Preliminary Proceedings of the Tenth Workshop on Language Descriptions Tools and Applications LDTA 2010

Claus Brabrand and Pierre-Etienne Moreau

March 27-28, 2010

Preface

This volume contains the preliminary proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010), held in Paphos, Cyprus on March 28-29, 2010. LDTA is a two-day satellite event of ETAPS (European Joint Conferences on Theory and Practice of Software) organized in cooperation with ACM Sigplan.

LDTA is an application and tool oriented forum on meta programming in a broad sense. A meta program is a program that takes other programs as input or output. The focus of LDTA is on generated or otherwise efficiently implemented meta programs, possibly using high level descriptions of programming languages. Tools and techniques presented at LDTA are usually applicable in the context of "Language Workbenches" or "Meta Programming Systems" or simply as parts of advanced programming environments or IDEs.

The prelimiary proceedings include an extended abstract based on the invited talk by Jean-Louis Giavitto (*"A Domain Specific Language for Complex Natural & Artificial Systems Simulations"*) and the 11 contributed papers that were selected for presentation and the preliminary proceedings by the programme committee from 30 submissions (i.e., 37% acceptance rate). Every submission was reviewed by at least three members of the program committee. In addition, the program committee sought the opinions of additional referees, selected because of their expertise on particular topics. The final selection of papers was made during the first week of February 2010.

We would like to thank all of the authors who submitted papers to the workshop, and the members of the programme committee for their excellent work. The program committee did not meet in person, but carried out extensive discussions during the electronic PC meeting via EasyChair. We would also like to thank the LDTA organizing committee (Giorgios Robert Economopoulos and Jurgen Vinju) for their assistance and sound counsil, Torbjörn Ekman for contributing to the organization, and the ETAPS organization.

The final proceedings will appear in the ACM Digital Library. All news about LDTA is announced on [`http://www.ldta.info`].

We hope you will enjoy reading these proceedings.

Best regards,
Claus Brabrand and Pierre-Etienne Moreau (co-chairs)

Program Committee

- Claus Brabrand, IT University of Copenhagen, Denmark (co-chair)
- Pierre-Etienne Moreau, Nancy-Universitié & LORIA, France (co-chair)
- Uwe Aßmann, Dresden University of Technology, Germany
- Emilie Balland, INRIA, France
- John Boyland, University of Wisconsin, USA
- Giorgios Robert Economopoulos, University of Southampton, UK
- Magne Haveraaen, University of Bergen, Norway
- Nigel Horspool, University of Victoria, Canada
- Shan Shan Huang, Logic Blox, USA
- Johan Jeuring, Utrecht University, The Netherlands
- Ondrej Lhotak, University of Waterloo, Canada
- Shane Markstrum, University of California, USA
- Peter D. Mosses, Swansea University, UK
- Klaus Ostermann, Philipps-Universität Marburg, Germany
- Marc Pantel, University of Toulouse, France
- Elizabeth Scott, Royal Holloway, University of London, UK
- Eli Tilevich, Virginia Tech, USA
- Eelco Visser, Delft University of Technology, The Netherlands
- Joost Visser, Software Improvement Group, The Netherlands
- Tobias Wrigstad, Purdue, USA

Table of Contents

LDTA 2010

Tenth Workshop on Language Descriptions, Tools and Applications

[Paphos, Cyprus (March 27-28, 2010)]

Saturday (March 27, 2010):

Time	Papers/Presentations	Authors/Speakers
09:00-09:15	Welcome to LDTA 2010	- Pierre-Etienne Moreau & Claus Brabrand
09:15-10:30	SESSION 1: [Invited Talk]	
	- *A Domain Specific Language for Complex Natural & Artificial Systems Simulations*	- Jean-Louis Giavitto
10:30-11:00	*- Tea/Coffee -*	
11:00-12:30	SESSION 2: [Program Analysis]	
	- *On the Rôle of Min. Typing Derivations in Type-driven Program Transformation*	- Stefan Holdermans & Jurriaan Hage
	- *GamaSlicer: an Online Laboratory for Program Verification and Analysis*	- Daniela da Cruz, Pedro Rangel Henriques, & Jorge Sousa Pinto
	- *Dependence Condition Graph for Semantics-Based Abstract Program Slicing*	- Agostino Cortesi & Raju Halder
12:30-14:00	*- LUNCH -*	
14:00-15:00	SESSION 3: [Anniversary Talk]	
	- *Ten Years of LDTA*	- Görel Hedin
15:00-16:00	SESSION 4: [Grammars]	
	- *Faster Ambiguity Detection by Grammar Filtering*	- Bas Basten & Jurgen Vinju
	- *Tear-Insert-Fold Grammars"*	- Adrian Johnstone & Elizabeth Scott
16:00-16:30	*- Tea/Coffee -*	
16:30-18:00	SESSION 5: [Panel Discussion]	
	- *Panel Discussion*	- < TBA >

Sunday (March 28, 2010):

Time	Papers/Presentations	Authors/Speakers
09:00-10:30	SESSION 6: [Mixed LDTA Topics]	
	- *Embedding a Web-Based Workflow Management System in a Functional Language*	- Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, & Peter Achten
	- *Specifying Generic Java Programs: two Case Studies*	- Alain Giorgetti, Claude Marché, Elena Tushkanova, & Olga Kouchnarenko
	- *Language Description for Frontend Implementation*	- Anya Helene Bagge
10:30-11:00	*- Tea/Coffee -*	
11:00-12:30	SESSION 7: [Domain Specific Languages]	
	- *DSL Tools: Less Maintenance?*	- Paul Klint, Tijs van der Storm, & Jurgen Vinju
	- *Using DSLs for Developing Enterprise Systems*	- Margus Freudenthal
	- *Formally Specified Type Checkers for Domain Specific Languages*	- Arjan v.d. Meer, Alexander Serebrenik, Mark v.d. Brand, & Albert Hofkamp
12:30-14:00	*- LUNCH -*	

Proceedings

A Domain Specific Language for Complex Natural and Artificial Systems Simulations

Jean-Louis Giavitto

CNRS – IBISC, Université d'Evry, Genopole,
523 place des Terrasses de l'Agora, 91000 Evry, France

giavitto@ibisc.univ-evry.fr,
http://mgs.spatial-computing.org

A DSL for Systems Biology. Domain specific languages are often designed to incorporate domain-specific knowledge in order to enhance expressiveness (for the programmer) and quality, flexibility, maintenability, ..., of the produced softwares. In this talk I introduce a language initially developed to ease the modeling and the simulation of developmental processes in biology. In this application domain, one must face:

- dynamical processes that are located and move in space,
- processes that interact locally with their (spatial) neighbors,
- a spatial neighborhood that is build (computed) and adjusted as a result of the process activities,
- various style of processes (numerical simulation of ODE and PDE, stochastic processes and discrete deterministic or non-deterministic transition systems).

Ideally, the specification of the biological models should be small and expressive, theoretically well founded and close to the concepts used by the modelers.

These requirements lead to the design of a rule based programming language called MGS. MGS is based on a notion of *spatial n-ary interaction* : such interaction represents by a rule the local evolution of a small subsystems. The structure of the subsystems is represented through topological relationships and is subject to possible drastic changes in the course of time.

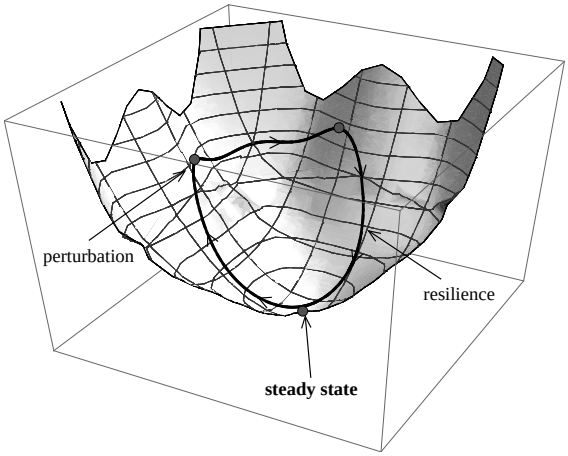
MGS: Topological Rewriting. An MGS rule can also be interpreted as a rewriting rule on *topological chains*. A topological chain is a mathematical structure introduced in algebraic topology and used, e.g. to compute the boundary of an object composed of several elementary pieces of space glued together. With a sufficiently abstract notion of space, MGS subsumes several kind of rewriting (multiset rewriting, word rewriting) and several bio-inspired models of computation (cellular automata, Lindenmayer systems, membrane systems). And a "dictionary" can be developed to link notions relevant to rewriting systems and those relevant to the simulation of dynamical systems:

Rewriting Systems	Dynamical Systems
term	state
set of rules	evolution function
derivation	trajectory
rule application strategy	management of time
normal form	steady state, fixed point
...	...

Algorithmic Examples. MGS has been validated on several large scale examples in the simulation of cellular processes in computational biology. However, the notions of *topological collection* and *transformation* initially introduced to ease the modeling of dynamical systems with a dynamical structure have proven to be useful in algorithmic tasks and we will present during the talk several examples.

MGS rules can represent arbitrary complex computations because the pattern language is very sophisticated. However, in a lot of applications, it appears that the rules corresponds to chain homomorphisms (i.e. transformations that respect the algebraic structure of chains). The computational content of a chain homomorphism is clear: it corresponds to a simple pattern of distributed computations and hence can be viewed as a skeletons that package useful and reusable patterns of parallel and distributed computations.

Autonomic Systems, Dynamical Systems and Rule Based Programming. Recently, the programming style promoted by MGS has been advocated as well suited to express autonomic properties (the so called self-* properties): the reaction rules correspond to the local actions to be taken to react to a perturbation.



The approach can be described as "autonomic computing via trajectory stabilization". In this point of view, an autonomic system is seen as a distributed dynamical system. In the diagram above, the states of this dynamical system are

figured as the ground plane and the system's evolutions are given by a trajectory. The surface represents some potential function, for instance a quantitative evaluation of the divergence of the system from a desired behavior.

When some transient perturbations make the system leave its steady state, the local transformations triggered by the matching of some rules eventually lead to the return of the system's state to an admissible state.

Acknowledgments. Olivier Michel and Antoine Spicher at the University of Creteil are gratefully acknowledged for our long standing collaboration. They have made the MGS project possible and fruitful. Thanks are also due to Pascal Fradet at INRIALPES, Jean-Pierre Banâtre at IRISA, Christophe Godin at INRIA-Montpellier, Annick Lesne at IHES and Przemyslaw Prusinkiewicz at University of Calgary for stimulating discussions and interactions. The works presented here have been partially funded by the University of Evry, the Genopole, the CNRS, the ANR projects NanoProg and AutoChem, and the Paris Institute of Complex System.

References

1. MGS Web Site (Download, Examples, papers, Related links). `http://mgs.spatial-computing.org` See also the web pages of the MGS team: `http://www.lacl.fr/~aspicher`, `http://www.lacl.fr/~michel` and `http://www.ibisc.fr/~giavitto/doku`
2. J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher. Computation in space and space in computation. In *Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 137–152, Le Mont Saint-Michel, Sept. 2005. Springer.
3. J.-L. Giavitto, O. Michel, and A. Spicher. *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, chapter: Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems, pages 235–254. Springer, november 2008.
4. J.-L. Giavitto and A. Spicher. Topological rewriting and the geometrization of programming. *Physica D*, 237(9):1302–1314, jully 2008.
5. IGEM. Modeling a synthetic multicellular bacterium. Modeling page of the Paris team wiki at iGEM'07, 2007. `http://parts.mit.edu/igem07/index.php/Paris/Modeling`.
6. O. Michel. There's plenty of room for unconventional programming languages, or, declarative simulations of dynamical systems (with a dynamical structure), Dec. 2007. Habilitation Manuscript. `http://www.ibisc.univ-evry.fr/~michel/Hdr/hdr.pdf`.
7. O. Michel and F. Jacquemard. *An Analysis of a Public-Key Protocol with Membranes.* In "Applications of Membrane Computing", pages 281–300. Natural Computing Series. Springer Verlag, 2005.
8. O. Michel, A. Spicher, and J.-L. Giavitto. Rule-based programming for integrative biological modeling – application to the modeling of the λ phage genetic switch. *Natural Computing*, 8(4):865–889, Dec. 2009.
9. A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.

On the Rôle of Minimal Typing Derivations in Type-driven Program Transformation*

Stefan Holdermans and Jurriaan Hage

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{stefan,jur}@cs.uu.nl

Abstract

Standard inference algorithms for type systems involving ML-style polymorphism aim at reconstructing principal types for all let-bound identifiers. Using such algorithms to implement modular program optimisations by means of type-driven transformation techniques generally yields suboptimal results. We demonstrate how this defect can be made up for by using algorithms that target at obtaining so-called minimal typing derivations instead. The resulting approach retains modularity and is applicable to a large class of polyvariant program transformations.

1 Introduction

Type-driven program transformation typically proceeds in two logical phases:

1. an *analysis* phase in which the program under transformation is annotated in accordance with a (nonstandard) type system capable of expressing certain properties of interest; and
2. a *synthesis* phase in which the annotations from the previous phase are used to drive the actual transformation of the source program into a target program.

Often such a transformation establishes some form of program optimisation.

A manifest advantage of using types in the analysis phase is that a wide range of techniques and idioms from type systems can be adopted in the design and implementation of transformations. Of particular interest is the use of parametric polymorphism—as found in modern functional programming languages like ML [18] and Haskell [20]—to boost the precision of type-based analyses and

*Submitted as a research paper to LDTA 2010.

to yield transformations that naturally support separate compilation. However, when incorporating ML-style polymorphism into the analysis phase of a type-driven transformation, carefulness is in order: although in literature it is often suggested that implementations of such analyses can well be based on standard inference algorithms for reconstructing types in the Hindley-Milner discipline, in practice the use of such algorithms easily leads to suboptimal transformations. This paper offers a closer look at the problem:

- We demonstrate where adaptations of standard type-reconstruction algorithms for analysis in optimising type-driven program transformations fall short. In particular, we argue that the incentive of such algorithms to associate each let-bound identifier with the principal type scheme of its definiens is at odds with the objective to deliver transformations that are as good as possible.
- To be able to substantiate this claim, we formalise, in the context of modular program optimisation, the notion of "best" transformations with respect to a given polymorphic type system. Concretely, we require such transformations, foremost, to guarantee full correctness in the presence of separate compilation and, next, to subject isolated compilation units to as agressive as possible intramodular optimisation.
- In the process, we articulate the connection between best transformations and so-called *minimal typing derivations* [6].

Throughout the paper we consider, as an example, a simple type-driven transformation for removing dead code from programs written in what is essentially an extended version of the call-by-name lambda-calculus. We stress, however, that the concepts under discussion apply to a whole class of type-driven program transformations, including, for instance, parallelisation [12], dethunkification [2], and update avoidance [22]. Indeed, we consider the most important contribution of this paper its depiction of a general type-based methodology for modular optimising program transformations.

2 Elementary Dead-code Elimination

Let c range over an abstract set of constant symbols and x over a countable infinite set of variable symbols. Then, consider the set of terms given by

$$t \quad ::= \quad c \mid x \mid \lambda x.\, t_1 \mid t_1\ t_2 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \mid \bot.$$

That is, terms are built from constants, variables, lambda-abstractions, function applications, (nonrecursive) local definitions, and the special constant $\bot$ representing a nonterminating or failing computation. As usual, function application associates to the left and lambda-abstractions extend as far to the right as possible.

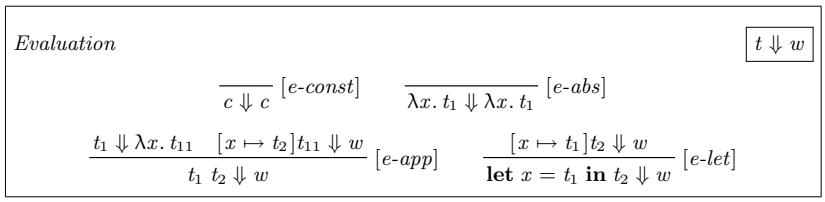


Figure 1: Natural semantics.

Terms are evaluated under a call-by-name strategy. Successful evaluation of a closed term t yields a weak-head normal form w, which is either a constant or a lambda-abstraction:

$$w \quad ::= \quad c \mid \lambda x.\, t_1.$$

Formally, t evaluates to w if the judgement $t \Downarrow w$ can be produced from the set of inference rules in Figure 1. Note that, under this nonstrict semantics, the evaluation of the program $(\lambda x.\, \lambda y.\, x)\ t_1\ t_2$ does not require the second argument term t_2 to be reduced to weak-head normal form. It is the goal of *dead-code elimination* to, within a given program, identify as many of such nonrequired terms as possible and subsequently remove them from the program.

In the sequel, we consider a type-driven approach to dead-code elimination for our term language that breaks down into a type-based *liveness analysis* and a translation that replaces dead terms by the special constant $\bot$.

