SIGPLAN Symposium on Compiler Construction, (1982), pp. 22–31, https://doi.org/10.1145/800230.806977.

[5] The checkstyle team, checkstyle – coding, 2018. Accessed on 2018-04-10.

[6] S. Hong, M. Sevenich, J. Lugt, gm_comp, a compiler for green-marl written in c + +, 2014. Accessed on 2018-04-10.

[7] L.C.L. Kats, E. Visser, The Spoofax language workbench: rules for declarative specification of languages and IDEs, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, ACM, Reno/Tahoe, Nevada, 2010, pp. 444–463, https://doi.org/10.1145/1869459.1869497.

[8] T. Vollebregt, L.C.L. Kats, E. Visser, Declarative specification of template-based textual editors, in: A. Sloane, S. Andova (Eds.), Proceedings of the International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, 2012, pp. 1–7, https://doi.org/10.1145/2427048.2427056. Tallinn, Estonia, March 31, - April 1, 2012

[9] H. van Antwerpen, P. Néron, A.P. Tolmach, E. Visser, G. Wachsmuth, A constraint language for static semantic analysis based on scope graphs, in: M. Erwig, T. Rompf (Eds.), Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM, ACM, 2016, pp. 49–60, https://doi.org/10.1145/2847538.2847543. St. Petersburg, FL, USA, January 20, - 22, 2016

[10] J. Smits, E. Visser, FlowSpec: declarative dataflow analysis specification, in: B. Combemale, M. Mernik, B. Rumpe (Eds.), Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE, ACM, 2017, pp. 221–231, https://doi.org/10.1145/3136014.3136029. Vancouver, BC, Canada, October 23–24, 2017

[11] D.J. Pearce, J. Noble, Structural and Flow-sensitive types for Whiley, Technical Report, School of Engineering and Computer Science, Victoria University of Wellington, 2011.

[12] Jetbrains, Type checks and casts: 'is' and 'as' - kotlin programming language, 2018. Accessed on 2018-04-12.

[13] Red Hat, Inc., Eclipse ceylon: Quick introduction, 2018. Accessed on 2018-04-12.

[14] O. Shivers, Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned (with retrospective), in: K.S. McKinley (Ed.), Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 1988, pp. 257–269, https://doi.org/10.1145/989393.989421. 1979–1999, A Selection

[15] Y. Smaragdakis, M. Bravenboer, O. Lhoták, Pick your contexts well: understanding object-sensitivity, in: T. Ball, M. Sagiv (Eds.), Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, ACM, 2011, pp. 17–30, https://doi.org/10.1145/1926385.1926390. 2011, Austin, TX, USA, January 26–28, 2011

[16] J.B. Kam, J.D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (1977) 305–317.

[17] L.C.L. Kats, K.T. Kalleberg, E. Visser, Domain-specific languages for composable editor plugins, Electron Notes Theor. Comput. Sci. 253 (7) (2010) 149–163, https://doi.org/10.1016/j.entcs.2010.08.038.

[18] P. Néron, A.P. Tolmach, E. Visser, G. Wachsmuth, A theory of name resolution, in: J. Vitek (Ed.), roceedings of the 24th European Symposium on Programming Languages and Systems, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, Lecture Notes in Computer Science, 9032 Springer, 2015, pp. 205–231, https://doi.org/10.1007/978-3-662-46669-8_9. ETAPS 2015, London, UK, April 11–18, 2015. Proceedings

[19] C. Wimmer, T. Würthinger, Truffle: a self-optimizing runtime system, in: G.T. Leavens (Ed.), Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity, ACM, 2012, pp. 13–14, https://doi.org/10.1145/2384716.2384723. SPLASH '12, Tucson, AZ, USA, October 21–25, 2012

[20] S. Horwitz, A.J. Demers, T. Teitelbaum, An efficient general iterative algorithm for dataflow analysis, Acta Informatica 24 (6) (1987) 679–694.

[21] M. Jourdan, D. Parigot, Techniques for improving grammar flow analysis, in: N.D. Jones (Ed.), Proceedings of the 3rd European Symposium on Programming, Copenhagen, Denmark, Lecture Notes in Computer Science, 432 Springer, 1990, pp. 240–255. May 15–18, 1990, Proceedings

[22] J.B. Kam, J.D. Ullman, Global data flow analysis and iterative algorithms, J. ACM 23 (1) (1976) 158–171, https://doi.org/10.1145/321921.321938.

[23] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.

[24] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-Marl: a DSL for easy and efficient graph analysis, in: T. Harris, M.L. Scott (Eds.), Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, 2012, pp. 349–362, https://doi.org/10.1145/2150976.2151013. London, UK, March 3–7, 2012

[25] Oracle Corporation, PGX 1.1.0 Documentation – Closeness Centrality, 2015. Accessed on 2018-02-27.

[26] A. Bavelas, Communication patterns in task-oriented groups, J. Acoust. Soc. Am.

