TU Delft

Delft University of Technology

FlowSpec: Declarative Dataflow Analysis Specification

Smits, Jeff; Visser, Eelco

DOI
10.1145/3136014.3136029

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering

Citation (APA)
Smits, J., & Visser, E. (2017). 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. (pp. 221-231). Vancouver, BC, Canada: Association for Computing Machinery (ACM). DOI: 10.1145/3136014.3136029

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

FlowSpec: Declarative Dataflow Analysis Specification

Jeff Smits
TU Delft
The Netherlands
j.smits-1@tudelft.nl

Eelco Visser
TU Delft
The Netherlands
e.visser@tudelft.nl

Abstract

We present FlowSpec, a declarative specification language for the domain of dataflow analysis. FlowSpec has declarative support for the specification of control flow graphs of programming languages, and dataflow analyses on these control flow graphs. We define the formal semantics of FlowSpec, which is rooted in Monotone Frameworks. We also discuss implementation techniques for the language, partly used in the prototype implementation built in the Spoofax Language Workbench. Finally, we evaluate the expressiveness and conciseness of the language with two case studies. These case studies are analyses for Green-Marl, an industrial, domain-specific language for graph processing. The first case study is a classical dataflow analysis, scaled to this full language. The second case study is a domain-specific analysis of Green-Marl.

CCS Concepts • **Software and its engineering → Domain specific languages**;

Keywords control flow graph, dataflow analysis

ACM Reference Format:
Jeff Smits and Eelco Visser. 2017. FlowSpec: Declarative Dataflow Analysis Specification. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3136014.3136029

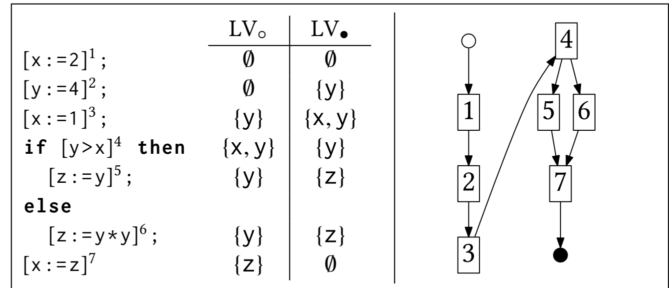


Figure 1. Classical dataflow analysis Live Variables (LV). On the left is an example program in the While language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the centre is the analysis result. The $LV_\circ$ and $LV_\bullet$ are before and after the CFG node's variables accesses respectively.

1 Introduction

Dataflow analysis is a static analysis that answers questions on what *may* or *must* happen before or after a certain point in a program's execution. With dataflow analysis we can answer whether a value written to a variable *here* may be read *later*. Such dataflow analyses can be used to inform optimisations.

For example, consider Live Variables analysis, illustrated in Figure 1. This type of dataflow analysis can identify dead code, which can be removed as an optimisation. In the example this would be statement 1 since it writes x which is overwritten by statement 3 without being read in between. The Live Variables analysis provides a set of variables which are read before being written after each statement in $LV_\bullet$. The figure shows this in the $LV_\bullet$ set of statement 1, which does not contain x.

Dataflow may also be part of a language's static semantics. For example, in Java a final field in a class must be initialised by the end of construction of an object of that class. Since constructor code can have conditional control flow, a dataflow analysis is necessary to check that all possible execution paths through constructors actually assign a value to the final field [Gosling et al. 2005, sect. 16.9].

Dataflow analyses are often operationally encoded, whether in a general purpose language, an attribute grammar system or a logic programming language. This encoding is both an overhead for the engineer implementing it, as well as an overhead in decoding for anyone who wishes to understand the analysis.

In formal, mathematical descriptions of a dataflow analysis, the common patterns are often factored out. This shows commonalities between different analyses, allows the study of those commonalities and differences, as well as general

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SLE'17, October 23–24, 2017, Vancouver, Canada

© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5525-4/17/10...$15.00
https://doi.org/10.1145/3136014.3136029

proofs about common pieces. But this abstraction again gives an overhead for understanding any particular analysis, as it first requires the understanding of the common framework in full generality.

Our goal is to provide a concise, readable, declarative specification language for dataflow analysis that is more intuitive than traditional encodings and decomposed descriptions. Our domain-specific meta-language, FlowSpec, can provide a unified view of a dataflow analysis, as opposed to an abstract formal description, while focussing on *What*, instead of *How*.

This paper makes the following contributions:

- We present FlowSpec, a new declarative programming language for Dataflow Analysis, introduced in Section 2.
- We define the dynamic semantics of the language, by using Monotone Frameworks (see Section 2) as a semantic model. The semantics can be found in Section 3.
- We show the expressivity of the language in multiple examples and describe two larger case studies in Section 5.