In the analysis phase we make use of types τ, annotated with liveness properties D and L, ranged over by φ. The idea is to associate the property D with dead code, i.e., code that is guaranteed not to be evaluated, and the property L with live code, i.e., code that may be evaluated. Types are then constructed from a base type $\mathtt{base}$ and annotated function types ${\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2}$:

$$\begin{array}{lcl} \varphi & ::= & \mathsf{D} \mid \mathsf{L} \\ \tau & ::= & \mathtt{base} \mid {\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2}. \end{array}$$

Initially, having type environments Γ map from variables x to pairs τ^{φ} consisting of a liveness type τ and an annotation φ, the transformation is expressed through judgements of the form

$$\Gamma \vdash t \triangleright t' : \tau^{\varphi},$$

indicating that, in the type environment Γ, the source term t can be safely transformed into the target term t' as its liveness properties are captured by the type τ and the annotation φ. As a notational convenience, pairs $({\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2})^{\varphi}$, of which the first component denotes a function type, will be written as ${\tau_1}^{\varphi_1} \xrightarrow{\varphi} {\tau_2}^{\varphi_2}$ and the now annotated function-space constructor $\xrightarrow{\cdot}$ associates to the right.

The rules for deriving transformations are given in Figure 2. The axiom [*t-const*] expresses that constants are considered to be of base type and, depending

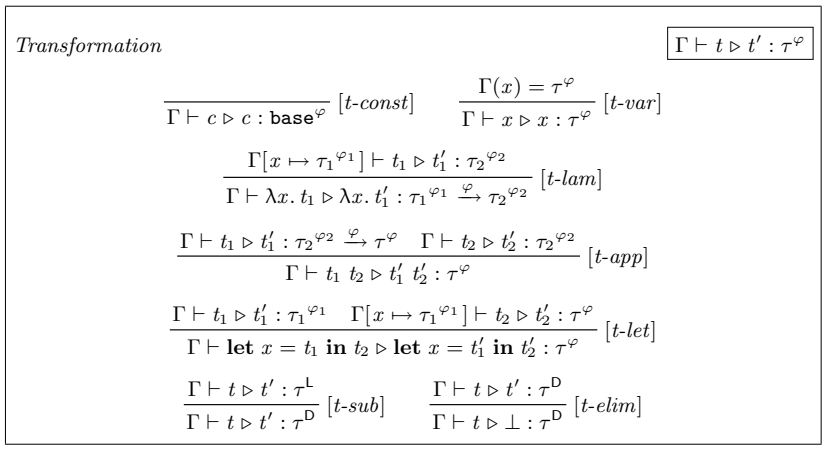


Figure 2: Monovariant dead-code elimination.

on the context in which they appear, can be either dead or live. The rule [*t-var*] states that the type and annotation assigned to a variable have to agree with the corresponding entry in the type environment. In the rule [*t-lam*], assumptions are made for the type and annotation for the formal parameter of a lambda-abstraction and the body of the abstraction is analysed and transformed in a type environment that reflects these assumptions; note that the body is analysed independent from the liveness of the abstraction itself. The rule for applications, [*t-app*], requires that the type and annotation for the argument have to match the type and annotation for the formal parameter of the function; moreover, if the application is live (i.e., if its result may be evaluated), then so is the function. In the rule [*t-let*] for transforming local definitions, the type and annotation obtained for the definiens are added to the type environment and the extended type environment is used to analyse and transform the body of the definition. Rule [*t-sub*] introduces *subeffecting*: it allows for variables that are live at their binding sites to be considered dead at some of their use sites, effectively allowing for more subterms to be identified as dead. As far as the transformation from source to target terms is concerned, all of the aforementioned rules simply carry out the identity transformation; hence, crucial to the intended optimisation is the rule [*t-elim*], which states that a dead term may be eliminated and replaced by the special constant $\bot$. Note that we have not included any rule that deals with occurrences of $\bot$ in source terms; such terms are simply considered ill-typed.

Assuming that a given program as a whole may be evaluated (and thus has to receive the annotation L), transformation proceeds by identifying and eliminating as many D-annotated terms as possible.

Example 1. Consider again the program $(\lambda x.\, \lambda y.\, x)\ t_1\ t_2$ and assume that t_1 and

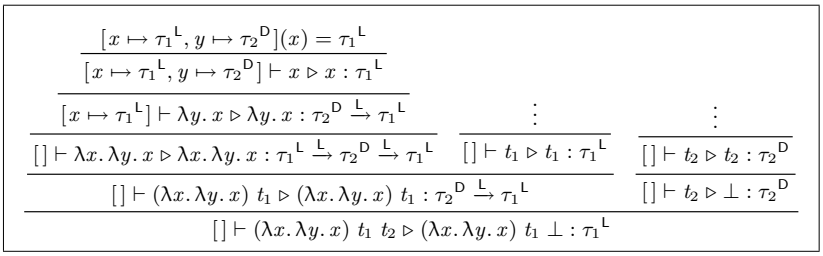


Figure 3: Example derivation.

t_2 are closed subterms of types τ_1 and τ_2, respectively. Then, as the derivation in Figure 3 demonstrates, the second argument term t_2 is in fact dead and can be safely eliminated, yielding the target program $(\lambda x.\, \lambda y.\, x)\ t_1\ \bot$. ∎

Here, "safely" means that the transformation preserves the semantics of the source program. For instance, in the example above, we have that for any weak-head normal form w with $t_1 \Downarrow w$, both the source and the target program evaluate to w.

3 Polyvariant Liveness Analysis

Liveness, as determined in the analysis phase of the transformation from the previous section, is not an intrinsic property: whether a term is live or dead depends on the context in which it appears. As the following two examples illustrate, this is a concern especially for higher-order functions.

Example 2. Let t_1 and t_2 be closed terms of base type in the program

$$\textbf{let}\ \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x)\ \textbf{in}\ \mathit{twice}\ (\lambda y.\, t_1)\ t_2,$$

in which *twice* is applied to a function that never evaluates its argument. Then, *twice* is assigned the liveness type $(\mathtt{base}^{\mathsf{D}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{D}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ and dead-code elimination results in $\textbf{let}\ \mathit{twice} = \lambda f.\, \lambda x.\, f\ \bot\ \textbf{in}\ \mathit{twice}\ (\lambda y.\, t_1)\ \bot$. ∎

Example 3. But in the program

$$\textbf{let}\ \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x)\ \textbf{in}\ \mathit{twice}\ (\lambda z.\, z)\ t,$$

with t a closed term of base type, *twice* is applied to a function of type $\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ and, so, analysis of *twice* yields $(\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$, leaving no terms to be identified as dead. ∎

These examples show that what liveness type to assign to a higher-order function depends on the functions to which it is applied. However, in scenarios that require separate compilation, the arguments to which a function is applied

are, in general, not known at compile-time. So, if a higher-order function like *twice* in the previous examples is exported by a separately transformed module, its liveness analysis becomes a delicate matter.

Since our aim is to facilitate safe, i.e., semantics-preserving, transformations, a straightforward approach to analysing exported or open-scope functions is to subject them to what Wansbrough [23] calls *pessimisation*. That is, we simply assume that any formal parameters of functional type are to be bound to functions that may use all of their arguments in order to produce a result. For instance, if the function *twice* from Examples 2 and 3 above were to be analysed pessimistically, it would receive the liveness type $(\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ (cf. Example 3). Obviously, this strategy leads to a safe transformation as there can be no harm in binding a live argument to a dead parameter: it will just not be used. The other way around, i.e., binding a dead argument to a live parameter, would, however, be unsafe as dead arguments are to be replaced by $\bot$.

Unfortunately, the effects of pessimisation propagate to the use sites of higher-order functions, causing fewer subterms to be identified as dead:

Example 4. Assume that

$$\Gamma \vdash \mathit{twice} \triangleright \mathit{twice} : (\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$$

and that t_1 and t_2 are closed subterms of base type. Then, in the program $\mathit{twice}\ (\lambda y.\, t_1)\ t_2$ (cf. Example 2), the second argument term t_2 is to be assumed live and cannot be eliminated during dead-code elimination. ∎

A better but more involved solution to the problem of dealing with open-scope higher-order functions is to make the transformation *polyvariant* or *context-sensitive*. In type-driven transformation, this is typically achieved by allowing abstraction over the properties of interest in the analysis phase. The resulting type system makes essential use of polymorphic types, much like those of ML and Haskell, but with the important difference that terms are polymorphic in their annotations rather than their types.

To make our dead-code elimination polyvariant, we extend the annotation language with annotation variables drawn from a countable infinite set ranged over by β. Moreover, the set of concrete annotations is thought of as a two-point join-semilattice with $\mathsf{D} \sqsubset \mathsf{L}$ and least upper bounds $\varphi_1 \sqcup \varphi_2$:

$$\varphi \quad ::= \quad \mathsf{D} \mid \mathsf{L} \mid \beta \mid \varphi_1 \sqcup \varphi_2.$$

The type language is stratified into monomorphic types τ and possibly polymorphic type schemes σ:

$$\begin{array}{rcl} \tau & ::= & \mathtt{base} \mid {\tau_1}^{\varphi_1} \to {\tau_2}^{\varphi_2} \\ \sigma & ::= & \tau \mid \forall\beta.\, \sigma_1. \end{array}$$

The annotation variable β is bound in $\forall\beta.\, \sigma_1$; we write $\mathit{fav}(\Gamma)$ for the set of annotation variables that appear free in Γ and $\forall(\beta_1, \cdots, \beta_n).\, {\tau_1}^{\varphi}$ for the pair consisting of the type scheme $\forall\beta_1.\, (\cdots (\forall\beta_n.\, \tau_1) \cdots)$ and the annotation φ. Transformations are now expressed through judgements of the form

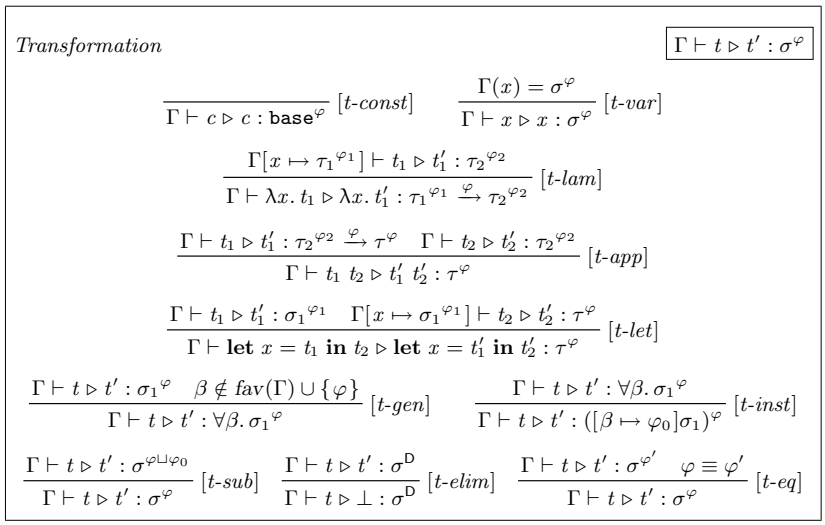


Figure 4: Polyvariant dead-code elimination.

$$\Gamma \vdash t \triangleright t' : \sigma^{\varphi},$$

with type environments Γ mapping from variables x to pairs σ^{φ}.

The rules that constitute the polyvariant transformation are given in Figure 4. Rules [*t-const*], [*t-lam*], and [*t-app*] are identical to their monovariant counterparts, while, in comparison to Figure 2, rules [*t-var*] and [*t-elim*] just make mention of type schemes rather than types. Rule [*t-let*] indicates that let-bound identifiers can have polymorphic types. In rule [*t-sub*], subeffecting is expressed in terms of the least-upper bound operator $\sqcup$. Rule [*t-eq*] expresses that definitional equivalent annotations are interchangeable; here, equivalence, formally defined in Figure 5, simply conveys that annotations are indeed interpreted as elements of a join-semilattice.

Example 5. Using the rules from Figure 4, the function *twice*, defined as

$$\lambda f.\, \lambda x.\, f\ (f\ x),$$

can now be assigned the polymorphic liveness type

$$\forall \beta.\, (\mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}},$$

indicating that the liveness of its second argument depends on the liveness properties of its first argument. ∎

Example 6. Assume that *twice* has the polymorphic liveness type

$$\forall \beta.\, (\mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathsf{base}^{\beta} \xrightarrow{\mathsf{L}} \mathsf{base}^{\mathsf{L}}$$

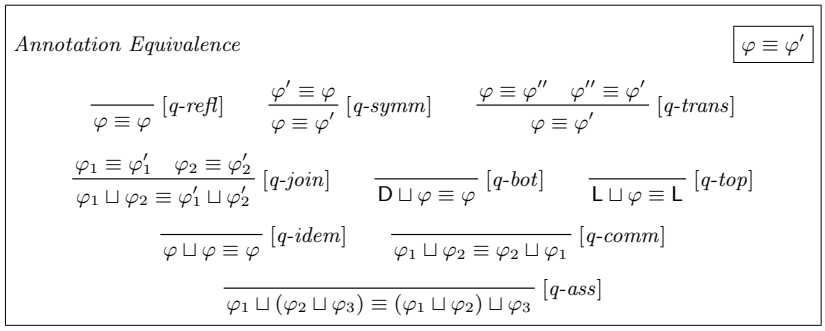


Figure 5: Definitional equivalence of annotations.

and that t_1 and t_2 are closed subterms of base type. Then, in the program *twice* $(\lambda y.\, t_1)\ t_2$, the liveness variable β can be instantiated with D. Consequently, the argument t_2 is annotated with $\mathtt{base}^{\mathsf{D}}$ as well, yielding the target program *twice* $(\lambda y.\, t_1)\ \bot$. ∎

A crucial observation regarding polymorphically driven transformations is that, although pessimisation is no longer propagated to the use sites of open-scope higher-order functions (cf. Example 4), these functions are themselves still transformed pessimistically. For instance, having associated the polymorpic type $\forall\beta.\, (\mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\beta} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$ with the function $\lambda f.\, \lambda x.\, f\ (f\ x)$, we need to consider all possible instantiations of the liveness variable β. In particular, we need to prepare for β being instantiated with L, meaning that the function bound to f requires its argument in order to produce a result. Hence, to keep the transformation safe, no terms can be eliminated from the definition of the higher-order function.

Now, let us formalise our notion of safety. To this end, we first make precise what it means for two terms to have the same semantics.

Definition 1. Two weak-head normal forms w_1 and w_2 are *extensionally equal*, written $w_1 \sim w_2$, if

1. $w_1 = w_2$, or
2. for all terms t_0 and weak-head normal forms w_1', $w_1\ t_0 \Downarrow w_1'$ implies that there exists a weak-head normal form w_2' such that $w_2\ t_0 \Downarrow w_2'$ and $w_1' \sim w_2'$. ∎

We say that two terms have the same semantics if they evaluate to extensionally equal normal forms. Safety of the transformation then follows from the following correctness theorem:

Theorem 2 (Semantic Correctness). If $\Gamma \vdash t \triangleright t' : \sigma^{\mathsf{L}}$ and $t \Downarrow w$, then there exists a w', such that $t' \Downarrow w'$ and $w \sim w'$. ∎

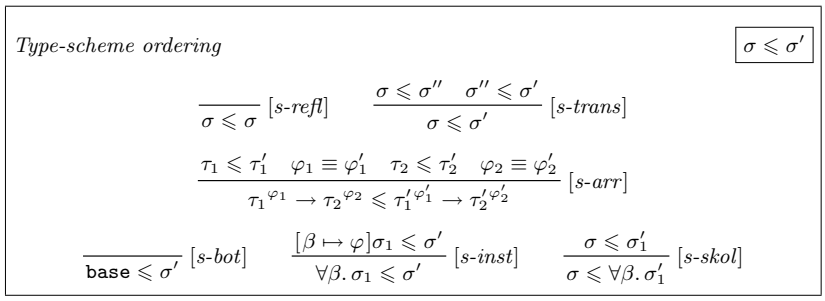


Figure 6: Partial order on annotated type schemes.

4 Minimal Typing Derivations

A clear advantage of a type-driven approach to program transformation is that a wide range of techniques and results from type systems can be readily adapted to a transformational setting. An example was given in the previous section, where an adaptation of ML-style polymorphism was used to render type-driven dead-code elimination more context-sensitive. Now, when implementing the resulting transformation, it seems natural to consider an analogous adaptation of the standard Algorithm W [7] for reconstructing types in ML-like languages and, indeed, such an approach is often suggested in literature. However, as it turns out, carefulness is in order as straightforward adaptations of Algorithm W and other standard inference algorithms, in general, result in transformations that are suboptimal in a sense that will be made precise below.

As we will demonstrate shortly, the main defect of standard inference algorithms in a transformational setting is their incentive to associate *all* let-bound identifiers with their *principal types* [11].

For the polymorphic type system from Section 3, principal types may be defined in terms of a partial order on annotated type schemes, presented in Figure 6.