(1950).
[27] S.R. Kosaraju, Strong-connectivity algorithm, 1978.
[28] F. Allen, J. Cocke, A catalogue of optimizing transformations, in: R. Rustin (Ed.), Design and Optimization of Compilers, 1st, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 1–30.
[29] Oracle Corporation, PGX 1.1.0 Documentation – List of Built-in Algorithms, 2015. Accessed on 2018-02-27.
[30] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. a language and toolset for program transformation, Sci. Comput. Program. 72 (1–2) (2008) 52–70, https://doi.org/10.1016/j.scico.2007.11.003.
[31] M. Bravenboer, A. van Dam, K. Olmos, E. Visser, Program transformation with scoped dynamic rewrite rules, Fundam. Inform. 69 (1–2) (2006) 123–178. https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06
[32] G.A. Kildall, A unified approach to global program optimization, POPL, (1973), pp. 194–206.
[33] T. Ekman, G. Hedin, The jastadd system - modular extensible compiler construction, Sci. Comput. Program. 69 (1–3) (2007) 14–26, https://doi.org/10.1016/j.scico.2007.02.003.
[34] D.E. Knuth, Semantics of context-free languages, Theory Comput. Syst. 2 (2) (1968) 127–145. http://www.springerlink.com/content/m2501m07m4666813/
[35] E. Söderberg, T. Ekman, G. Hedin, E. Magnusson, Extensible intraprocedural flow analysis at the abstract syntax tree level, Sci. Comput. Program. 78 (10) (2013) 1809–1827, https://doi.org/10.1016/j.scico.2012.02.002.
[36] G. Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000).
[37] H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher-order attribute grammars, Proceedings of the PLDI, (1989), pp. 131–145.
[38] E. Magnusson, G. Hedin, Circular reference attributed grammars - their evaluation and applications, Sci. Comput. Program. 68 (1) (2007) 21–37, https://doi.org/10.1016/j.scico.2005.06.005.
[39] E. Magnusson, T. Ekman, G. Hedin, Extending attribute grammars with collection attributes–evaluation and applications, Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation, (2007), https://doi.org/10.1109/SCAM.2007.13.
[40] E.V. Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Sci. Comput. Program. 75 (1–2) (2010) 39–54, https://doi.org/10.1016/j.scico.2009.07.004.
[41] L.C.L. Kats, A.M. Sloane, E. Visser, Decorated attribute grammars: Attribute evaluation meets strategic programming, in: O. de Moor, M.I. Schwartzbach (Eds.), Proceedings of the 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, Lecture Notes in Computer Science, 5501 Springer, 2009, pp. 142–157, https://doi.org/10.1007/978-3-642-00722-4_11. York, UK, March 22–29, 2009. Proceedings
[42] A.M. Sloane, M. Roberts, L.G.C. Hamey, Respect your parents: How attribution and rewriting can get along, in: B. Combemale, D.J. Pearce, O. Barais, J.J. Vinju (Eds.), Proceedings of the 7th International Conference Software Language Engineering, Lecture Notes in Computer Science, 8706 Springer, 2014, pp. 191–210 , https://doi.org/10.1007/978-3-319-11245-9_11. Västeras, Sweden, September 15–16, 2014. Proceedings
[43] M. Bravenboer, Y. Smaragdakis, Strictly declarative specification of sophisticated points-to analyses, in: S. Arora, G.T. Leavens (Eds.), Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, ACM, 2009, pp. 243–262, https://doi.org/10.1145/1640089.1640108.
[44] Y. Smaragdakis, G. Balatsouras, Pointer analysis, Found. Trends Program. Lang. 2 (1) (2015) 1–69, https://doi.org/10.1561/2500000014.
[45] M. Madsen, M.-H. Yee, O. Lhoták, From datalog to flix: a declarative language for fixed points on lattices, in: C. Krintz, E. Berger (Eds.), Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, ACM, 2016, pp. 194–208, https://doi.org/10.1145/2908080.2908096. Santa Barbara, CA, USA, June 13–17, 2016
[46] T. Szabó, S. Alperovich, M. Völter, S. Erdweg, An extensible framework for variable-precision data-flow analyses in mps, in: D. Lo, S. Apel, S. Khurshid (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE, ACM, 2016, pp. 870–875, https://doi.org/10.1145/2970276.2970296. Singapore, September 3–7, 2016
[47] T. Szabó, S. Erdweg, M. Völter, Inca: a dsl for the definition of incremental program analyses, in: D. Lo, S. Apel, S. Khurshid (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE, ACM, 2016, pp. 320–331, https://doi.org/10.1145/2970276.2970298. Singapore, September 3–7, 2016
[48] T. Szabó, M. Völter, S. Erdweg, Inca$_l$: A dsl for incremental program analysis with lattices, Proceedings of the International Workshop on Incremental Computing (IC), (2017). Talk proposal; full article not published yet
[49] P. Klint, T. van der Storm, J.J. Vinju, EASY meta-programming with Rascal, in: J.M. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE, Lecture Notes in Computer Science, 6491 Springer, 2009, pp. 222–289, https://doi.org/10.1007/978-3-642-18023-1_6. Braga, Portugal, July 6–11, 2009. Revised Papers
[50] M. Hills, Streamlining control flow graph construction with dcflow, in: B. Combemale, D.J. Pearce, O. Barais, J.J. Vinju (Eds.), 7th International Conference on Software Language Engineering, SLE, Lecture Notes in Computer Science, 8706 Springer, 2014, pp. 322–341 , https://doi.org/10.1007/978-3-319-11245-9_18. Västeras, Sweden, September 15–16, 2014. Proceedings
[51] P. Ginsbach, L. Crawford, M.F.P. O'Boyle, Candl: a domain specific language for compiler analysis, in: C. Dubach, J. Xue (Eds.), Proceedings of the 27th International Conference on Compiler Construction, ACM, 2018, pp. 151–162, https://doi.org/10.1145/3178372.3179515. February 24–25, 2018, Vienna, Austria
[52] G. Konat, L.C.L. Kats, G. Wachsmuth, E. Visser, Declarative name binding and scope rules, in: K. Czarnecki, G. Hedin (Eds.), Proceedings of the 5th International Conference on Software Language Engineering, SLE, Lecture Notes in Computer Science, 7745 Springer, 2012, pp. 311–331, https://doi.org/10.1007/978-3-642-36089-3_18. Dresden, Germany, September 26–28, 2012, Revised Selected Papers