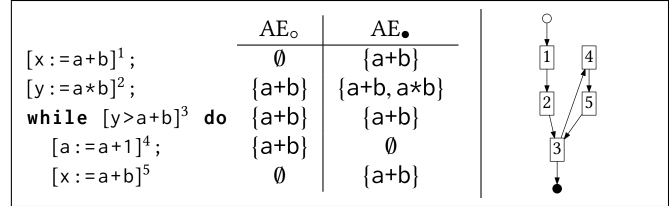
In Section 4 we discuss our prototype implementation of FlowSpec in the Spoofax language workbench, in Section 6 we discuss related work and Section 7 concludes the paper.

2 FlowSpec and Monotone Frameworks

Our main inspiration for FlowSpec is Monotone Frameworks [Kam and Ullman 1977], a formal method for describing dataflow analyses. We assume the reader is somewhat familiar with this work and give a summary here. Throughout this paper we use the notation from [Nielson et al. 2005], which uses the dual notation of the original publication (e.g. $\sqcup$ instead of $\wedge$).

In short, Monotone Frameworks are a general lattice theoretic framework for the description of dataflow analyses. It captures the commonalities of intra-procedural, flow-sensitive dataflow 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 dataflow analysis, but also a simple worklist algorithm that can perform the analysis.

We define the semantics of FlowSpec in Section 3, by mapping our DSL constructs onto Monotone Frameworks. In this section we will introduce the language by example, as well as refer to the corresponding Monotone Frameworks terminology. We use two classical, set-based analyses as examples of this section. Live Variables analysis was already introduced in Figure 1. The second example is Available Expressions, which can be found in Figure 2. Available Expressions is another classical dataflow analysis, that identifies expressions that have been calculated before. These expressions are *available* while variables used in the expressions are not



	$AE_\circ$	$AE_\bullet$
`[x:=a+b]`^{1};	$\emptyset$	{a+b}
`[y:=a*b]`^{2};	{a+b}	{a+b, a*b}
while `[y>a+b]`^{3} **do**	{a+b}	{a+b}
`[a:=a+1]`^{4};	{a+b}	$\emptyset$
`[x:=a+b]`^{5}	$\emptyset$	{a+b}

Figure 2. Classical dataflow analysis Available Expressions (AE). 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 centre is the analysis result. The open and closed dots on the analysis abbreviation are before and after a CFG node's effect respectively.

$S ::= n$ `:=` $a \mid S_1; S_2$	Stmt
$\mid$ **if** b **then** S_1 **else** $S_2 \mid$ **while** b **do** S	
$a ::= x \mid n \mid a_1\ op_a\ a_2$	AExp
$b ::=$ **true** $\mid$ **false** $\mid$ **not** $b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2$	BExp
$B ::= n$ `:=` $a \mid b$	Labeled blocks

Figure 3. Grammar of the While language used in the examples. Literals are on a grey background. $op_.$ are arithmetic, boolean, and relational operations. Boolean expressions are only labeled when they are conditions of a statement.

```
cfg Root(s) = entry → cfg s → exit

cfg a@Assign(_, _) = entry → a → exit

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

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

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

Figure 4. Control flow graph rules for the While language. Each rule can have one or more chains of edges, where the virtual entry and exit nodes point out the actual start and end of the local control flow given. Note how the rule for **if** has two chains from the condition to the different branches, and the rule for **while** uses the condition twice to make a loop.

written to. The analysis result can be used for an optimisation that eliminates common subexpressions. In the example, the while condition (labelled 3) uses a+b, but this expression is available already according $AE_\circ$. The grammar of the example language can be found in Figure 3.

We will now discuss the features of FlowSpec and the related components that Monotone Frameworks require.

Control flow graphs Since the dataflow analysis is flow-sensitive, the first thing we need is a control flow graph. We build this control flow graph as edges between relevant

parts of the program. In FlowSpec, the **cfg** function can be defined to provide a control flow graph. In Figure 4 we show the specification of **cfg** for the While language used in the examples. The function is defined case-by-case on nodes of the Abstract Syntax Tree (AST) and maps these nodes to local control flow. In the body of the function, virtual entry and exit nodes are accessible, to mark entry and exit of the local control flow. Recursive calls can be used to connect control flow of sub-trees. Matched AST nodes can be used directly in the control flow graph to make them control flow graph nodes. Using the name bound to an AST node twice in the same **cfg** rule reuses the same control flow graph node. Structurally equal AST nodes from different parts of the program are considered different control flow graph nodes.

In Monotone Frameworks program fragments are labelled to make this distinction of control flow nodes clear. A control flow graph F is seen as a set of edges (a subset of $\mathbf{Lab} \times \mathbf{Lab}$) between different labels.

The direction that the control flow graph goes into is found in the direction of the edges. Flipping all edges of the graph gives the reverse control flow graph. In Monotone Frameworks one component that must be consistent with the forward or reverse control flow graph is the set of 'extremal labels' $E \in \mathcal{P}(\mathbf{Lab})$. These are normally the global entry nodes for the forward control flow graph, and the global exit nodes for the reverse control flow graph, but the framework allows for a different set if desired. In FlowSpec we leave these nodes implicit and always use the relevant extremal nodes of the control flow graph, depending on forward or reverse usage.

Dataflow type and transfer functions The result of a dataflow analysis is called a dataflow property in FlowSpec. 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, or transfer function, on this data. One can see this as an abstracted version of the effect that the program fragment would have during program execution or a trace thereof.

In Figure 5 we show this transfer function for Live Variables analysis. The specification at the top uses a single rule that matches any AST node `t` with a control flow edge to `s`. This rule says that `Live t` is defined in terms of `Live s`, with the kill set subtracted and the gen set added. These kill and gen sets mark the analysis as a classical dataflow analysis. FlowSpec's unified rule defines both the direction of the dataflow analysis, in this case backward, and defines the dataflow property in terms of itself elsewhere in the graph. But we still have these kill and gen functions that are factored out. In the bottom of the figure we show the idiomatic FlowSpec specification of Live Variables, which inlines the kill and gen functions. In this case we still use something similar to gen; `Reads` is a property that just contains the read variables in an expression. Note that a more specific