Theorem 3 (Principal Types). If $\Gamma \vdash t \triangleright t' : \sigma^{\mathsf{L}}$, then there exists a type scheme $\sigma_\star$ such that $\Gamma \vdash t \triangleright t' : \sigma_\star{}^{\mathsf{L}}$ for some t' and $\sigma_\star \leqslant \sigma''$ for all σ'' and t'' with $\Gamma \vdash t \triangleright t'' : \sigma''{}^{\mathsf{L}}$. ∎

Intuitively, a term's principal type $\sigma_\star$ is the most polymorphic type assignable, guaranteeing the highest degree of context-sensitivity. However, assigning principal types to all let-bound identifiers, as Algorithm W and the like aim at, typically results in analyses that are "too polyvariant":

Example 7. Consider again the program

$$\textbf{let } \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x) \textbf{ in } \mathit{twice}\ (\lambda y.\, t_1)\ t_2$$

from Example 2, in which t_1 and t_2 are closed subterms of base type. Assigning *twice* its principal type

$$\forall(\beta_1,\cdots,\beta_6).\,(\mathtt{base}^{\beta_1} \xrightarrow{\beta_1\sqcup\beta_2\sqcup\beta_3\sqcup\beta_4} \mathtt{base}^{\beta_1\sqcup\beta_2\sqcup\beta_3}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\beta_1\sqcup\beta_5} \xrightarrow{\beta_6} \mathtt{base}^{\beta_2}$$

results in a conservative transformation that accounts for the possibility that β_1 will be instantiated with L. Consequently, the given program is transformed into

$$\textbf{let } \mathit{twice} = \lambda f.\,\lambda x.\,f\ (f\ x) \textbf{ in } \mathit{twice}\ (\lambda y.\,t_1)\ \bot,$$

missing out on the opportunity to eliminate the subterm $(f\ x)$ from the definiens of *twice* (cf. Example 2). ∎

As the example demonstrates, context-sensitivity here comes at the expense of conservativeness. Hence, when assigning a possibly redundant polyvariant liveness type to a locally defined function (or, more general, a function with a closed scope), one risks reducing the opportunities for dead-code elimination unnecessarily.

Note, however, that that is not to say that polyvariance is to be avoided for closed-scope functions altogether. Indeed, polyvariance still plays a valuable rôle in keeping pessimisation from propagating to the use sites of higher-order functions.

Example 8. Let t_0 be a binary operation on terms of base type, and t_1 and t_2 closed subterms of base type. Then, assigning a polyvariant liveness type to the locally defined higher-order function *twice* in the program

$$\textbf{let } \mathit{twice} = \lambda f.\,\lambda x.\,f\ (f\ x) \textbf{ in } t_0\ (\mathit{twice}\ (\lambda y.\,t_1)\ t_2)\ (\mathit{twice}\ (\lambda z.\,z)\ t_2)$$

allows for the elimination of the dead argument term t_2 in the first application of *twice*, yielding

$$\textbf{let } \mathit{twice} = \lambda f.\,\lambda x.\,f\ (f\ x) \textbf{ in } t_0\ (\mathit{twice}\ (\lambda y.\,t_1)\ \bot)\ (\mathit{twice}\ (\lambda z.\,z)\ t_2).$$

If we were to assign a monomorphic type to *twice* instead, the only safe choice would be $(\mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}) \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathtt{base}^{\mathsf{L}}$, which forces the analysis to identify both occurrences of t_2 as live, preventing the elimination of the first occurrence. ∎

The problem that a single application of a monomorphically typed higher-order function to a live argument forces all arguments to that function to be live, is known as the *poisoning problem* [24] and the example above shows how it is solved by allowing close-scoped functions to have polymorphic types.

In summary, the problem of standard inference algorithms is not so much that they assign polymorphic types to local functions, but rather that they associate local functions with their most polymorphic type. A better approach would be to assign local functions types that are only as polymorphic as needed:

- If a closed-scope higher-order function is only applied to dead arguments, the relevant formal parameter should receive the monomorphic annotation D, so that the body of the function can be optimised as agressively as possible.

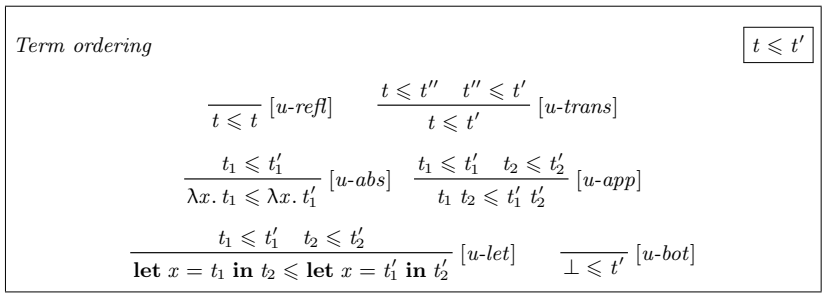


Figure 7: Partial order on terms.

- If a closed-scope higher-order function is only applied to live arguments, its formal parameter could just as well receive the monomorphic annotation L as nothing can be gained by making it context-sensitive.
- If a closed-scope higher-order function may be applied to both dead and live arguments, its formal parameter should be annotated with a polymorphic annotation variable in order to avoid the poisoning problem.

At the same time, to ensure the highest degree of safety and flexibility, exported (i.e., open-scope) functions should be assigned their principal types.

The approach outlined above is suggestive of adapting Bjørner's notion of *minimal typing derivations* [6] to our transformational setting. A typing derivation for a given term and type is minimal if no other typing derivation for the same term and type would avoid type abstractions where the derivation under consideration could not. In our situation, we are interested in derivations for the principal types of separately compiled terms; these derivations then need to be minimal with respect to abstraction over liveness properties.

By definition, the minimality of a typing derivation can not be read from the type that is assigned to a term. Instead, in order to state that a given transformation is not only correct, but also the "best" of all possible transformations of a term, we also need to take into account, as an abstraction of the derivation, the target term that is produced by the transformation. In this target term, unnecessary liveness abstractions that trigger suboptimal transformations, show up as uneliminated terms that could have otherwise been replaced by $\bot$.

To make our notion of best transformations precise, we define a partial order on terms, given in Figure 7. Note that the ordering is congruent and has $\bot$ as its least element. Intuitively, we have that $t \leqslant t'$ if t has more code eliminated than t'.

Now, in addition to the correctness of the transformation (Theorem 2), we have that dead-code elimination yields a target term that is at least as "good" as the corresponding source term:

Proposition 4. If $\Gamma \vdash t \triangleright t' : \sigma^{\varphi}$, then $t' \leqslant t$. ∎

However, we actually wish for a stronger result: in general, a term admits multiple transformations and we are interested in the best of these transformations. But what constitutes the best transformation? First of all, to support separate transformation, we demand that a transformation is as context-sensitive as possible. Therefore, it seems natural to require that the best transformation for an exported term corresponds to the term's principal type (cf. Theorem 3). But still, there may be many derivations that result in a principal type for a given term. From these derivations, we will favour the one that maximises the number of eliminated subterms. Paramountly, the following theorem guarantees the existence of such derivations:

Theorem 5 (Principal Solutions). If $\Gamma \vdash t \triangleright t' : \sigma^{\mathsf{L}}$, then there exists a type scheme $\sigma_\star$ and a term $t'_\star$ such that

1. $\Gamma \vdash t \triangleright t'_\star : \sigma_\star{}^{\mathsf{L}}$,
2. $\sigma_\star \leqslant \sigma''$ for all σ'' and t'' for which $\Gamma \vdash t \triangleright t'' : \sigma''^{\mathsf{L}}$, and
3. $t'_\star \leqslant t''$ for all t'' for which $\Gamma \vdash t \triangleright t'' : \sigma_\star{}^{\mathsf{L}}$. ∎

It then remains to come up with an algorithm that computes such principal solutions. As argued, straightforward adaptions of Algorithm W and the like will not do as these are primarily concerned with computing principal types. Instead, one needs an algorithm that computes minimal derivations for principal types. In Appendix A, we give such an algorithm and corroborate that it indeed produces principal solutions.

5 Related Work

Minimal typing derivations were considered in the context of ML by Bjørner [6] as an alternative to the typing derivations produced by standard inference algorithms such as Algorithm W. Applications that benefit from minimal typing derivations for traditional (i.e., nonannotated) type systems, include unboxing analysis and resolution of overloading. Bjørner gives an algorithm for computing minimal typing derivations, but, contrasting to the algorithm that we present in Appendix A and which computes principal solutions directly, Bjørner's algorithm merely postprocesses suboptimal derivations produced by conventional algorithms such as Algorithm W.

In this paper, we have focussed on enhancing transformation systems that exploit ML-style let-polymorphism as a means to support separate compilation. Orthogonally, others have made efforts toward increasing the modularity of type-based analyses, most notably by considering type systems that admit, in addition or in lieu of mere principal types, so-called *principal typings* [14] and that allow for genuine compositional analysis. Such systems have been succesfully developed on top of rank-2 intersection types [10, 3, 13, 4] and, at the cost of increased implementation effort, approaches based on the work of Kfoury and others [16, 15] seem to allow for analyses that involve intersection types of

arbitrary finite rank. We believe that such systems are amendable to notions of intramodular optimality that are similar to our notion of prinicipal solutions.

Although dead code does not occur often in hand-written code, it does arise frequently as a result from optimising program transformations such as inline expansion and constant propagation (see, for instance, Aho et al. [1]). Also, programs extracted from proofs conducted in logical frameworks typically carry a significant share of dead code [19].

Dead-code elimination is a special instance of useless-code elimination which intends to avoid computations that have no effect on the outcome of a computation, thus reducing execution time. Dead-code elimination only aims to identify expressions that never need to be evaluated and is mainly intended to reduce program size. A related analysis is useless-variable elimination, which intends to discover variables and arguments to functions that are not relevant to the outcome. Damiani and Giannini [9] suggest that an effective approach to useless-code elimination is to first replace unneeded computations by $\perp$ (dead-code elimination) and then apply useless-variable elimination to further optimise the program. Many authors have contributed to the investigation of type-driven useless-variable elimination. Kobayashi [17], for example, defines a type and effect system for useless-variable elimination and presents a reconstruction algorithm that closely follows Algorithm W, enjoying similar properties in terms of ease of implementation and efficiency. Kobayashi is the first to provide examples of the interaction between useless-variable elimination and polymorphism: useless-variable elimination can make functions more polymorphic and polymorphism allows for more useless-variable elimination. For a more comprehensive overview of the field, the reader is referred to Berardi et al. [5] and Daminani [8].

6 Conclusion

Within the context of modular type-driven program transformation, we have considered how minimal typing derivations may amplify program optimisation.

In the interest of separate compilation, selecting the principal type for an exported function implies that no assumption is made about the contexts in which the function will be used, and, thus, ensures that full flexibility is maintained. The choice for a minimal typing derivation, on the other hand, ensures that, local to a module, the number of opportunities for optimisation is maximised. Importantly, this is done without endangering safety and without changing the principal types of any exported functions.

We have illustrated our approach by means of an intentionally easy polyvariant transformation for dead-code analysis. We stress, however, once more that our ideas also apply to other, more involved, type-based transformations such as parallelisation, dethunkification, and update avoidance.

Acknowledgements. This work was supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Boston, Massachusetts, 2nd edition, 2006.

[2] Torben Amtoft. Minimal thunkification. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Static Analysis, Third International Workshop, WSA '93, Padova, Italy, September 22–24, 1993, Proceedings*, volume 724 of *Lecture Notes of Computer Science*, pages 218–229. Springer-Verlag, 1993.

[3] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9–11, 1997*, pages 1–10. ACM Press, 1997.

[4] Anindya Banerjee and Thomas P. Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathemathical Structures in Computer Science*, 13(1):87–124, 2003.

[5] Stefano Berardi, Mario Coppo, Ferruccio Damiani, and Paola Giannini. Type-based useless-code elimination for functional programs. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation, International Workshop SAIG 2000, Montreal, Canada, September 20, 2000, Proceedings*, volume 1924 of *Lecture Notes in Computer Science*, pages 172–189. Springer-Verlag, 2000.

[6] Nickolaj Bjørner. Minimal typing derivations. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida (USA), June 25–26, 1994*, pages 120–126, 1994. The proceedings of the workshop have been published as a technical report (2265) at the Institute National Recherche en Informatique et Automatique.

[7] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.

[8] Ferruccio Damiani. A conjunctive type system for useless-code elimination. *Mathematical Structures in Computer Science*, 13(1):157—197, 2003.

[9] Ferruccio Damiani and Paola Giannini. Automatic useless-code elimination for HOT functional programs. *Journal of Functional Programming*, 10(6):509–559, 2000.

[10] Ferruccio Damiani and Frédéric Prost. Detecting and removing dead-code using rank 2 intersection. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop, TYPES 1996, Aussois, France, December 15–19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 66–87. Springer-Verlag, 1998.

[11] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the Americal Mathematical Society*, 146:29–60, 1969.

[12] Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic parallelization of lazy functional programs. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26–28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 1992.

[13] Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 209–221. ACM Press, 1998.

[14] Trevor Jim. What are principal typings and what are they good for? In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, Papers Presented at Symposium, St. Petersburg Beach, Florida, 21–24 January 1996*, pages 42–53. ACM Press, 1996.

[15] Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and Joe B. Wells. Relating typabililty and expressiveness in finte-rank intersection type systems (Extended abstract). In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27–29, 1999*, pages 90–101. ACM Press, 1999.

[16] Assaf J. Kfoury and Joe B. Wells. Principality and decidable type inference for finte-rank intersection types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 161–174. ACM Press, 1999.

[17] Naoki Kobayashi. Type-based useless variable elimination. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22–23, 2000*, pages 84–93, 2000.

[18] Robin Milner, Mads Tofte, Robin Harper, and David B. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[19] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Langauges, Austin, Texas, January 1989*, pages 89–104. ACM Press, 1989.

[20] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

[21] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[22] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25–28 June 1995*, pages 1–11. ACM Press, 1995.

[23] Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, University of Cambridge, 2002.

[24] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 15–28. ACM Press, 1999.

GamaSlicer: an Online Laboratory for Program Verification and Analysis

Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto

Informatics Department, University of Minho
Braga, Portugal

Abstract

In this paper we present the GamaSlicer tool, which is primarily a semantics-based program slicer that also offers formal verification (generation of verification conditions) and program visualization functionality. The tool allows users to obtain slices using a number of different families of slicing algorithms (precondition-based, postcondition-based, and specification-based), from a correct software component annotated with pre and postconditions (contracts written in JML-annotated Java). Each family in turn contains algorithms of different precision (with more precise algorithms being asymptotically slower). A novelty of our work at the theoretical level is the inclusion of a new, much more effective algorithm for specification-based slicing, and in fact other current work at this level is being progressively incorporated in the tool.

The tool also generates (in a step-by-step fashion) a set of verification conditions (as formulas written in the SMT-lib language, which enables the use of different automatic SMT provers). This allows to establish the initial correctness of the code with respect to their contracts.

1 Introduction

The goal of program verification is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behavior).

Modern program verification systems rely on the use of a *Verification Conditions Generator* (VCGen for short), a program that reads in a piece of code together with a specification, and produces a set of first-order proof obligations (called *verification conditions*) whose validity will imply that the code is (partially or totally) correct with respect to the specification. The underlying

theoretical framework is some program logic, typically Hoare logic [12] (but other more recent and sophisticated logics exist, such as separation logic [17]). Specifications are expressed in terms of preconditions and postconditions, which are in general formulas of first-order logic.

In this context, the *Design by Contract* approach to software development [15], which proposes that a program's procedures / methods be annotated with contracts that can be checked at runtime, may be fruitfully combined with program verification, since the notion of contract coincides precisely with the above notion of specification. The idea behind contract-based verification is that a program (defined as a set of mutually recursive procedures with contracts) is correct if all its individual procedures are correct with respect to their own individual contracts. Thus procedures can be independently verified in a modular way.

The combination of *slicing techniques* with program verification has been proposed [11, 5, 4] with a double goal: on one hand, slicing may help in optimizing the verification process, allowing one to delete the statements that do not contribute to the validity of a given specification (the resulting slice is smaller and thus easier to verify than the original program). On the other hand, verification inspires a novel, much more powerful way of slicing programs (with respect to traditional, syntax-based slicing), based on their axiomatic semantics.

In this paper we introduce GamaSlicer, a tool that includes both traditional program verification functionality (it is a Verification Conditions Generator) and also a highly parameterizable semantic slicer. It includes all the published algorithms as well as a new, more effective algorithm that we have developed. The tool is extremely useful for comparing the effectiveness of slicing algorithms; it also allows users to perform diverse workflows such as for instance

- First verifying a program with respect to a given specification; then slicing it with respect to that specification to check if there are irrelevant commands (with respect to that specification).
- From a verified program, producing a specialization by weakening the specification and slicing the program with respect to the weaker spec. This may be useful in a program reuse context, in which a weaker contract is required of a component than the actually implemented contract.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background on program verification and slicing to give theoretical support to GamaSlicer. Section 3 presents the architecture of GamaSlicer, and its features. Section 4 discusses applications and usage of GamaSlicer. Section 5 closes the papers with some remarks and future work.

2 Theoretical Background

We give here a brief overview of the theoretical framework underlying GamaSlicer, but we omit formal definitions and proofs. The reader is directed to [10, 7] for more details.

$$
\begin{array}{rcl}
\mathsf{wp}\,(\mathbf{skip}, Q) & = & Q \\
\mathsf{wp}\,(x := e, Q) & = & Q[x \mapsto e] \\
\mathsf{wp}\,(C_1; C_2, Q) & = & \mathsf{wp}\,(C_1, \mathsf{wp}\,(C_2, Q)) \\
\mathsf{wp}\,(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) & = & (b \rightarrow \mathsf{wp}\,(C_t, Q))\ \&\&\ (\ !b \rightarrow \mathsf{wp}\,(C_f, Q)) \\
\mathsf{wp}\,(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) & = & I \\
\mathsf{wp}\,(\mathbf{call}\ \mathbf{f}(\overline{e}), Q) & = & \mathsf{pre}(\mathbf{f})[\overline{x} \mapsto \overline{e}], \text{where } \overline{x} = \mathbf{param(f)} = x_1, ..., x_n \text{ and } \overline{e} = e_1, ..., e_n. \\
\mathsf{wp}\,(y := \mathbf{fcall}\ \mathbf{f}(\overline{e}), Q) & = & \mathsf{pre}(\mathbf{f})[\overline{x} \mapsto \overline{e}] \\
\mathsf{wp}\,(C; \mathbf{return}\, e, Q) & = & \mathsf{wp}\,(C, Q[result \mapsto e]) \\
 & & \\
\mathsf{VC}(\mathbf{skip}, Q) & = & \mathtt{true} \\
\mathsf{VC}(x := e, Q) & = & \mathtt{true} \\
\mathsf{VC}(C_1; C_2, Q) & = & \mathsf{VC}(C_1, \mathsf{wp}\,(C_2, Q))\ \&\&\ \mathsf{VC}(C_2, Q) \\
\mathsf{VC}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) & = & \mathsf{VC}(C_t, Q)\ \&\&\ \mathsf{VC}(C_f, Q) \\
\mathsf{VC}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C, Q) & = & (I\ \&\&\ b) \rightarrow \mathsf{wp}\,(C, I)\ \&\&\ \mathsf{VC}(C, I)\ \&\&\ (I\ \&\&\ !b) \rightarrow Q \\
\mathsf{VC}(\mathbf{call}\ \mathbf{f}(\overline{x}), Q) & = & \mathsf{post}(\mathbf{f})[\overline{x} \mapsto \overline{e}] \rightarrow Q \\
\mathsf{VC}(y := \mathbf{fcall}\ \mathbf{f}(\overline{x}), Q) & = & \mathsf{post}(\mathbf{f})[\overline{x} \mapsto \overline{e}] \rightarrow Q[y \mapsto result] \\
\mathsf{VC}(C; \mathbf{return}\, e, Q) & = & \mathsf{VC}(C, Q[result \mapsto e])
\end{array}
$$

Figure 1: VCGen algorithm: weakest precondition and verification conditions

Program Verification. The verification infrastructure consists of a program logic used to assert the partial correctness of individual procedures, from which a VCGen algorithm is derived (Figure 1). The VCGen is proved correct, i.e. it is guaranteed to generate, from a Hoare triple $\{P\}\, C\, \{Q\}$, a set of proof obligations whose validity is sufficient for the triple to be valid, i.e. for the program C to be correct with respect to precondition P and postcondition Q (technically, the correctness result states that there must exist a Hoare logic derivation having the triple as conclusion).

The standard method for generating verification conditions relies on an algorithm that uses the weakest precondition strategy. Our VCGen algorithm is based on the usual function $\mathsf{wp}\,(C, Q)$ that calculates, from a command C and a postcondition Q, the weakest precondition required to grant the truth of Q after terminating executions of C; the auxiliary function $\mathsf{VC}(C, Q)$ returns a set of verification conditions sufficient to ensure the validity of the Hoare triple $\{\mathsf{wp}\,(C, Q)\}\, C\, \{Q\}$. Notice that determining weakest preconditions of loops is straightforward in our context, since all loops are annotated with invariants that are taken into account.

For a given Hoare triple, the VCGen simply collects the verification conditions generated for each procedure and function with the VC function using the contract's postcondition, with an additional condition stating that the contract precondition must be stronger than the calculated weakest precondition. Let Π be a program consisting of procedures C_i with $i \in \{1, \ldots, n\}$, each annotated with a contract consisting of precondition P_i and postcondition Q_i. Then the verification conditions for Π are given as

$$\bigcup_{i=1 \ldots n} \{P_i \rightarrow \mathsf{wp}\,(C_i, Q_i)\} \cup \mathsf{VC}(C_i, Q_i)$$

Thus the validity of the verification conditions generated from Π implies the correctness of all the procedures in Π with respect to their contracts.

A final note with respect to establishing the validity of first-order logic formulas: the VCGen produces verification conditions whose validity must be checked by some external proof tool; the slicing algorithms described next also require the use of an external proof tool. GamaSlicer uses SMT-solvers for this purpose.

Program Slicing. The basic idea of *slicing* – a code analysis technique introduced by Weiser [18] – is to isolate a subset of program statements, either those (directly or indirectly) *contributing to* the value of a set of variables V_s at a program location p, or those *influenced by* the value of that set of variables at location p. These two forms are known as *backward slicing* and *forward slicing* respectively; $C(p, V_s)$ is called a *slicing criterion*. Statements not interfering with the set of variables isolated are removed, enabling software engineers to concentrate on the statements that are relevant for the task at hand.

Comuzzi et al [5] and Chung et al [4] have provided algorithms for code analysis enabling to identify spurious commands (commands that do not contribute to the validity of the postcondition). The former paper presents a variant of program slicing, called *p-slice* or *predicate slice*, using Dijkstra's weakest preconditions to determine which statements will affect a specific predicate. The latter argues that the information present in the annotations helps to produce more precise slices by removing statements that are not relevant to that specification.

These two papers lay the foundations of slicing based on preconditions (a semantic form of forward slicing) or on postconditions (a semantic form of backward slicing). Both notions can be implemented at different levels of precision (more precision requires slower execution times; for this reason GamaSlicer allows the user to select between linear-time and quadratic time algorithms on the length of the program) The notion of *specification-based slicing*, also introduced by Chung, can be informally described as follows: given a program C which is correct with respect to precondition P and postcondition Q, it is safe (in the sense that the resulting program will still be correct with respect to the same specification) to remove from C all the statements that will never be executed (because P precludes that execution), as well as all the statements whose execution does not affect the truth of Q in the final state of the program.

The algorithm proposed [4] for implementing specification-based slicing consists of first slicing the program with respect to the precondition (ignoring the postcondition), and then with respect to the postcondition (ignoring the precondition). In our theoretical work we have found that a much more precise specification-base slice is obtained by algorithms that use simultaneously the precondition and postcondition. The trade-off between execution time and precision is still present, so different algorithms can be given based on this principle.

Examples comparing all these notions of slicing and the algorithms used to calculate them, together with the definition of our own specification-based slicing, can be found in [7]. Note that since semantic slicing relies on first-order logic (a given statement is sliced off if some first-order formula can be proved), it is of course a conservative transformation – if some formula (possibly valid) cannot be proved, then the corresponding command will not be sliced off.

Visualization. The problem of *visualizing program slices* is a part of the larger problem of *program visualization* – the relevant question here is how to display the interdependencies and relationships among program components in an user-friendly way, that effectively helps understanding programs. As slicing removes non-relevant statements from the source code, it is important that the visual representation clearly distinguishes sliced statements from other, remaining statements. This is a challenge.

The visualization of a program, using a System Dependency Graph (SDG) actually helps in perceiving the relationships holding among program components, and has been widely used by many other tools [13, 3, 1, 2]. However, as we are dealing with programs with contracts, we felt the need to extend this notion of SDG. We call this extension *Annotated System Dependence Graph*, SDG_a for short. Essentially, an SDG_a is an SDG in which some nodes are *annotated*. An annotated node is a block composed by a statement or a predicate (a control statement or entry node) and one or more annotations (a precondition, a post-condition, or an invariant). We are currently looking at suitable ways to exhibit the SDG_a in the context of GamaSlicer in order to visualize assertion-annotated programs as well as calculated program slices.

3 GamaSlicer, an Overview

GamaSlicer is an online laboratory[1] that includes a VCGen, a parameterizable slicer with a choice of algorithms, and visualization functionality (not present in the current version). It works on Java programs with JML annotations (the standard specification language for Java [14]). Instead of programs consisting of sets of procedures, one has classes with their methods, sharing a set of class/instance variables instead of global variables. The fundamental idea remains the same: the verification task concerns a set of mutually recursive methods.

The architecture of GamaSlicer, inspired by that of a compiler (or generally speaking a language processor), is depicted in Figure 2. It is composed by the following blocks: a Java/JML front-end (a parser and an attribute evaluator); a verification conditions generator; a proof obligations generator; a theorem-prover (not included in Figure 2, as we call an external one instead of building our own); a slicer; and an annotated system dependency graph (SDG_a) visualizer (to be incorporated in the next version of the tool).

The tool outputs proof obligations written in the SMT-Lib (Satisfiability Modulo Theories library) language. We chose SMT-Lib since it is nowadays the language employed by most provers used in program verification, including, among many others, Z3 [8], Alt-Ergo [6], and `Yices` [9].

After uploading a file containing a piece of Java code together with a JML specification, the code is recognized by the front-end (a C# analyzer produced automatically from an attribute grammar with the help of the ANTLR parser

[1] Current version available at `http://gamaepl.di.uminho.pt/gamaslicer`

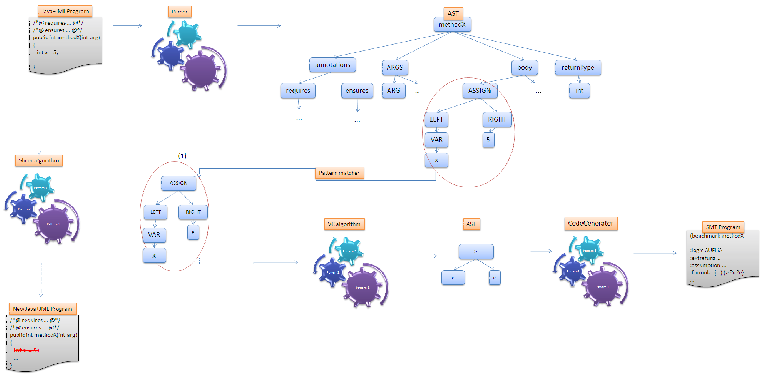


Figure 2: GamaSlicer architecture

generator [16]), and is transformed into an AST. During this first step also an identifiers table is built.

In the next step, the VCGen, implemented in C# as a tree-walker evaluator, traverses the AST to generate the verification conditions, using a kind of tree-pattern matching strategy (represented in Figure 2); each time a tree matches a pattern (corresponding to the eight cases identified in the algorithm of Figure 1), it is transformed and marked as *visited*. Before this operation is performed, a tree traversal is done to look for methods with contracts; this search returns a set of subtrees, and the VCGen algorithm is applied to each one.

In a third step, the new AST (now with verification conditions attached to the nodes) is traversed by the SMT code generator (another C# tree-walker evaluator) to produce SMT proof obligations. If selected by the user, an external automatic Theorem-Prover is then called to prove the generated SMT formulae. Actually, when this facility is activated, three theorem provers are called, to allow for results and performances to be compared.

Alternatively to step three, a fourth step can be performed. Using the AST to derive the SDG_a for the Java/JML class under analysis, the contract-based slicer applies the user-selected algorithm in order to detect and remove a set of statements that are not relevant with respect to the specification annotated in the program, following the discussion in the previous section.

The fifth step is a graphical add-on, that is currently being developed; it takes the SDG_a, before or after slicing, and exhibits the data and control flow inside the methods and inside the class. Statements deleted by the (contract-based) slicer are illustrated on the SDG_a graphical representation – although implemented and tested separately, this visualizer is not yet integrated in the publicly available version of GamaSlicer.

The intermediate information that becomes available after each of the above

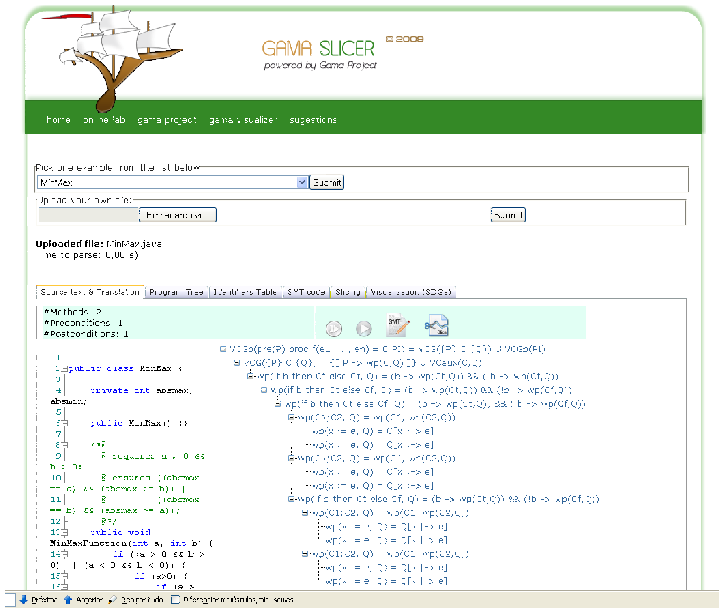


Figure 3: Tab 1: Source Program and VCGen Rules applied

steps is displayed in an internet browser window distributed by six main tabs:

- *Tab 1*: contains the Java/JML source program to be verified and also the rules applied along the verification conditions generation process; some statistical information is also displayed at the top of the window, describing the program size and annotation complexity (see Figure 3).
- *Tab 2*: contains the syntax tree (AST) generated by the front-end.
- *Tab 3*: contains the identifiers table built during the analysis phase (also at step one).
- *Tab 4*: contains the generated SMT code; it will also display a table with the verification status of each formula (`sat`, `unsat`, `unknown`) after calling a theorem-prover; for each prover invoked, the time consumed to prove the formulae is also displayed.
- *Tab 5*: contains the new program produced by the slicer (applying the specification-based slicing techniques to the original program); notice that useless statements identified by the slicer are not actually removed, but shown in red and strike-out style (see Figure 4).

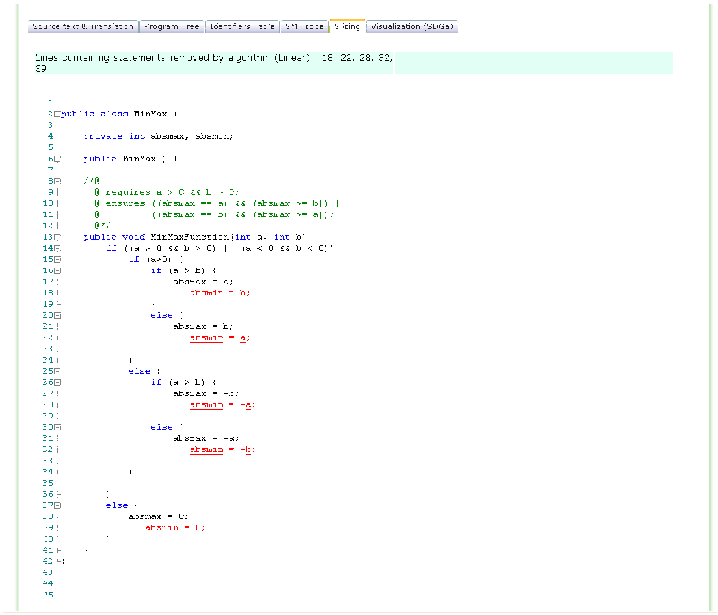


Figure 4: Tab 5: Sliced Program

- *Tab 6*: displays the SDG_a as the visual representation of the program, giving an immediate and clear perception of the program complexity, with its procedures and the relationships among them; nodes in the program's graph that were sliced away are marked in red (helping to understand quickly which statements were removed / preserved). As navigation capabilities over the visual representation, GamaSlicer allows to expand and collapse nodes, and to jump into the source code by clicking a node.

4 GamaSlicer, Usage and Applications

The first motivation for the design and development of this online laboratory was to make available a tool that implements semantic slicing algorithms, with the well-known applications made possible by the combination of verification and slicing technology, while at the same time allowing us to test our improved slicing algorithms.

We now list some applications of the tool. Note that in a properly annotated program, slicing should leave the code intact, thus one first application is to remove useless code. Other applications involve slicing the program using to weaker specifications (with respect to the original satisfied by the program).