```
Live t → s = Live s - kill t + gen t

fun kill (t: term): Set string = match t with
| Assign(n, _) ⇒ {n}
| _ ⇒ {}

fun gen (t: term): Set string = match t with
| Assign(_, e) ⇒ gen e
| Ref(n) ⇒ {n}
| Gt(e1, e2) ⇒ gen e1 + gen e2
| Lt(e1, e2) ⇒ gen e1 + gen e2
| Eq(e1, e2) ⇒ gen e1 + gen e2
| Not(e) ⇒ gen e
| Plus(e1, e2) ⇒ gen e1 + gen e2
| Mul(e1, e2) ⇒ gen e1 + gen e2
// Etc.
```

```
Live Assign(n, e) → s = Live s - {n} + Reads e

Live t → s = Live s + Reads t
```

Figure 5. Live Variables specification in FlowSpec. The top has the classical specification in terms of gen and kill sets, whereas the bottom is the idiomatic specification for FlowSpec. `Reads` is defined as a property to avoid recomputation, but otherwise looks similar to gen.

```
Live _ → exit = {"EAX"}
```

Figure 6. Example of extremal value $\{EAX\}$ in FlowSpec for Live Variables

name already helps to show the intention of the analysis, and makes it more reusable. We now have two rules: one for the assignment, and one default rule for everything else.

The type of the dataflow property is L in our Monotone Frameworks presentation. The transfer functions $f_\ell : L \rightarrow L$ are connected to the labels ℓ of the control flow graph nodes. Monotone Frameworks also take a component ι, the extremal value. This is the value with which the extremal labels are initialised. For example, in Live Variables analysis we assumed all variables to be dead at the end of the program, but many other assumptions could be useful. In Figure 6 we show the FlowSpec representation of extremal values, by a rule that uses exit in the control flow graph edge pattern, which in this case stands for the *global* exit.

Lattices and termination The control flow graph can split and merge because of conditional control flow such as an **if** statement. We can propagate data along both edges of a split, but need to do something about the data coming from multiple directions at a merge. In FlowSpec this translates to a control flow edge pattern that can match multiple edges in case of a merge. The solution employed is that the data is merged before the transfer function of the merging node is applied. Monotone frameworks require a complete

lattice instance[1] $(\top, \bot, \sqsubseteq, \sqcup, \sqcap)$ for the type L of the dataflow property, and uses the least-upper bound $\sqcup$ at merge points in the control flow. FlowSpec does the same by requiring a lattice instance for a property that uses control flow edge patterns. In our examples the `MaySet` and `MustSet` are lattice instances that use the `Set` type:

```
prop Live : MaySet string
prop Available : MustSet term (allExprs program)
```

A `MaySet` performs set unions at control flow merge points and compares with non-strict subset inclusion. A `MustSet` uses intersection and non-strict superset comparison. It requires a bottom element to be used in the analysis, where that bottom element is the full set of possible values in the analysis. In this case we use a function that gathers all expressions from the entire input program. The `MaySet` does not need this extra argument because the bottom element of the lattice is the empty set.

In Figure 7 we show the Monotone Frameworks instance of Available Expressions. This analysis is a must analysis, as seen in the top-left where the lattice of L is defined. It is a forward analysis, which uses the normal control flow, and starts at the start of the graph with no available expressions. Since the analysis is classical, we can define the transfer function in terms of gen and kill sets. An assignment to x kills all available expressions that use x. Any expression generates new available expressions.

For comparison, we also provide the FlowSpec specification of Available Expressions in Figure 8. Here instead of filtering all expressions from the program to build a kill set, we directly filter the set of available expressions. Note that we reuse `Reads` here, which was first used in Live Variables.

Apart from split and merges, control flow graph can also have cycles. To obtain a valid, terminating analysis, the transfer functions f_ℓ need to be monotone increasing with respect to the lattice. This allows a monotone framework to calculate cycles to a fixed point. To make sure this calculation terminates, the lattice must adhere to the ascending chain condition. In other words, the lattice must have a finite height.

Together the Monotone Frameworks components can be used to give a formulaic description of the Analysis:

$$\text{Analysis}_\circ(\ell) = \bigsqcup \{\text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell$$

$$\text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

$$\text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell))$$

Again, the open dot is the analysis result before the effect of ℓ, and the closed dot is for after the effect of ℓ.

A trivial fixed point for our formulas is $\forall \ell.\text{Analysis}_\circ(\ell) = \top = \text{Analysis}_\bullet(\ell)$. $\top$ is the value of L that reads as "could be anything", the coarsest approximation available. Although some approximation is necessary to keep the analysis decidable, 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. Of course the accuracy of this fixed point is still dependent on the choice of lattice L and transfer functions f.

In the original work [Kam and Ullman 1977] 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 is *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 [Nielson et al. 2005, sect. 2.4.2].

L	$\mathcal{P}(\mathbf{AExp})$	$f_\ell(l) = (l \setminus kill([B]^\ell)) \cup gen([B]^\ell)$ where $[B]^\ell \in blocks(\mathbf{Prog})$
$\sqsubseteq$	$\supseteq$	
$\bigsqcup$	$\bigcap$	
$\bot$	$\mathbf{AExp}(\mathbf{Prog})$	
ι	$\emptyset$	
E	$\{init(\mathbf{Prog})\}$	
F	$flow(\mathbf{Prog})$	

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

Figure 7. Available Expressions instance for Monotone Frameworks. **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.