An obvious one is program comprehension: slicing a method according to different contracts (by weakening the original contract) allows one to understand the relation between each code section and each part of the initial contract, and also to slice programs for reuse or specialization purposes.

GamaSlicer can also be used to help in detecting and fixing errors, both in annotations and in the methods' code, since it exhibits in a versatile and user-friendly tree fashion all the rules used to generate the proof obligations; this output can optionally be obtained by executing the VCGen algorithm step-by-step. The proof obligations, written in the SMT-Lib language, are also displayed, allowing the user to analyze the formulae and their verification status.

Due to the outputs delivered and the interaction modes, we have found that GamaSlicer can assist programmers with the improvement of their skills to annotate programs. This application trend is clearly reinforced by the slicer, which detects and displays statements that do not contribute to the specification.

From our experience, this system can also be a useful teaching tool, since it allows students to observe step-by-step how verification conditions are generated. For example, since loop invariants and procedure contracts are annotated into the code, the only arbitrary choice is in the rule for the sequence command $C_1 ; C_2$, in which an intermediate assertion R must be guessed. This is the motivation for introducing a strategy for the construction of proof trees, based on the notion of weakest precondition. The result is a mechanical method for constructing derivations, which is inbuilt in the VCGen. This transition from program logic to VCGen is not trivial to understand, and this tool clearly helps in that process. In an advanced course, it is also useful that students can easily modify the underlying algorithms (both VCGen and slicing).

5 Conclusion

In this paper we propose a demonstration of a tool that is intended, primarily, as a development laboratory to test ideas and algorithms in the context of *verification condition generation* and *semantics-based slicing*; even so, it is, to our knowledge, the only available tool that implements semantic slicing based on contracts, as well as integration of verification and slicing capabilities. The tool is web-based, which means that everyone can use it without having to download any source or executable code. Its interface is very simple and its usage completely intuitive. A repository of Java/JML example programs is available from our online Laboratory.

As future work, we intend to work on the scalability of the tool to handle real-size code, as well as improving the visualizer component and including other slicing algorithms which are currently being developed.

References

[1] Paul Anderson and Tim Teitelbaum. Software inspection using Codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.

[2] Giuliano Antoniol, Roberto Fiutem, G. Lutteri, Paolo Tonella, S. Zanfei, and Ettore Merlo. Program understanding and maintenance with the CANTO environment. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 72, Washington, DC, USA, 1997. IEEE Computer Society.

[3] Francoise Balmas. Displaying dependence graphs: a hierarchical approach. *J. Softw. Maint. Evol.*, 16(3):151–185, 2004.

[4] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.

[5] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.

[6] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[7] Daniela da Cruz, Jorge Sousa Pinto, and Pedro Rangel Henriques. Specification-based slicing and slice graphs. `http://alfa.di.uminho.pt/~danieladacruz/techReportCPH09b.pdf`, October 2009.

[8] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[9] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[10] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. Technical Report DI-CCTC-08-01, Universidade do Minho, 2008.

[11] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. *icsm*, 00:138, 2001.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[13] Jens Krinke. Visualization of program dependence and slices. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2004.

[15] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[16] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[17] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[18] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Dependence Condition Graph for Semantics-based Abstract Program Slicing

Raju Halder and Agostino Cortesi

Università Ca' Foscari, Venezia, Italy
{halder, cortesi}@unive.it

Abstract

Many slicing techniques have been proposed based on the traditional Program Dependence Graph (PDG) representation. In traditional PDGs, the notion of dependency between statements are based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Mastroeni and Zanardini introduced a semantics-based dependency both at concrete and abstract domain. This (abstract) semantic dependency is computed at expression level over all possible (abstract) states appearing at program points. In this paper we strictly improve this approach by (*i*) considering the semantic relevancy of the statements (not only the expressions), and (*ii*) adopting conditional dependency. This allows us to transform the semantics-based (abstract) PDG into an semantics-based (abstract) Dependence Condition Graph (DCG) that enables to identify the conditions for dependence between program points. The resulting program slicing algorithm is stricly more accurate than the Mastroeni and Zanardini's one.

1 Introduction

Program slicing is an well-known decomposition technique that extracts from programs the statements which are relevant to a given behavior. In particular, a slice is an executable program whose behavior must be identical to a specific subset of the original behavior, and is obtained by deleting statements from the original program. The notion of program slice was originally introduced by Mark Weiser [29]. It is a fundamental operation for addressing many software-engineering problems, including program understanding, debugging, maintenance, testing, parallelization, integration, software measurement etc [7, 17, 21, 1, 19, 16, 14, 12, 8, 23, 24].

Static [29] and dynamic slicing [15] are two different types of slicing techniques where the former is done at compile time while later is performed at runtime with user inputs. A static slice preserves the program's behavior with respect to variables for all program inputs. The slicing criteria is denoted by

$\langle p, v\rangle$ where v is the variable of interest at program point identified by line number p. This is not restrictive, as it can be easily extended to slicing with respect to a set of variables V, formed from the union of the slices on each variable in V. In contrast, in dynamic slicing, programmers are more interested in a slice that preserves the program's behavior for a specific program input rather than for all program inputs: the dependencies that occur in a specific execution of the program are taken into account.

Over the last 25 years, many slicing techniques have been proposed based on the traditional Program Dependence Graph (PDG) representation [18, 13, 27, 11, 10, 22, 24, 6, 23, 26]. The PDG makes explicit both the data and control dependencies for each operation in a program. Data dependencies have been used to represent only the relevant data flow relationship of a program while the control dependencies are derived from the actual control flow graph. The sub-graph of the PDG induced by the control dependence edges is called the control dependence graph (CDG) and the sub-graph induced by the data dependence edges is called the data dependence graph (DDG).

In traditional PDGs, the notion of dependency between statements are based on syntactic presence of a variable in the definition of the another variable or on a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependencies.

Mastroeni and Zanardini in [20] introduced a semantics-based dependency which fills up the existing gap between syntax and semantics. Based on this semantic dependency, a more precise PDG can be obtained by removing the false dependencies from the traditional syntactic PDG. The semantic dependency can also be lifted to an abstract domain where dependencies are computed with respect to some specific properties of interest rather than concrete values. This (abstract) semantic dependency is computed at expression level over all possible (abstract) states appearing at program points.

In [28], Sukumaran et al. introduced Dependence Condition Graph (DCG), a refinement of PDGs based on the notion of conditional dependency. This is obtained by adding the annotations which encodes the condition under which a particular dependence actually arises in a program execution.

In this paper we combine these two concepts: semantic data dependency [20] and conditional dependency [28], (*i*) by considering the semantic relevancy of the statements (not only the expressions), and (*ii*) by adopting conditional dependency. This allows us to transform the semantics-based (abstract) PDG into an semantics-based (abstract) DCG that enables to identify the conditions for dependence between program points. The resulting (abstract) slicing algorithm is strictly more precise than the one in [20].

The rest of the paper is organized as follows: Section 2 provides a motivating example. Section 3 recalls some basic ideas about abstract interpretation theory covering the abstract semantics of the expressions, statements and the induced partitioning over the abstract domain. Section 4 introduces the (abstract) semantic relevancy of the statements. In section 5 we extend the semantics-based (abstract) PDGs into DCGs. Finally, Section 6 draws our conclusions.

2 A motivating example

In [20], the traditional syntactic based PDG is refined by introducing the notion of semantic dependency both at concrete and abstract level. The semantic dependency computes the data dependency between an expression e and the set of variables $var(e)$ involved in that e at a program point p. The expression e does not semantically depend on a variable $x \in var(e)$ if the evaluation of e over any two different states σ_1 and σ_2 appearing at p where $\forall y \in var(e) \wedge y \neq x : \sigma_1(y) = \sigma_2(y)$, results the same values for e. Although the presence of variable x in expression e shows the syntactic data dependency of e on x, semantically there is no such dependency. For instance, the expression $e = x + x - 2x + 4$ does not depend semantically on x. The concrete semantic data dependency can easily be lifted to an abstract domain representing specific property of interest. The abstract semantic data dependency describes the semantic dependency on abstract values rather than the concrete values. This (abstract) semantic dependency is used to eliminate some irrelevant dependencies from the the traditional PDG, resulting a more precise slice.

However, if we observe carefully, we see that the semantic dependency derived at expression level [20] does not always result into a more precise PDG for the program.

Example 1 *Consider the program P and the corresponding traditional Program Dependence Graph (G_{pdg})[1] as depicted in Figure 1. Suppose we are interested only on the sign of the program variables, and we consider the abstract domain SIGN = $\{\bot, +, 0, -, 0^+, 0^-, \top\}$ where 0^+ represents $\{x \in Z : \ x \geq 0\}$ and 0^- represents $\{x \in Z : \ x \leq 0\}$. After computing the abstract semantic data dependency [20] w.r.t. the sign property, we get the semantics-based abstract PDG shown in Figure 2(a). Observe that, since the semantic dependency w.r.t. SIGN removes the data dependency between y_4 and w at statement 12 (as "4w mod 2" always yields to 0), the corresponding data dependence edge $5 \xrightarrow{w} 12$ is disregarded from the traditional PDG. The slicing algorithm based on this semantics-based abstract PDG with the criteria $\langle 13, y \rangle$ would return the program depicted in Figure 2(b). At program point 9, the variable x_2 may have any abstract value in $\{+, 0, -\}$. Since the evaluation of the expression $x_2 \times 2$ over all these possible abstract values yields to the dependency of the expression $x_2 \times 2$ on x_2 w.r.t. SIGN, the dependency is included in the semantics-based abstract PDG by the edge linking node 9 with node q_2.*

However, the execution of statement 9 does not affect at all the sign of x. This "false positive" is due to the fact that the semantic dependency in [20] is defined at expression level. The abstract semantics of the program should says that statement 9 is not relevant for the slicing criteria $\langle 13, y \rangle$. Thus, slicing with criteria $\langle 13, y \rangle$ should have the correct, and more precise, slice shown in Figure 3(b), as the sign of x at line 11 and 12 in the original program is the same as that in the input value.

The point we raise is that the semantics-based (abstract) PDG obtained from the (abstract) semantic dependency [20] can be improved w.r.t. accuracy if, before deriving the dependency at expression level, we compute the semantic relevancy of statements

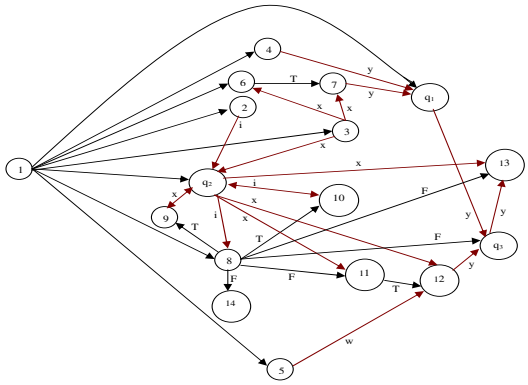
[1] The node in G_{pdg} corresponding to the statement ϕ_i (i^{th} ϕ statement) in P_{ssa} is labeled as q_i.

```
1.   start
2.   i=-2;
3.   x=input;
4.   y=input;
5.   w=input;
6.   if(x ≥ 0)
7.         y = 4x^3;
8.   while(i ≤ 0){
9.         x = x × 2;
10.        i = i + 1;}
11.  if(x ≤ 0)
12.        y = x^2 + 4w mod 2;
13.  print(x,y);
14.  stop
```

(a) Program P

```
1.   start
2.   i1=-2;
3.   x1=input;
4.   y1=input;
5.   w=input;
6.   if(x1 ≥ 0)
7.         y2 = 4 × (x1)^3;
φ1   y3 = f(y1, y2)
     while(
φ2   (x2, i2) = f((x1, i1), (x3, i3))
8.   i2 ≤ 0
     ){
9.         x3 = x2 × 2;
10.        i3 = i2 + 1;}
11.  if(x2 ≤ 0)
12.        y4 = (x2)^2 + 4w mod 2;
φ3   y5 = f(y3, y4);
13.  print(x2, y5);
14.  stop
```

(b) P_{ssa}: SSA form of program P

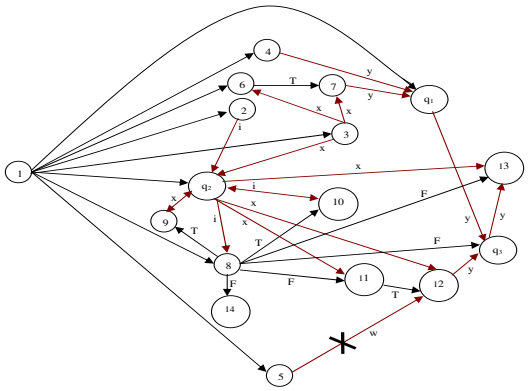


(c) G_{pdg}: PDG of P_{ssa}

Figure 1: The traditional Program Dependency Graph (PDG)

in the program. This way, we can refine the PDG, by eliminating the nodes corresponding to the semantically irrelevant statements from the PDG. In similar way, we can eliminate all nodes corresponding to a conditional block if all the statements in that block are not semantically relevant at all, as illustrated in Section 4. In our example, Figure 3(a) depicts a more precise semantics-based abstract PDG $G^{r,d}_{pdg}$ obtained by computing first the abstract semantic relevancy at statement level which removes the node corresponding to the statement 9, and then by computing the abstract semantic data dependency [20] which removes the data dependency on w at statements 12.

By following [28], we can extend every semantics-based (abstract) PDG obtained so far into the semantics-based (abstract) Dependence Condition Graphs DCG with the annotation $e^b \triangleq \langle e^R, e^A \rangle$ over all the data/control dependence edges e (from e.src to e.tgt): e^R is referred to as Reach Sequence and represents the conditions required to reach e.tgt from e.src, and e^A is referred to as Avoid Sequence and used to avoid re-definitions (in case of control dependence edge, e^A is $\emptyset$). An execution ψ over an abstract domain, is said to satisfy e^b for an data dependence edge e if it satisfies all



(a) G^d_{pdg} : *PDG of* P_{ssa} *after computing Semantic Dependency w.r.t. SIGN* [20]

```
1.    start
2.    i=-2;
3.    x=input;
4.    y=input;
6.    if(x ≥ 0)
7.          y = 4x^3;
8.    while(i ≤ 0){
9.          x = x × 2;
10.         i = i + 1; }
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

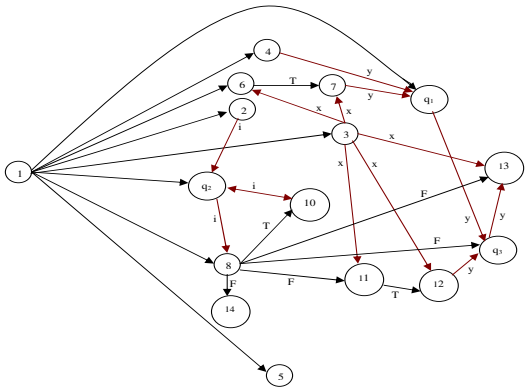
(b) *Slice with criteria* $\langle 13, y\rangle$ *computed from* G^d_{pdg}

Figure 2: Semantics-based abstract PDG and corresponding slice after computing Semantic Dependency w.r.t. SIGN

the conditions in e^R *and at the same time it avoids the conditions represented by* e^A*; this means that the execution* ψ *ensures that the abstract value computed at e.src successfully reaches e.tgt and has an impact on the abstract value of e.tgt. We can extend this to any data dependence PDG path (known as* ϕ*-sequence)* $\eta = e_1 e_2 \ldots e_n$*: given a* ϕ*-sequence* $\eta = e_1 e_2 \ldots e_n$ *and an execution* ψ *over an abstract domain, we say that* ψ *satisfies* η *(written as* $\psi \vdash \eta$*) if the abstract value computed at* e_1*.src reaches to* e_n*.tgt and has an impact on the abstract value of e.tgt in* ψ*.*

Recall the algorithm to compute DCG reported in [28]. For the program in Figure 1 and its semantics-based abstract PDG $G^{r,d}_{pdg}$ *in Figure 3(a), consider the control dependence path* $p \triangleq 1 \rightarrow 8 \xrightarrow{false} 11$ *and the control dependence edge* $c \triangleq 11 \xrightarrow{true} 12$*. We get the reach sequence for c,* $c^R = \{11 \xrightarrow{true} 12\}$ *which means that to execute the statement 12 successfully the condition at 11 must be true. Now there is one data dependence edge:* $d = 3 \xrightarrow{x} 12$ *that has statement 12 as target. For this data dependence edge, we get the reach sequences as* $d^R = \{1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$*. This means that the condition* $1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$ *has to be satisfied so that 3 and 12 both can get executed.*

To compute the avoid sequence for the edge $d_1 = \phi_1 \xrightarrow{y} \phi_3$*, consider two data*



(a) $G^{r,d}_{pdg}$: *PDG of* P_{ssa} *by computing Statement Relevancy first, and then Semantic Dependency w.r.t. SIGN– Observe that node* 9 *does not appear anymore.*

```
1.    start
3.    x=input;
4.    y=input;
6.    if(x ≥ 0)
7.          y = 4x^3;
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

(b) *Slice with criteria* $\langle 13, y\rangle$ *obtained from* $G^{r,d}_{pdg}$

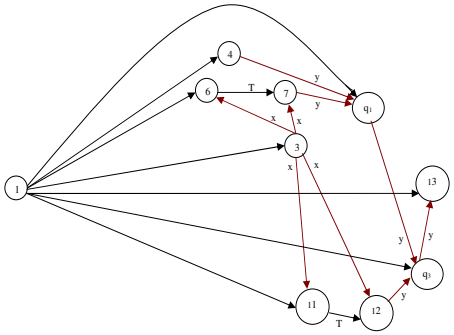
Figure 3: Semantics-based abstract PDG and corresponding slice by computing Statement Relevancy first, and then Semantic Dependency w.r.t. SIGN

dependence edges $d_1 = \phi_1 \xrightarrow{y} \phi_3$ *and* $d_2 = 12 \xrightarrow{y} \phi_3$ *with* ϕ_3 *as target. Following the algorithm of [28], we get* $d^A_1 = (\phi_1 \xrightarrow{y} \phi_3)^A = \{1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$. *This reflects the fact that the "if" condition at 11 must be false in order to guarantee that the definition at* ϕ_1 *is not re-defined at 12 and can reach* ϕ_3. *Similarly, we get* $(4 \xrightarrow{y} \phi_1)^A = \{1 \rightarrow 6 \xrightarrow{true} 7\}$. *Figure 4(a) depicts the DCG annotations over the data dependence edges of* $G^{r,d}_{pdg}$.

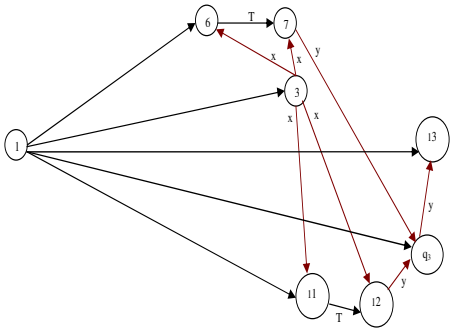
Given the slicing criteria $\langle 13, y\rangle$, *the sub-PDG* $G^{r,d}_s$ *obtained after performing backward slicing technique on* $G^{r,d}_{pdg}$ *is shown in Figure 4(b). Let us consider the PDG path* $\eta = 4 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_3 \xrightarrow{y} 13$ *in* $G^{r,d}_s$ *which is a* ϕ*-sequence [28] and indicates the flow of definition at 4 to 13. Observe that, since the abstract values of x may have any value from the set* $\{+, 0, -\}$, *there are no such execution* ψ *over the abstract domain SIGN which avoids both* $(4 \xrightarrow{y} \phi_1)^A$ *and* $(\phi_1 \xrightarrow{y} \phi_3)^A$ *simultaneously i.e.* $\forall \psi : \psi \nvdash \eta$. *For each execution over the abstract domain of signs, at least one of the conditions among* $6 \xrightarrow{true} 7$ *and* $11 \xrightarrow{true} 12$ *will be satisfied. This means that the definition at 4 is over-written by 7 or 12 and never reaches 13. Since there exists no semantically realizable PDG path between 4 and 13, we can remove the node 4 from* $G^{r,d}_s$ *as depicted in Figure 4(c) and thus, we get the final more precise slice as shown in Figure 4(d).*

d	d^R	d^A
$4 \xrightarrow{y} \phi_1$	$\emptyset$	$1 \rightarrow 6 \xrightarrow{true} 7$
$7 \xrightarrow{y} \phi_1$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 6$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 7$	$1 \rightarrow 6 \xrightarrow{true} 7$	$\emptyset$
$3 \xrightarrow{x} 11$	$1 \rightarrow 8 \xrightarrow{false} 11$	$\emptyset$
$3 \xrightarrow{x} 12$	$1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$	$\emptyset$
$3 \xrightarrow{x} 13$	$1 \rightarrow 8 \xrightarrow{false} 13$	$\emptyset$
$12 \xrightarrow{y} \phi_3$	$\emptyset$	$\emptyset$
$\phi_1 \xrightarrow{y} \phi_3$	$1 \rightarrow 8 \xrightarrow{false} \phi_3$	$1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$
$\phi_3 \xrightarrow{y} 13$	$\emptyset$	$\emptyset$

(a) *The annotations* $\langle d^R, d^A \rangle$ *over the data dependence edges* d *of the PDG* $G^{r,d}_{pdg}$



(b) $G^{r,d}_s$: *sub−PDG obtained after performing backward slicing on* $G^{r,d}_{pdg}$ *w.r.t.* $\langle 13, y \rangle$