```
Available Assign(n, e) ← p = result
where exprs = Available p + SubExprs e
      result = { expr | expr in exprs,
                 !(contains n (Reads expr)) }

Available t ← p = Available p + SubExprs t
```

Figure 8. Available Expressions specification in FlowSpec

Monotone Frameworks notation recap To summarise, 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 dataflow 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.

[1] Technically bounded meet-semilattices are sufficient, which do not require a top element.

M ::= `module` n $(G \mid D \mid R)^*$	Modules
G ::= `cfg` p `=` C (`,` $C)^*$	CFG rules
D ::= `prop` n `=` τ	Property defs
R ::= n P `=` e `where` $(n$ `=` $e)^*$	Property rules
P ::= p `→` E	Match ahead
| p `←` E	Match behind
| p	Match tree
C ::= E `→` E (`→` $E)^*$	Chains
E ::= `entry` | `exit` | `cfg` n | n	Chain elements
n	names
p	patterns
e	expressions
τ	types

Figure 9. The basic grammar of FlowSpec. Literals are on a grey background.

3 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 Figure 9 for a small syntax definition of the language, from which we will use non-terminals to introduce judgements of the semantics.

Control flow graphs The special function **cfg**: **term** → **cfg** is defined case-wise with AST patterns. To model the behaviour of the virtual entry and exit nodes in these rules, we employ a constraint based semantics, given in Figure 10. The smallest set that satisfies these constraints is the control flow graph that the **cfg** function gives. We use $[\![p]\!]^{a^\ell} = \Gamma$ 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 two labels left of the turnstile are the virtual entry and exit labels, which 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 [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 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 rule constraints $\quad \boxed{\ell, \ell \vdash [\![G]\!]^{a^\ell} \supseteq \mathbf{Lab} \times \mathbf{Lab}}$

$$\frac{[\![p]\!]^{a^\ell} = \Gamma \wedge \ell_\circ, \ell_\bullet, \Gamma \vdash C_i \supseteq g_i \wedge 1 \leq i \leq m}{\ell_\circ, \ell_\bullet \vdash [\![\mathbf{cfg}\ p = C_1, \ldots, C_m]\!]^{a^\ell} \supseteq g_i}\ [\text{rule}_i]$$

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

$$\frac{\ell_\circ, \ell, \Gamma \vdash E_1 \supseteq g_1 \quad \wedge\ \ell, \ell_\bullet, \Gamma \vdash E_2 \to \ldots \to E_m \supseteq g_2}{\ell_\circ, \ell_\bullet, \Gamma \vdash E_1 \to E_2 \to \ldots \to E_m \supseteq g_1 \cup g_2}\ [\text{noedge}]$$

$$\frac{\ell_\circ, \ell_1, \Gamma \vdash E_1 \supseteq g_1 \wedge \ell_1 \neq \ell_2 \quad \wedge\ \ell_2, \ell_\bullet, \Gamma \vdash E_2 \to \ldots \to E_m \supseteq g_2 \wedge g_3 = \{(\ell_1, \ell_2)\}}{\ell_\circ, \ell_\bullet, \Gamma \vdash E_1 \to E_2 \to \ldots \to E_m \supseteq g_1 \cup g_2 \cup g_3}\ [\text{edge}]$$

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

$$\frac{}{\ell_\circ, \ell_\circ, \Gamma \vdash \mathbf{entry} \supseteq \emptyset}\ [\text{en}] \qquad \frac{}{\ell_\bullet, \ell_\bullet, \Gamma \vdash \mathbf{exit} \supseteq \emptyset}\ [\text{ex}]$$

$$\frac{\Gamma(n) = a^\ell}{\ell, \ell, \Gamma \vdash n \supseteq \emptyset}\ [\text{lab}]$$

$$\frac{\Gamma(n) = a^\ell \wedge \ell_\circ, \ell_\bullet \vdash [\![\mathbf{cfg}\ p = c_1, \ldots, c_m]\!]^{a^\ell} \supseteq g}{\ell_\circ, \ell_\bullet, \Gamma \vdash \mathbf{cfg}\ n \supseteq g}\ [\text{cfg}]$$

Figure 10. Semantic constraints of the **cfg** function definition in FlowSpec

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

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \wedge \Gamma' \vdash [\![B[n\ n_{adj} := l]]\!] = \Gamma'' \quad \wedge\ \Gamma'' \vdash [\![e[n\ n_{adj} := l]]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ p \leftarrow n_{adj} = e\ \mathbf{where}\ B]\!]^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\text{transfw}]$$

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \wedge \Gamma' \vdash [\![B[n\ n_{adj} := l]]\!] = \Gamma'' \quad \wedge\ \Gamma'' \vdash [\![e[n\ n_{adj} := l]]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ p \rightarrow n_{adj} = e\ \mathbf{where}\ B]\!]^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\text{transbw}]$$

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \wedge \Gamma' \vdash [\![B]\!] = \Gamma'' \quad \wedge\ \Gamma'' \vdash [\![e]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ p = e\ \mathbf{where}\ B]\!]^{a^\ell} \Rightarrow f_\ell^n(l) = e_\lambda}\ [\text{trans}]$$

Figure 11. Mapping of transfer functions in FlowSpec to Monotone Frameworks

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

D ::= ...	definitions
| `type` n `=` $n\ \tau^*$(`|` $n\ \tau^*$)*	Type definitions
| `fun` n (`(` n `:` τ `)`)* `=` e	Function definitions
| `lattice` n (τ | (`(` n `:` τ `)`)* `where` l^*	Lattice definitions
l ::= `type` `=` τ | `lub` $n\ n$ `=` e | `leq` $n\ n$ `=` e	Lattice components
| `bottom` `=` e | `top` `=` e | `glb` $n\ n$ `=` e	

Figure 12. The types and function part of FlowSpec's grammar. Literals are on a grey background.

$\text{Analysis}_\circ(\ell) = \bigsqcup_{(\ell,\ell')\in F} \text{Analysis}_\bullet(\ell')$ for recursive calls instead. This means that we can map our property rules onto mathematical transfer functions, which is what we do in Figure 11. Again, we abstract over pattern matching, and here we also abstract over the execution of the property rules themselves. We consider these rule bodies as polymorphic lambda calculus à la Hindly-Milner. There may be calls to other properties inside one being defined, as long as there are no cyclic dependencies between properties.

We use $\mathcal{F}$ for the transfer function space and B for the bindings in the where clause. 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.

Lattices Users of FlowSpec can define their own algebraic data types and lattice definitions on these types. Of the 5-tuple $(\top, \bot, \sqsubseteq, \sqcup, \sqcap)$, $\sqcap$ and $\top$ 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 Figure 12. 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 Figure 13. Lattices are *required* in the type position of a dataflow property definition, *unless* the dataflow property rules are based purely on AST matching and do not use the control flow graph. In that case the property is similar to a tabulated function.

Built-in data 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` definition do not need to be built in, these can be defined as part of the standard library.

4 Implementation

Our prototype implementation of FlowSpec is implemented in, and integrated with the Spoofax [Kats and Visser 2010] Language Workbench. At the time of writing the prototype is incomplete, therefore we do not claim this as a contribution of this paper. However, we can describe the main component of the implementation, which is the worklist algorithm derived from the one for Monotone Frameworks [Kildall 1973]. In Figure 14 we present the algorithm in pseudo code.

```
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

  leq l r = lub l r == r
```

Figure 13. A constant propagation type and lattice in FlowSpec for Live Variables. Although the $\sqcup$ and $\sqsubseteq$ operations may be derived from each other, we currently require both to be defined.

```
for Prop in topologically ordered Properties:
  if Prop.direction = forward:
    F' := F
  else:
    F' := F.flipEdges()

  for ℓ in F':
    if ℓ in F'.E:
      Prop∘(ℓ) := ι
    else:
      Prop∘(ℓ) := ⊥

  W := E
  while W is not empty:
    (ℓ, ℓ') = W.pop()
    if f_ℓ^Prop(Prop∘(ℓ)) ⋢ Prop∘(ℓ'):
      Prop∘(ℓ') :=
        f_ℓ(Prop∘(ℓ)) ⊔ Prop∘(ℓ')
      for (ℓ', ℓ'') in F':
        W.push(ℓ'')