(c) $G^{r,d,c}_s$: *sub−PDG obtained from* $G^{r,d}_s$ *after computing Conditional Dependencies*

```
1.    start
3.    x=input;
6.    if(x ≥ 0)
7.          y = 4x^3;
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

(d) *Slice with criteria* $\langle 13, y \rangle$ *computed from* $G^{r,d,c}_s$

Figure 4: Semantics-based abstract sub-PDG and corresponding slice *w.r.t.* $\langle 13, y \rangle$ after computing Conditional Dependencies *w.r.t.* SIGN

3 Abstract Interpretation

Abstract Interpretation, originally introduced by Cousot and Cousot is a well known semantics-based static analysis technique [3, 4, 5, 9]. Its main idea is to relate concrete and abstract semantics where the later are focussing only on some properties of interest. Abstract semantics is obtained from the concrete one by substituting concrete domains of computation and their basic concrete semantic operations with abstract domains and corresponding abstract semantic operations. This can be expressed by means of closure operators.

An (*upper*) *closure* operator on C, or simply a closure, is an operator $\rho : C \to C$ which is monotone, idempotent, and extensive. The upper closure operator is the function that maps the concrete values with their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, the operator $Par : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associates each set of integers with its parity, $Par(\bot) = \bot$, $Par(S) = EVEN = \{n \in \mathbb{Z} \mid n \text{ is even}\}$ if $\forall n \in S$. n is even, $Par(S) = ODD = \{n \in \mathbb{Z} \mid n \text{ is odd}\}$ if $\forall n \in S$. n is odd, and $Par(S) = I\ don't\ know = \mathbb{Z}$ otherwise.

Abstract Semantics: Expressions and Statements

As in [20], we consider the IMP language [30]. The statements of a program P act on a set of constants $\mathbb{C} = const(P)$ and set of variables $VAR = var(P)$. A program variable $x \in VAR$ takes its values from the semantic domain $\mathbb{V} = Z_{\mho}$ where, $\mho$ represents an undefined or uninitialized value and Z is the set of integers. The arithmetic expressions $e \in Aexp$ and boolean expressions $b \in Bexp$ are defined by standard operators on constants and variables. The set of states Σ consists of functions $\sigma : VAR \to \mathbb{V}$ which maps the variables to their values. For the program with k variables $x_1, \ldots, x_k$, the state is denoted by k-tuples: $\sigma = \langle v_1, \ldots, v_k \rangle$, where $v_i \in \mathbb{V}, i = 1, ..., k$ and hence, set of states $\Sigma = (\mathbb{V})^k$. Given a state $\sigma \in \Sigma$, $v \in \mathbb{V}$, and $x \in VAR$: $\sigma[x \leftarrow v]$ denotes a state obtained from σ by replacing its contents in x by v, *i.e.* define

$$\sigma[x \leftarrow v](y) = \begin{cases} v & if\ x = y \\ \sigma(y) & if\ x \neq y \end{cases}$$

The semantic of arithmetic expression $e \in Aexp$ over the state σ is denoted by $E[\![e]\!](\sigma)$ where, the function E is of the type $Aexp \to (\sigma \to \mathbb{V})$. Similarly, $B[\![b]\!](\sigma)$ denotes the semantics of boolean expression $b \in Bexp$ over the state σ of type $Bexp \to (\sigma \to T)$ where T is the set of truth values.

The Semantics of statement s is defined as a partial function on states and is denoted by $S[\![s]\!](\sigma)$ which defines the effect of executing s in σ. The partial semantics $P[\![s]\!]_p(\sigma)$ is obtained by collecting all possible states appearing at program point p inside s when s is executed in σ.

Consider an abstract domain ρ on values. The set of abstract states is denoted by $\Sigma^\rho \triangleq \rho(\wp(\mathbb{V}))^k$. The abstract semantics $E[\![e]\!]^\rho(\epsilon)$ of expression e is defined as the best correct approximation of $E[\![e]\!]$: let $\sigma = \langle v_1, \ldots, v_k \rangle \in \Sigma$ and $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho : E[\![e]\!]^\rho(\epsilon) = \rho(\{E[\![e]\!](u_1, \ldots, u_k) \mid \forall i.\ u_i \in \rho(v_i)\})$.

Similarly, semantics $P[\![s]\!]^\rho_p(\epsilon_0)$ (where $\epsilon_0 = \langle \top, \ldots, \top \rangle$) computes a safe over-approximation of the minimal abstract state $\epsilon'_p \in \Sigma^\rho$ which describes variables

at p:

$$P[\![s]\!]^{\rho}_{p}(\epsilon_0) \triangleq \epsilon_p \geq \epsilon'_p = \rho(\bigcup\{P[\![s]\!]_p(\sigma) \mid \sigma \in \Sigma\})$$

Given a state ϵ, a covering $\{\epsilon_1, \ldots, \epsilon_l\}$ is a set of states such that ϵ describes the same set of concrete states as all the ϵ_i: $\epsilon = \cup_i \epsilon_i$.

Partitions, atoms

Given $\rho \in uco(\wp(Z))$, the induced partition $\Pi(\rho)$ of ρ is the set $\{V_1, \ldots, , V_j\}$, partition of $\mathbb{V}$, characterizing classes of values undistinguishable by ρ: $\forall i.\ \forall x, y \in V_i.\ \rho(x) = \rho(y)$. A domain ρ is partitioning if it is the most concrete among those inducing the same partition: for a partition P, $\rho = \sqcap\{\omega \mid \Pi(\omega) = P\}$. If ρ is partitioning, $\Pi(\rho)$ is the set of atoms of ρ, viewed as a complete lattice, *i.e.*, atoms of partitioning domain are the abstractions of singletons.

4 Semantic Relevancy

Given a program P, we define a new semantics-based (abstract) PDG, constructed in two steps:

(1) First, the semantic relevancy of all the statements of the program P are computed. Corresponding to each semantically relevant statement of the program, we create a node in the PDG. The nodes corresponding to the conditional ($if, if-else$) or repetitive statements (*while*) and their control dependencies are inserted if the corresponding blocks are semantically relevant (*i.e.* if at least one statement in the block is semantically relevant).

(2) Second, this PDG is refined by computing the semantic dependency at expression level [20] for all relevant statements to keep only the semantic data dependencies.

Definition 1 *If $\forall \epsilon \in \Sigma^{\rho}$: $P[\![s]\!]^{\rho}_{p}(\epsilon) = \epsilon$, the statement s is not semantically relevant with respect to the abstract domain ρ.*

In other words, the statement s at program point p is semantically irrelevant if no changes take place in the abstract states ϵ occurring at p, when s is executed over ϵ. The statements which do not contribute to any change in the states occurring at that program points are considered semantically irrelevant.

Note that atomicity of the abstract value for each variable in the abstract state ε *w.r.t.* property ρ is one of the crucial requirements during computation of ρ-relevancy of the statements. These atomic abstract values are obtained from induced partitioning. The following example shows how to compute the semantic relevancy for the statements by using covering techniques.

Example 2 *Consider the abstract domain of parity PAR and an abstract state $\varepsilon = \langle even, odd, \top \rangle$ for the variables $x, y, z \in dom(\varepsilon)$. The induced partition for the domain PAR is $\Pi(PAR) = \{even, odd\}$. Since $\top$ is not an atomic state for the domain PAR, we can instead consider a covering $\{\varepsilon_1, \varepsilon_2\}$ for the state ε, where $\varepsilon_1 = \langle even, odd, even \rangle$ and $\varepsilon_2 = \langle even, odd, odd \rangle$. Hence, the relevancy for the statement s at program point p (if ε occurs at p) is computed over ε_1 and ε_2. Observe that, the elements in the cover contains atomic abstract values (atoms, in other word) for the variables.*

Consider the example shown in Figure 1. One possible abstract state at program points 9 and 10 w.r.t. SIGN is $\varepsilon = \langle -, \top, \top, \top \rangle$ *where* $dom(\varepsilon) = \langle i, x, y, w \rangle$*. Since the values for* x, y, w *are provided by the user, it can be any value from the set* $\{+, 0, -\}$ *and is denoted by the top element* $\top$ *of the lattice for SIGN. When we compute the semantic relevancy of statements 9 and 10, the execution over the abstract state* ε *can not reveal the fact of semantic relevancy because of the overapproximated state and lack of preciseness. Therefore, we compute the semantic relevancy of 9 and 10 over the covering of* ε *i.e.* $\{\langle -, -, -, - \rangle, \langle -, 0, -, - \rangle, \langle -, +, -, - \rangle, \ldots, \langle -, +, +, + \rangle\}$*. Computing over these covering states, we can easily conclude the semantic (ir)relevancy of 9 and 10.*

Treating Conditional Statements

Example 1 illustrated how to obtain a more precise PDG by eliminating the nodes corresponding to the semantically irrelevant statements in the program. In this section we describe how we can disregard any conditional dependency if all statements in that conditional block are not semantically relevant at all.

Example 3 *Consider the program of Figure 5(a) and the property PARITY. The traditional PDG of it, is shown in Figure 5(b). At program point 2 and 3, the parity of x and y are odd and even respectively. These properties of x and y remains unchanged during the while loop. Statements 5 and 6, therefore, are not semantically relevant w.r.t. PARITY. Thus, we can disregard the nodes corresponding to these statements yielding to the more precise PDG shown in Figure 5(c). Note that the node corresponding to the repetitive statement "while" (node 4) does not appear in Figure 5(c) because there is no semantically relevant statement in the while block.*

We follow the same approach in case of "if" conditional block: the node corresponding to the "if" conditional statement is disregarded from the PDG if the corresponding block has no semantically relevant statement inside.

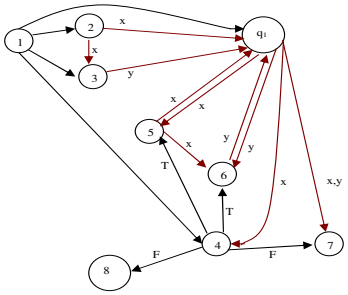
We now discuss about the "$if-else$" conditional block appears in the program. We may distinguish four cases:

1. All statements in "if" block are semantically irrelevant, whereas some/all statements in "*else*" block are semantically relevant *w.r.t.* property ρ.
2. Some/all statements in "if" block are semantically relevant, whereas all statements in "*else*" block are semantically irrelevant *w.r.t.* property ρ.
3. Some/all statements in both "if" and "*else*" blocks are semantically relevant *w.r.t.* property ρ.
4. All statements in both "if" and "*else*" blocks are semantically irrelevant *w.r.t.* property ρ.

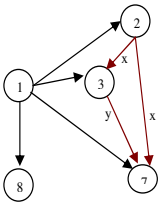
In case 4, we can disregard all the nodes corresponding to the complete "$if-else$" block including the conditional statement from the PDG as all the statements in both "if" and "*else*" blocks are semantically irrelevant. In case 1, we can replace all nodes corresponding to the all statements in "if" block with a single node that corresponds to the statement *skip* with no data flow, whereas in case 2, we can disregard all nodes corresponding to the complete "*else*" block. Observe that, when some or all statements are semantically relevant in either blocks,

1. Start
2. $x_1 = 1;$
3. $y_1 = 2 \times x_1;$
   while(
ϕ_1 $(x_2, y_2) = f((x_1, y_1), (x_3, y_3))$
4. $x_2 < 20$){
5. $x_3 = x_2 + 2;$
6. $y_3 = y_2 + 2 \times x_3^2;$}
7. *print*(x_2, y_2);
8. Stop

(a) Program P



(b) Traditional PDG of the Program P



(c) Semantics-based abstract PDG of P after computing semantic relevancy *w.r.t. PARITY*

Figure 5: Treating "*while*" conditional block

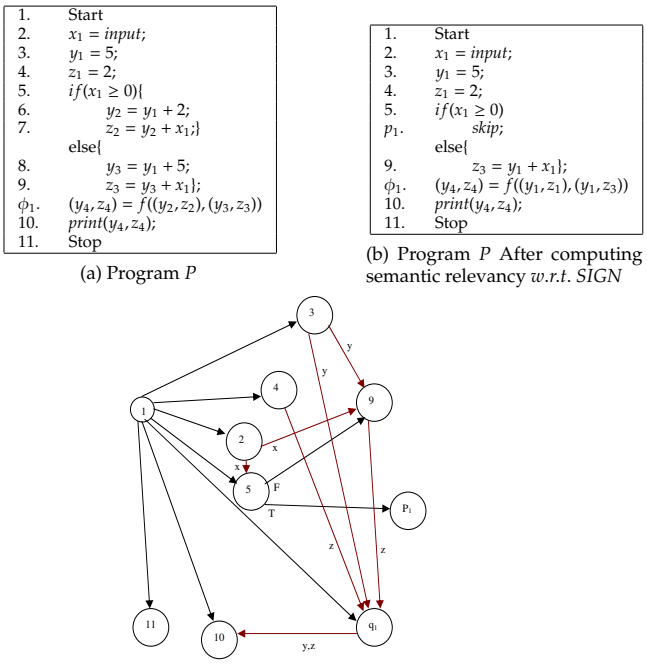
only the nodes corresponding to those semantically irrelevant statements (if exists) will be disregarded from the PDG as in case 3. Lets illustrate with an example shown in Example 4.

Example 4 *Consider the program of Figure 6(a). Observe that the statements 6 and 7 in the "if" conditional block are semantically irrelevant w.r.t. the sign property of the variables. The execution of the statements 5 and 6 over all possible abstract states appearing at those program points do not change the sign of y and z. But the "else" block is semantically relevant as it contains semantically relevant statement 9 w.r.t. the sign property. According to case 1, we can't remove the "if" block. Hence, we replace these semantically irrelevant statements in the "if" block with the statement "skip". The corresponding form of the program and semantics-based abstract PDG are shown in Figure 6(b) and 6(c) respectively.*

In [20], the problem related to the control dependency is not addressed: even if they consider (abstract) semantic dependencies, they still could not be able to reach the most precise slice for a given criterion. For example, consider the following example:

4. ....
5. *if* $((y + 2x \ mod \ 2) == 0)$ *then*
6. w=5;
7. else w=5;
8. ....

Here the abstract semantic dependency says that the condition of "*if*" statement is only dependent on y. But it does not say anything about the dependency



(a) Program P

(b) Program P After computing semantic relevancy *w.r.t. SIGN*

(c) *Semantics−based Abstract PDG of P after computing Semantic Relevancy w.r.t. SIGN*

Figure 6: Treatment of "if-else" conditional block

between w and y. Observe that although w is invariant *w.r.t.* the evaluation of the guard, this is not captured by [20].

The semantic relevancy over statement level can resolve this issue of independency. Let us denote the complete "$if - else$" block by s. The semantics of s says that $\forall \sigma_1, \sigma_2 \in \Sigma$ appearing at program point 5, $P[\![s]\!](\sigma_1) = \sigma_1'$ and $P[\![s]\!](\sigma_2) = \sigma_2'$ implies $\sigma_1' = \sigma_2'$. It means that there is no control of the "$if - else$" over the resultant state which is invariant. So we can replace the complete conditional block by the single statement $w = 5$.

Refined Abstract PDGs

We are now in position to formalize the algorithm to construct semantics-based (abstract) PDG $G^{r,d}_{pdg}$ of a program P, as shown in Figure 7. We use the notation s_i to denote i^{th} statement at program point p_i in the program P. The notation ε_{ij} represents the j^{th} abstract state appearing at program point p_i *w.r.t.* property ρ. The input of the algorithm is the program P and output is the semantics-based (abstract) PDG.

Step 3 computes the semantic relevancy *w.r.t.* property ρ for all the statements of the program and at step 6 inserts nodes corresponding to the semantically relevant statements except the conditional (if, $if - else$) and repetitive (*while*) statements. Steps 9, 10, 11 and 12 deal with how to insert a node into

Algorithm 1: REFINE-PDG

Input: Program P

Output: Semantics-based (Abstract) PDG $G^{r,d}_{pdg}$

1. FOR each statement s_i DO
2. FOR all $\varepsilon_{ij} \in \Sigma^{\rho}$ DO
3. Compute Semantic relevancy of s_i;
4. END FOR
5. IF s_i is semantically relevant and is not conditional/repetitive statement THEN
6. Insert the node corresponding to s_i into the PDG;
7. END IF
8. END FOR
9. FOR each "$if-else$" conditional statement DO
10. Case 1: "if" block is semantically irrelevant but has semantically relevant "*else*" block: insert a node into the PDG corresponding to the statement "*skip*" which replaces all statements in the "if" block, and insert a node into the PDG corresponding to the "$if-else$" conditional statement, and obtain the control dependencies;
11. Case 2: "if" block is semantically relevant but has semantically irrelevant "*else*" block: insert a node into the PDG corresponding to the "if"conditional statement only and disregard the "*else*" block completely and obtain the control dependencies;
12. Case 3: Both "$if-else$" blocks are semantically relevant: insert a node into the PDG corresponding to the "$if-else$" conditional statement and obtain the control dependencies;
13. END FOR
14. FOR each "*while*" repetitive or "if" conditional statement DO
15. Case 1: "*while*" block is semantically relevant: insert a node into the PDG corresponding to the repetitive statement "*while*" and obtain the control dependencies;
16. Case 2: "if" block is semantically relevant: insert a node into the PDG corresponding to the "if" conditional statement and obtain the control dependencies;
17. END FOR
18. Apply expression level semantic dependency computation to obtain semantic data dependencies;

Figure 7: Algorithm to generate precise semantics-based (abstract) PDG

the PDG corresponding to the "$if-else$" conditional statement based on the semantic relevancy of the "if" and "*else*" blocks and follow the rules as discussed before. Steps 14, 15 and 16, similarly, deal with how to insert a node into the PDG corresponding to the "if" conditional statement and "*while*" repetitive statement based on the semantic relevancy of the corresponding blocks. Finally, semantic data dependencies are included into the PDG at step 18. The idea to obtain an semantics-based (abstract) PDG is to unfold the program into an equivalent program where only statements that have an impact *w.r.t.* the abstract domain are combined with the semantic data flow *w.r.t.* the same domain.

5 Dependence Condition Graphs (DCGs)

In this section, we extend the semantics-based (abstract) PDG into Dependence Condition Graph DCG [28]. A DCG is built from a PDG by annotating each edge e of the PDG with information e^b whose semantic interpretation encodes the condition for which the dependence represented by that edge actually arise in a program execution. The annotation e^b on any edge e (from *src* to $e.tgt$) is a pair $\langle e^R, e^A \rangle$. The first component e^R is referred to as Reach Sequence, and represents the condition that should be true for an execution to ensure that the target $e.tgt$ of e is executed once the source $e.src$ of e is executed, if e is a control edge, or that $e.tgt$ is reached from $e.src$ if e is a data edge. The component e^A is referred to as Avoid Sequence which is only relevant for data edges (for control edges it is $\emptyset$), and captures the possible condition under which the assignment at $e.src$ can be overwritten before it reaches to $e.tgt$.

Sukumaran et al. [28] also described the semantics of DCG in terms of execution semantics of the program over concrete domain. This concrete semantics of the DCG can easily be lifted to the abstract domain with respect to the property of interest. We skip the details of the abstract semantics of the DCG for brevity. The (abstract) semantics of the DCG says about the condition under which any execution ψ over an abstract domain *w.r.t.* property ρ satisfies any PDG path $\eta = e_1 e_2 \dots e_n$, $n \geq 1$ (written as $\psi \vdash \eta$). For any ϕ-sequence η [28], any execution ψ over an abstract domain will satisfy η if the abstract value computed at $e_1.src$ reaches to $e_n.tgt$ and has an impact on it in ψ.

Theorem 1 *For a ϕ-sequence $\eta = e_1 e_2 \dots e_n$ and execution ψ over an abstract domain, we say that $\psi \vdash \eta$ if the abstract value computed at $e_1.src$ reaches to $e_n.tgt$ and has impact on the abstract value of $e.tgt$ in ψ.*

Consider a semantics-based (abstract) PDG $G^{r,d}_{pdg}$ with the DCG annotations over the data/control edges. If we are given a slicing criteria $\langle p, v \rangle$, the corresponding sub-PDG $G^{r,d}_{s}$ can be obtained by applying the PDG-based slicing techniques [23] on $G^{r,d}_{pdg}$. This sub-PDG $G^{r,d}_{s}$ can be made more precise by removing the node x from the sub-PDG if all the existing PDG paths η between x and p in $G^{r,d}_{s}$ are semantically unrealizable under the (abstract) semantics of the DCG annotations *i.e.* $\forall \eta$ and $\forall \psi$: $\psi \nvdash \eta$. Thus, the corresponding node x will not appear in the slice. In Figure 8, we formalize the algorithm to compute the precise slice based on DCG where the input is the semantics-based (abstract) PDG $G^{r,d}_{pdg}$ with all the DCG annotations and the slicing criteria $\langle p, v \rangle$.

The results on the dependency graph discussed so far have an impact on the different forms of static slicing: backward slicing, forward slicing, and chopping. The backward slice with respect to variable v at program point p consists of the program points that affect v. Forward slicing is the dual of backward slicing. The forward slice with respect to variable v at program point p consists of the program points that are affected by v. Chopping is a combination of backward slicing and forward slicing. A slicing criterion for chopping is represented by a pair $\langle s, t \rangle$ where s and t denote the source and sink respectively. In particular, chopping of a program *w.r.t.* $\langle s, t \rangle$ identifies a subset

Algorithm 2: REFINE-subPDG

Input: Semantics-based (Abstract) PDG $G_{pdg}^{r,d}$ with DCG annotations and slicing criteria $\langle p, v \rangle$
Output: (Abstract) Slice

1. Apply the PDG-based slicing technique on $G_{pdg}^{r,d}$ *w.r.t.* $\langle p, v \rangle$ and obtain the sub-PDG $G_s^{r,d}$;

2. Perform step 2(a) for all nodes x in $G_s^{r,d}$ and obtain precise sub-PDG $G_s^{r,d,c}$:

2(a). FOR all PDG path $\eta = e_1 e_2 \dots e_n$ between x and p: if $\forall\psi$, $\forall\eta$: $\psi \nvdash \eta$, remove the corresponding node x from $G_s^{r,d}$;

3. Compute the slice based on $G_s^{r,d,c}$ *w.r.t.* $\langle p, v \rangle$;

Figure 8: Slicing based on DCG

of its statements that account for all influences of source s on sink t.

The slicing based on dependency graph is slightly restrictive in the sense that the dependency graph permits slicing of a program with respect to program point p and a variable v that is defined or used at p, rather than *w.r.t.* arbitrary variable at p. PDG-based backward program slicing is performed by walking the graph backwards from the node of interset in linear time [23]. The walk terminates at either entry node or already visited node. Thus for a vertex n of the PDG, the slice *w.r.t.* the variable v at n is a graph containing all vertices that can reach directly or indirectly to n and affect the value of v via flow or control edges.

In case of forward slicing technique based on PDG representation, similarly, we traverse the graph in forward direction rather than backward from the node of interest. We can use the standard notion of *chop* of a program with respect to two nodes s and t in slicing technique [14, 25]: $chop(s,t)$ is defined as the set of inter-procedurally valid PDG paths from s to t where s, t are real program nodes, in contrast of ϕ nodes in SSA form of the program. We define as follows [28]: $AC(s,t)$ is defined to be true if there exists at least one execution ψ over abstract domain that satisfies a valid PDG path η between s and t *i.e.* $AC(s,t) \triangleq \exists\psi : AC(s,t,\psi)$ and $AC(s,t,\psi) \triangleq \exists\eta \in chop(s,t) : \psi \vdash \eta$. The $\neg AC(s,t)$ implies $\forall\psi$ and $\forall\eta \in chop(s,t) : \psi \nvdash \eta$ which indicates that $chop(s,t)$ is empty.

6 Conclusion

The combination of results on the refinement of dependency graphs with static analysis techniques discussed in this paper may give rise to further interesting applications to enhance the accuracy of the static analysis and for accelerating the convergence of the fixed-point computation. This is the topic of our ongoing research.

References

[1] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings. of International Conference on Software Maintenance*, pages 124–133. IEEE Computer Society Washington, DC, USA, 1995.

[2] A. Cortesi and S. Bhattacharya. A framework for property-driven program slicing. In *Proceedings of the 1st International Conference on Computer, Communication, Control and Information Technology*, pages 118–122, Kolkata, India, 2009. Macmillan Publishers India Ltd.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, 1977. ACM Press.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press.

[5] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the 29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 178–190, Portland, OR USA, January 16–18 2002. ACM Press.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[7] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[8] R. Gerber and S. Hong. Slicing real-time programs for enhanced schedulability. *ACM Trans on Programming Languages and Systems*, 19(3):525–555, May 1997.

[9] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM (JACM)*, 47(2):361–416, March 2000.

[10] Rajeev Gopal. Dynamic program slicing based on dependence relations. In *Proceedings. Conference on Software Maintenance*, pages 191–200, Sorrento, Italy, Oct 1991.

[11] D. Goswami and Rajib Mall. An efficient method for computing dynamic program slices. *Information Processing Letters*, 81(2):111–117, January 2002.

[12] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of Conference on Software Maintenance*, pages 299–308, Orlando, FL, USA, Nov 1992. IEEE Computer Society.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Progr. Lang. Systems*, 12(1):26–60, January 1990.

[14] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. *ACM SIGSOFT Software Engineering Notes*, 19(5):2–10, December 1994.

[15] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[16] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. In *5th International Workshop on Program Comprehension (WPC '97)*, pages 80–89, 1997.

[17] Jens Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7):35 – 42, July 1998.

[18] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, Williamsburg, Virginia, 1981. ACM Press.

[19] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[20] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 125–134, San Francisco, California, USA, 2008. ACM Press.

[21] Markus Muller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 647 – 656, Hersonissos, Greece, 2001. ACM press.

[22] G. B. Mund and Rajib Mall. An efficient interprocedural dynamic slicing method. *The Journal of Systems and Software*, 79(6):791–806, June 2006.

[23] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Softw. Eng. Symp. on Practical Softw. Dev. Environments*, pages 177–184, New York, USA, 1984. ACM Press.

[24] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[25] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proc 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, June 1995.

[26] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5-6):779–804, 1991.

[27] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proc. 21st International Conf. on Software Engineering*, May 1999.

[28] S. Sukumarana, A. Sreenivasb, and R. Metta. The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems & Structures*, 36(96–121):577–581, April 2010.

[29] Mark Weiser. Program slicing. *IEEE Trans. on soft. Eng.*, SE-10(4):352–357, July 1984.

[30] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

Faster ambiguity detection by grammar filtering

H. J. S. Basten and J. J. Vinju

Centrum Wiskunde & Informatica
P.O. Box 94079
NL-1090 GB Amsterdam, The Netherlands

Abstract

Real programming languages are often defined using ambiguous context-free grammars. Some ambiguity is intentional while other ambiguity is accidental. A good grammar development environment should therefore contain a static ambiguity checker to help the grammar engineer.

Ambiguity of context-free grammars is an undecidable property. Nevertheless, various imperfect ambiguity checkers exist. Exhaustive methods are accurate, but suffer from non-termination. Termination is guaranteed by approximative methods, at the expense of accuracy.

In this paper we combine an approximative method with an exhaustive method. We present an extension to the Noncanonical Unambiguity Test that identifies production rules that do not contribute to the ambiguity of a grammar and show how this information can be used to significantly reduce the search space of exhaustive methods. Our experimental evaluation on a number of real world grammars shows orders of magnitude gains in efficiency in some cases and negligible losses of efficiency in others.

1 Introduction

Real programming languages are often defined using ambiguous context-free grammars. Some ambiguities are intentional, while others are accidental. It is therefore important to know all of them, but this can be a very cumbersome job if done by hand. Automated ambiguity checkers are therefore very valuable tools in the grammar development process, even though the ambiguity problem is undecidable in general.

In [3] we compared the practical usability of several ambiguity detection methods on a series of grammars[1]. The exhaustive derivation generator AMBER [10] was the most practical in finding ambiguities for real programming languages, despite its possible nontermination. The main reasons for this are its accurate reports (Figure 1) that contain examples of ambiguous strings, and its impressive efficiency. It took about 7 minutes to generate all the strings of

[1] In the current paper we also use CFG ANALYZER [1] which was not included in [3].

```
GRAMMAR DEBUG INFORMATION
Grammar ambiguity detected. (disjunctive)
Two different ‘‘type_literals’’ derivation trees for the same phrase.