  for ℓ in F':
    Prop•(ℓ) := f_ℓ^Prop(Prop∘(ℓ))
```

Figure 14. Worklist algorithm used in the implementation

The outer loop uses a topological ordering on all defined properties to be able to use one property in another property's rules. The first inner loop initialises the property analysis. The extremal labels E of the control flow graph F' are initialised with the extremal value ι, everything else with $\bot$.

The **while** loop is the main loop. It pops an edge (ℓ, ℓ') off of worklist W, and uses 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. All edges starting at ℓ' are added to the worklist. 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.

Of course this algorithm is simplified for presentation purposes. We use a topological ordering of the strongly connected components (SCCs) of the control flow graph [Horwitz et al. 1987; Jourdan and Parigot 1990]. Within a strongly connected component the worklist strategy is used to find a fixed point for that part of the graph. The trade-off is the cost of calculating the topologically ordered SCCs, to get the optimal ordering for calculating the property. A number of strategies can be used to solve the fixed point computation with the SCC. We currently use a simple worklist algorithm based on a queue, but are looking into Round Robin algorithms based on the reverse post-order of the depth first spanning forest of the SCC [Kam and Ullman 1976]. There are also optimisation opportunities in the choice of data-structures we use in our implementation, which we intend to investigate.

5 Evaluation

We evaluate the expressiveness and conciseness of FlowSpec. Our two case studies are implementations of dataflow analyses for Green-Marl [Hong et al. 2012], a domain specific language for graph processing. Our first case study is Live Variables analysis for Green-Marl, to answer the question: How well does an analysis in FlowSpec scale, when we go beyond toy languages?

The second case study is a domain specific analysis that is particular to Green-Marl. This analysis, called Read-Write Analysis, is a bottom-up analysis that gathers data access information for later use in data dependence calculations. Here we compare our FlowSpec implementation with the formal specification of the analysis [Smits 2016].

5.1 Live Variables Analysis in Green-Marl

In Section 2 we presented the implementation of Live Variables analysis in FlowSpec for the While language. The While language has 4 different statements, each of which required a separate control flow graph rule, and one more for the root of the AST. This took just 7 lines of code. Green-Marl requires 39 control flow graph rules, one for each AST node up to expressions. The rules span 72 lines of code, excluding comments and empty lines. This is mostly due to code style, where we used two lines of code for even a simple rule. Figure 15 shows a sample of the control-flow graph code.

```
cfg Block(statements) =
  entry → cfg statements → exit

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

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

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

Figure 15. A sample of the control flow graph rules for Green-Marl

```
Reads IntLit(_) = {}
Reads VarRef(n) = {n}
Reads Not(e) = Reads e
Reads Mul(e1, e2) = Reads e1 + Reads e2
```

Figure 16. A sample of the Reads rules for Green-Marl

```
Live VarAssign(n) → s = Live s - n

Live PropAssign(n, p) → s = Live s - p + {n}

Live ElementAssign(n, _) → s = Live s - n

Live this → s = Live s + Reads this
```

Figure 17. A sample of the Live Variables rules for Green-Marl

One problem with expressivity that we noticed while writing these control flow graphs, is that our control flow graph rules do not support intermediate return statements in procedures. This is a non-local jump in control flow, where we need to know the exit label of the procedure that the return statement is in. We are currently working on a solution where the control flow graph can depend on name resolution information, to handle jumps to labels and return statements. Return statements already resolve to their enclosing procedure, which is used to type check return expressions.

Where conciseness is concerned, we observe that some language constructs in Green-Marl are similar in form and function to the point that they have the same control flow graph. This is illustrated in the two rules for InReverse and InPost, which are parts of the breadth-first search and depth-first search constructs respectively. Both consist of an optional filter expression and a statement. We intend to add an option to merge these rules into one, where multiple patterns can be used if they result in the same name bindings. This would result in the elimination of 16 rules, and as many lines of code, when we keep the patterns on separate lines.

Within the Live Variables analysis we use a property Reads, to extract the set of names that are read in an expression. This helper property counts 29 rules, one for every different expression constructor. Each of these rules is short, simple and on a single line (see Figure 16 for an example). Again we might merge these rules when we add the feature of multiple patterns in a single rule. For comparison, the WHILE example had only 8 rules.

The actual Live variables property only needs 4 rules (Figure 17). The first 3 rules match left-hand sides of different assignments in GREEN-MARL and add variables and whole properties to the set of live variables. The 4th rule is the default rule which propagates the live variables from the successor and adds the currently read variables.

5.2 Read-Write Analysis in GREEN-MARL

The Read-Write analysis is formally described in 45 rules [Smits 2016, ch. 4]. Our FLOWSPEC implementation counts 21 rules and 4 functions. Each function handles a number of cases that the formal description has separate rules for, but also handles the cases explicitly that the formal rules leave implicit. For example, in FLOWSPEC we use one rule and one 'function' to describe function calls of GREEN-MARL. The function takes no arguments, it is merely a definition of the set of function names that mutate their arguments. The formal rules handles function calls in 2 rules and the predefined set of function names.

Our case study of this analysis inspired the sketch of an extension of FLOWSPEC that use information on scopes and names to automatically filter names out of the analysis results when these go out of scope. Name abstraction rules can be specified to transform a more complex data-structure that contains a name into one that does not contain that name, abstracting it away. Something similar to this was used in the formal semantics of the analysis, although our FLOWSPEC feature is more general. It simplifies a number of complicated rules, such as the rule for blocks of statements (see Figure 18).

```
prop ReadWriteInfo = Set (name, Mode, Patt)

ReadWriteInfo Block(sts) =
  map (\(n1, m, p) →
    if contains n1 (Declarations sts)
      then None
      else match p with
      | Name n2 ⇒ if contains n2 (Declarations sts)
        then Some (n, m, Random)
        else Some (n, m, p)
      | _ ⇒ Some (n, m, p))
    ReadWriteInfo sts
```

```
prop ReadWriteInfo = Set (name, Mode, Patt)

abstract Patt with
| Name _ ⇒ Random

ReadWriteInfo Block(sts) =
  ReadWriteInfo sts
```

Figure 18. The Read-Write analysis rule for blocks of statements in GREEN-MARL at the top, and the same rule using the experimental 'automatic name filtering' feature at the bottom.

6 Related Work

We will shortly discuss the history of Monotone Frameworks which underlies our work, and some other systems and formalisms for implementing dataflow analysis.

Monotone Frameworks Monotone dataflow analysis frameworks [Kam and Ullman 1977] were first introduced as a generalisation over Killdall's lattice theoretic approach to dataflow analysis [Kildall 1973]. By replacing the distributivity requirement with a monotonicity requirement for the transfer function, Kam and Ullman found a way to describe many 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.

FLOWSPEC builds on Monotone Frameworks approach to provide a unified, domain-specific specification language.

Attribute grammars The JASTADD system [Ekman and Hedin 2007] supports attribute grammars [Knuth 1968] extended with a number of special attributes which allows a declarative intra-procedural control- and dataflow analysis specification [Söderberg et al. 2013]. In particular, these are reference attributes [Hedin 2000] for control-flow graph (CFG) edges, higher-order attributes [Vogt et al. 1989] for virtual CFG nodes, used for entry/exit of methods, circular attributes [Magnusson and Hedin 2007] for fixpoints of dataflow equations, and collection attributes [Magnusson et al. 2007] e.g. for the CFG where there are multiple successors.

In FLOWSPEC we use virtual entry and exit nodes throughout our control flow rules, although these are for ease of specification and are not included in the control flow graph. We support a number of collections similar to those in collection attributes. Our fixed point calculations on lattices are similar to those in the circular attributes. We provide a small functional language for defining more lattices, whereas JASTADD has an escape hatch to Java to expose more data-structures and lattice operations. In comparison, JASTADD is more general purpose and therefore can express more static analyses, whereas FLOWSPEC is more domain-specific to control and dataflow and can express those analyses more directly.

SILVER [Wyk et al. 2010] is another attribute grammar system and specification language that supports similar features to the JASTADD system. For control- and dataflow analysis, there is dedicated 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 dataflow properties in a declarative manner.

The Stratego strategic programming language was extended with attribute grammars in Aster [Kats et al. 2009]. 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.

The Stratego programming language was also directly applied to dataflow analysis by leveraging its dynamic rules [Bravenboer et al. 2006]. 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 sides. 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 dataflow analysis as a separate concern that allows us to treat it in domain-specific terms. By leaving out the transformation concern, we have a simpler, if less powerful language.

Kiama [Sloane et al. 2014] 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 addressed 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.

Relational Programming The Doop framework [Bravenboer and Smaragdakis 2009] uses a Datalog dialect for a declarative specification of static analyses such as context-sensitive pointer analysis. In a recent tutorial, 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 [Smaragdakis and Balatsouras 2015, p. 46].

The Flix programming language [Madsen et al. 2016] 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 dataflow 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.

Meta-programming environments The MPS language workbench[2] has MPS-DF, a special component for definition of dataflow analyses [Szabó et al. 2016a]. MPS-DF has support for building dataflow 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, of inter-procedurally by inlining method calls. To support this variability, two different dataflow graph builders need to be implemented for a procedure call element in the AST.

Another MPS related language is IncA [Szabó et al. 2016b], 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 lends itself well for certain analyses that can be modelled well with relations. Its limitations are around generating data at runtime that is not directly connected to the program, such as building intervals in an interval analysis.

In contrast FlowSpec has no problem with runtime generated data, but lacks the incremental analysis that makes IncA so scalable.

Rascal [Klint et al. 2009] provides a facility for control flow graph construction with DCFlow [Hills 2014], 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 dataflow algorithms on top of a CFG is still done in the Rascal language, without a special library or framework for the use-case.

7 Conclusion

We have presented FlowSpec, a declarative specification language for the domain of dataflow analysis. FlowSpec uses Monotone Frameworks as a semantics model, and we have presented its semantics as a mapping onto Monotone Frameworks. We have demonstrated a number of example specifications in FlowSpec and reported on two case studies of larger specifications. We also briefly discussed the prototype implementation and a number of details we are still exploring there.

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

Limitations and Future Work Currently we describe control flow as a purely local function that can be solved before the start of dataflow analysis. To allow breaks from loops and jumps to labels we would like to extend the **cfg** function, so it may use tree-based properties and name resolution to gain access to non-local jump targets. This may also be used for static dispatched procedure calls, possibly resulting in rather large control flow graphs.

In general the interaction between names, control- and dataflow, and types is of interest. We are integrating FlowSpec in Spoofax, which has domain specific support for name binding [Konat et al. 2012]. The theoretical foundation for the newest name binding support [Néron et al. 2015] 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 [van Antwerpen et al. 2016] can also model types of a programming language. If we can fully integrate our control- and dataflow work in this framework we can extend the expressiveness of the system to have name resolution or types that depend on control- and dataflow.

We wish to look into safety of the user-defined lattices and property rules. 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 randomised testing.

Furthermore, we would like to verify the correctness of control- and dataflow specifications relative to a dynamic semantics specifications. This would be an extension of the work on the language designer's workbench [Visser et al. 2014].

As we mention in our discussion of the prototype implementation, there are plenty of places where we can optimise the implementation. To do this in a constructive way we will first look into profiling and benchmarking the implementation when it is reasonably complete.

Acknowledgments

We would like to thank Peter Mosses, Guido Wachsmuth and the anonymous reviewers 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.

References

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. DOI: http://dx.doi.org/10.1145/1640089.1640108

Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* 69, 1-2 (2006), 123–178. DOI: http://dx.doi.org/openurl.asp?genre=article&issn=0169-2968&volume=69&issue=1&spage=123

Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69, 1-3 (2007), 14–26. DOI: http://dx.doi.org/10.1016/j.scico.2007.02.003

J. Gosling, B. Joy, G. Steele, and G. Bracha. 2005. *The Java Language Specification* (third edition ed.). Prentice Hall PTR, Boston, Mass.

Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).

Mark Hills. 2014. Streamlining Control Flow Graph Construction with DCFlow. In *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.), Vol. 8706. Springer, 322–341. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_18

Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott (Eds.). ACM, 349–362. DOI: http://dx.doi.org/10.1145/2150976.2151013

Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. 1987. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Informatica* 24, 6 (1987), 679–694.

Martin Jourdan and Didier Parigot. 1990. Techniques for Improving Grammar Flow Analysis. In *ESOP 90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings (Lecture Notes in Computer Science)*, Neil D. Jones (Ed.), Vol. 432. Springer, 240–255.

John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *J. ACM* 23, 1 (1976), 158–171. DOI: http://dx.doi.org/10.1145/321921.321938

John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7 (1977), 305–317.

Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. 2009. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Oege de Moor and Michael I. Schwartzbach (Eds.), Vol. 5501. Springer, 142–157. DOI: http://dx.doi.org/10.1007/978-3-642-00722-4_11

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. DOI: http://dx.doi.org/10.1145/1869459.1869497

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. 194–206.

Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. EASY Meta-programming with Rascal. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science)*, Joao M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 222–289. DOI: http://dx.doi.org/10.1007/978-3-642-18023-1_6

Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Theory Comput. Syst.* 2, 2 (1968), 127–145. DOI: http://dx.doi.org/content/m2501m07m4666813/

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 311–331. DOI: http://dx.doi.org/10.1007/978-3-642-36089-3_18

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. DOI: http://dx.doi.org/10.1145/2908080.2908096

Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. 2007. Extending Attribute Grammars with Collection Attributes–Evaluation and Applications. *Source Code Analysis and Manipulation, IEEE International Workshop on* 0 (2007). DOI: http://dx.doi.org/10.1109/SCAM.2007.13

Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37. DOI: http://dx.doi.org/10.1016/j.scico.2005.06.005

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of program analysis (2. corr. print)*. Springer. DOI: http://dx.doi.org/computer/theoretical+computer+science/book/978-3-540-65410-0

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_9

Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey. 2014. Respect Your Parents: How Attribution and Rewriting Can Get Along. In *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.), Vol. 8706. Springer, 191–210. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_11

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. DOI: http://dx.doi.org/10.1561/2500000014

Jeff Smits. 2016. *The Static Semantics of the Green-Marl Graph Analysis Language*. Master's thesis. Delft University of Technology. Advisor(s) Guido Wachsmuth. Available at http://resolver.tudelft.nl/uuid:4f07cbbb-d017-41e8-aba6-8ff0c19f258d.

Tamás Szabó, Simon Alperovich, Markus Völter, and Sebastian Erdweg. 2016a. An extensible framework for variable-precision data-flow analyses in MPS. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 870–875. DOI: http://dx.doi.org/10.1145/2970276.2970296

Tamás Szabó, Sebastian Erdweg, and Markus Völter. 2016b. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. DOI: http://dx.doi.org/10.1145/2970276.2970298

Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. 2013. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming* 78, 10 (2013), 1809–1827. DOI: http://dx.doi.org/10.1016/j.scico.2012.02.002

Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. DOI: http://dx.doi.org/10.1145/2847538.2847543

Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. DOI: http://dx.doi.org/10.1145/2661136.2661149

Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *PLDI*. 131–145.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54. DOI: http://dx.doi.org/10.1016/j.scico.2009.07.004