TREE 1
------
type_literals alternative at line 787, col 9 of grammar {
  VOID_TK
  DOT_TK
  CLASS_TK
}

TREE 2
------
type_literals alternative at line 785, col 16 of grammar {
  primitive_type alternative at line 31, col 9 of grammar {
    VOID_TK
  }
  DOT_TK
  CLASS_TK
}
```

Figure 1: Excerpt from an ambiguity report by AMBER on a Java grammar.

length 10 for Java. Nevertheless, this method does not terminate in case of unambiguity and has exponential performance. For example, we were not able to analyze Java beyond a sentence length of 12 within 15 hours.

Another good competitor was Schmitz's Noncanonical Unambiguity Test [8] (NU TEST). This approximative method always terminates and can provide relatively accurate results in little time. The method can be tuned to trade accuracy for performance. Its memory usage grows to unpractical levels much faster than its running time. For example, with the best available accuracy, it took more than 3Gb to fully analyze Java. A downside is that its reports can be hard to understand due to their abstractness (Figure 2).

In this paper we propose to combine these two methods. We show how the NU TEST can be extended to identify parts of a grammar that do not contribute to any ambiguity. This information can be used to limit a grammar to only the part that is potentially ambiguous. The smaller grammar is then fed to the exhaustive AMBER and CFG ANALYZER [1] methods to finally obtain a precise ambiguity report.

The goal of our approach is ambiguity detection that scales to real grammars and real sentence lengths, providing accurate ambiguity reports. Our new filtering method leads to significant decreases in running time for AMBER and CFG ANALYZER, which is a good step towards this goal.

Related Work Another approximative ambiguity detection method is the Ambiguity Checking with Language Approximation framework [4] by Brabrand, Giegerich and Møller. The framework makes use of a characterization of ambi-

```
5 potential ambiguities with LR(1) precision detected:
    (method_header -> modifiers type method_declarator throws . ,
     method_header -> modifiers VOID_TK method_declarator throws . )
    (method_header -> type method_declarator throws . ,
     method_header -> VOID_TK method_declarator throws . )
    (method_header -> type method_declarator throws . ,
     method_header -> modifiers VOID_TK method_declarator throws . )
    (method_header -> VOID_TK method_declarator throws . ,
     method_header -> modifiers type method_declarator throws . )
    (type_literals -> primitive_type DOT_TK CLASS_TK . ,
     type_literals -> VOID_TK DOT_TK CLASS_TK . )
```

Figure 2: Excerpt from an ambiguity report by NU TEST on a Java grammar.

guity into horizontal and vertical ambiguity to test whether a certain production rule can derive ambiguous strings. This method might be extended in a comparable fashion as we propose to extend the NU TEST here.

Other exhaustive ambiguity detection methods are [5] and [6]. These can benefit from our grammar filtering similarly to AMBER and CFG ANALYZER.

Outline In Section 2 we explain the NU TEST, how to extend it to identify harmless productions, and how to construct a filtered grammar. Section 3 contains an experimental validation of our method. We summarize our results in Section 4.

2 Filtering Unambiguous Productions

In this section we explain how to filter productions from a grammar that do not contribute to any ambiguity. We first briefly recall the basic NU TEST algorithm before we explain how to extend it to identify harmless productions. This section ends by explaining how to construct a valid filtered grammar that can be fed to any exhaustive ambiguity checker. A more detailed description of our method, together with proofs of correctness, can be found in [2].

2.1 Preliminaries

A *grammar* G is a four-tuple (N, T, P, S) where N is the set of nonterminals, T the set of terminals, P the set of productions over $N \times (N \cup T)^*$, and S is the start symbol. V is defined as $N \cup T$. We use $A, B, C, \ldots$ to denote nonterminals, and $\alpha, \beta, \gamma, \ldots$ to denote sentential forms: strings over V^*. The relation $\Longrightarrow$ denotes derivation of sentential forms. We say $\alpha B \gamma$ directly derives $\alpha\beta\gamma$, written as $\alpha B\gamma \Longrightarrow \alpha\beta\gamma$ if a production rule $B \to \beta$ exists in P. The symbol $\Longrightarrow^*$ means "derives in zero or more steps". An *item* indicates a position in a production rule using a dot, for instance as $S \to A \bullet BC$.

2.2 The Noncanonical Unambiguity Test

The Noncanonical Unambiguity test [8] by Schmitz is an approximated search for two different parse trees of the same string. It uses a *bracketed grammar*, which is obtained from an input grammar by adding a unique terminal symbol to the beginning and end of each production. The language of a bracketed grammar represents all parse trees of the original grammar.

From the bracketed grammar a *position graph* is constructed, in which the nodes are positions in strings generated by this grammar. The edges represent evaluation steps of the bracketed grammar: there are *derivation*, *reduction*, and *shift* edges. Derivations and reductions correspond to entries and exits of a production rule, while shifts correspond to steps inside a single production rule over terminal and nonterminal symbols.

A path through this position graph describes a parse tree of the original grammar. Therefore, the existence of two different paths of which the labels of shifts edges form the same string indicates the ambiguity of the grammar. So, position graphs help to point out ambiguity in a straightforward manner, but they are usually infinitely large. To obtain analyzable graphs Schmitz describes the use of equivalence relations on the nodes. These should induce conservative approximations of the unambiguity property of the grammar. If they report ambiguity we know that the input grammar is *potentially ambiguous*, otherwise we know for sure that it is unambiguous.

2.3 LR(0) Approximation

An equivalence relation that normally yields an approximated graph of analyzable size is the "item$_0$" relation [8]. We use item$_0$ here to explain the NU TEST for simplicity's sake, ignoring the intricacies of other equivalence relations.

The item$_0$ position graph of a grammar closely resembles its LR(0) parse automaton [7]. The nodes are labeled with the LR(0) items of the grammar and the edges correspond to actions. Every node with the dot at the beginning of a production of S is a *start node*, and every item with the dot at the end of a production of S is an *end node*. There are three types of transitions:

- Shift transitions, of form $A \to \alpha \bullet X\beta \overset{X}{\longmapsto} A \to \alpha X \bullet \beta$
- Derivation transitions, of form $A \to \alpha \bullet B\gamma \overset{d_i}{\longmapsto} B \to \bullet\beta$, where i is the number of the production $B \to \beta$.
- Reduction transitions, of form $B \to \beta \bullet \overset{r_i}{\longmapsto} A \to \alpha B \bullet \gamma$, where i is the number of the production $B \to \beta$.

The derivation and shift transitions are similar to those in an LR(0) automaton, but the reductions are different. The item$_0$ graph has reduction edges to every item that has the dot after the reduced non-terminal, while an LR(0) automaton jumps to a different state depending on the symbol that is at the top of the parse stack. As a result, a certain path through an item$_0$ graph with a d_i transition from $A \to \alpha \bullet B\gamma$ does not necessarily match an r_i transition to $A \to \alpha B \bullet \gamma$. The

language characterized by an item_0 position graph is thus a super-language of the language of parse trees of the original grammar.

2.4 Finding ambiguity in an item_0 position graph

To find possible ambiguity, we can traverse the item_0 graph using two cursors simultaneously. If we can traverse the graph while the two cursors use different paths, but construct the same string of shifted tokens, we have identified possible ambiguity.

An efficient representation of all such simultaneous traversals is a *position pair graph* (PPG). The nodes of this graph represent the pair of cursors into the original item_0 graph. The edges represent steps made by the two cursors, but not all transitions are allowed. An edge exists for either an individual derivation or reduction transitions by one of the cursors, or for a simultaneous shift transition of the exact same symbol by both cursors. An *end pair* is a pair pointing to two end nodes in the original graph.

Any path in a PPG that reaches an end pair has shifted the same symbols and describes two parse trees of the same string. We call such a path an *ambiguous path pair*, if the two paths it represents are not identical. The existence of ambiguous path pairs is indicated by a *join point*: a reduce transition from a pair with different items to a pair with identical items. Ergo, in the item_0 case we can efficiently detect (possible) ambiguity by constructing a PPG and looking for join points.

To optimize the process of generating PPGs we can omit certain nodes and edges. In particular, paths that derive non-terminals that have no item_0 conflicts at all can safely be replaced by a shift over the non-terminal. We call this process *terminalization* of a non-terminal. Such optimizations can significantly improve the size of the graph.

2.5 Filtering Harmless Production Rules

The NU TEST stops after a PPG is constructed and the ambiguous path pairs are reported to the user. In our approach we also use the PPG to identify production rules that certainly do not contribute to the ambiguity of the grammar. We call these *harmless* production rules.

The main idea is that a production rule is harmless if its items are not used in any ambiguous path pair. The set of ambiguous path pairs describes an over-approximation of the set of all parse trees of ambiguous strings. So, if a production is not used by this set it is certainly not used by any real parse tree of an ambiguous string.

Note that a production like that may still be used in a parse tree of an ambiguous sentence, but then it does not cause ambiguity in itself. In this case the ambiguity already exists in a sentential form in which the non-terminal of the production is not derived yet.

We use knowledge about harmless rules to filter the PPG and to eventually produce a filtered grammar containing only rules that potentially contribute to ambiguity. This is an outline of our algorithm:

1. Remove pairs not used on any ambiguous path pair.
2. Remove noticeably invalid (over-approximated) paths, until a fixed-point:
   (a) Remove incompletely used productions.
   (b) Remove unmatched derivation and reduction steps.
   (c) Prune dead ends and unreachable sub-graphs.
3. Collect the potentially harmful production rules that are left over.

Step 1 and Step 3 are the essential steps, but there is room for optimization. Because the item_0 graph is an over-approximation, collecting the harmful productions also takes parse trees into account that are invalid for the original grammar. There are at least two situations in which these can be easily identified and removed.

Incompletely Used Productions Consider that any path in the item_0 graph that describes a valid parse tree of the original grammar must exercise all items of a production. So, if any item for a production is not used by any ambiguous path pair, then the entire production never causes ambiguous parse trees for a sentence for the original grammar.

Note that due to over-approximation, other items of the identified production may still be used in other valid paths in the item_0 graph, but these paths will not be possible in the unapproximated position graph since they would combine items from different productions.

Once an incompletely used production is identified, all pairs that contain one of its items can be safely removed from the pair graph and new dead ends and unreachable sub-graphs can be pruned. This removes over-approximated invalid paths from the graph.

Unmatched derivations and reductions Furthermore, next to nodes we can also remove certain derivation and reduction edges from the PPG. Consider that any path in the item_0 graph that describes a valid parse tree of the original grammar must both derive and reduce every production that it uses. More specifically, if a d_i transition is followed from $A \to \alpha \bullet B\gamma$ to $B \to \bullet\beta$, the matching r_i transition from $B \to \beta\bullet$ to $A \to \alpha B \bullet \gamma$ must also be used, and vice versa. Therefore, if one of the two is used in the PPG, but the other is not, it can be safely removed, and the PPG can be pruned again.

The process of removing items and transitions can be repeated until no more invalid paths can be found this way. After that the remaining PPG uses only potentially harmful productions. We can gather them by simply collecting the productions from all items used in the graph. Note that the item_0 graph remains

an over-approximation, so we might collect productions that are actually harmless. Whether the reduction of the grammar will actually result in performance gains for exhaustive methods we investigate in Section 3.

2.6 Grammar Reconstruction

From applying the previous filtering process we are left with the set of productions that potentially lead to ambiguity. We want to use this set of productions as input to an exhaustive ambiguity detection method such as CFG ANALYZER or AMBER in order to get precise reports and clear example sentences. Note that the set of potentially ambiguous productions may be empty, in which case this step can be omitted completely.

The filtered set of productions can represent an incomplete grammar for two reasons. Firstly, non-terminals from the top[2] of the grammar may have been filtered. Secondly, non-terminals might not have any productions left, but they could still occur in productions of other non-terminals (they've been terminalized). To restore the reachability and productivity properties of the grammar, a new start symbol, new terminals, non-terminals, and production rules will have to be introduced.

Let us use P_h to denote the set of potentially harmful productions of a grammar. From P_h we can create a new grammar G' by constructing:

1. The set of defined non-terminals of P_h:
   $N_\mathrm{def} = \{A | A \to \alpha \in P_\mathrm{h}\}$.
2. The used but undefined non-terminals of P_h:
   $N_\mathrm{undef} = \{B | A \to \alpha B \beta \in P_\mathrm{h}\} \backslash N_\mathrm{def}$.
3. The unproductive non-terminals:
   $N_\mathrm{unpr} = \{A | A \in N_\mathrm{def}, \neg\exists u \in T^* \colon A \Longrightarrow^* u\}$.
4. The start symbols of P_h:
   $S_\mathrm{h} = \{A | A \in N_\mathrm{def}, \neg\exists (B \to \beta A \gamma) \in P_\mathrm{h})\}$
5. New productions to define a new start-symbol S':
   $P'_S = \{S' \to b_A A e_A |\ A \in S_\mathrm{h}$, b_A and e_A are fresh terminal symbols$\}$.
6. Productions to complete the unproductive and undefined non-terminals:
   $P' = P_\mathrm{h} \cup P'_S \cup \{A \to t_A \mid A \in N_\mathrm{undef} \cup N_\mathrm{unpr}, t_A$ is a fresh terminal$\}$.
7. The new set of terminal symbols:
   $T' = \{a | (A \to \beta a \gamma) \in P'\}$.
8. Finally, the new grammar:
   $G' = (N_\mathrm{def} \cup \{S'\}, T', P', S')$

These steps do not introduce new ambiguities because the production rules for S' we introduce have only one non-terminal and they start with unique terminal symbols. Furthermore the introduced terminals for undefined and unproductive non-terminals are unique.

[2]The top of a grammar are the non-terminals that are injected directly into the start symbol.

		Rules filtered				Time				Memory (Mb)			
Grammar	**Rules**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**
SQL.0	79	79	79	79	79	0.4s	0.4s	1.0s	3.1s	16	16	49	54
SQL.1	79	65	65	65	65	0.5s	0.4s	1.4s	3.9s	17	16	51	56
SQL.2	80	47	47	47	47	1.1s	1.1s	1.9s	6.0s	34	32	58	74
SQL.3	80	54	54	54	54	0.6s	0.5s	1.3s	3.9s	18	17	50	56
SQL.4	80	71	71	71	74	0.4s	0.4s	1.1s	3.1s	17	17	51	45
SQL.5	80	68	68	68	72	0.5s	0.4s	1.2s	3.9s	17	16	53	54
Pascal.0	176	21	30	176	176	2.4s	2.2s	2.4s	15.1s	50	42	160	181
Pascal.1	177	21	25	25	104	2.4s	2.4s	5.9s	40.3s	48	49	162	297
Pascal.2	177	21	25	25	104	2.3s	2.4s	5.8s	46.9s	52	51	159	325
Pascal.3	177	21	30	30	144	2.5s	2.2s	5.0s	20.7s	52	47	160	248
Pascal.4	177	20	24	24	103	2.4s	2.3s	5.9s	42.5s	50	50	163	294
Pascal.5	177	21	25	25	103	2.4s	2.3s	5.8s	32.8s	52	49	159	326
C.0	212	41	44	212	212	4.2s	3.9s	22.7s	-	88	83	248	-
C.1	213	41	44	44	44	4.3s	3.7s	2m08s	1h48m	100	80	465	3078
C.2	213	41	44	44	44	4.3s	3.9s	1m56s	57m49s	101	80	446	3235
C.3	213	40	43	43	43	4.2s	4.1s	2m11s	1h55m	87	81	435	2932
C.4	213	41	44	44	44	4.2s	3.9s	2m14s	1h43m	87	80	445	2996
C.5	213	40	43	43	43	4.3s	3.9s	2m01s	1h41m	91	80	434	3197
Java.0	349	56	69	349	349	8.2s	6.9s	1m06s	-	153	116	303	-
Java.1	350	56	69	69	74	8.2s	6.9s	12m34s	8h42m	144	118	881	3110
Java.2	350	53	65	65	70	8.8s	7.8s	33m32s	9h41m	168	124	1280	3219
Java.3	350	56	69	69	74	8.3s	6.9s	11m56s	7h18m	146	120	953	3140
Java.4	350	55	68	68	73	8.2s	6.6s	21m09s	8h21m	156	119	937	3114
Java.5	350	53	65	65	70	8.3s	6.9s	14m20s	7h44m	153	121	888	3120

Table 1: Results of Filtering (LR1 was run on C and Java after filtering first with SLR1, due to excessive memory usage).

3 Experimental validation

After constructing a new, much smaller, grammar we can apply exhaustive algorithms like Amber or Cfg Analyzer on it to search for the exact sources of ambiguity. The search space for these algorithms is exponential in the size of the grammar. Therefore our experimental hypothesis is:

> By filtering the input grammar we can gain an order of magnitude improvement in run-time when running Amber or Cfg Analyzer as compared to running them on the original grammar.

Since building an LR(0) PPG and filtering it is polynomial we also hypothesize:

> For many real-world grammars the time invested to filter them does not exceed the time that is gained when running Amber and Cfg Analyzer on the filtered grammar.

We will also experiment with other approximations, such as SLR(1), LALR(1) and LR(1) to be able to reason about the return of investment for these more precise approximations.

3.1 Experiment Setup

To evaluate the effectiveness of our approach we must run it on realistic cases. We focus on grammars for reverse engineering projects. Grammars in this area target many different versions and dialects of programming languages. They are subject to a lengthy engineering process that includes bug fixing and specialization for specific purposes. Our realistic grammars are therefore "standard" grammars for mainstream programming languages, augmented with small variations that reflect typical intentional and accidental deviations.

We have selected standard grammars for Java [12], C [13], Pascal [14] and SQL [15] which are initially not ambiguous. We labeled them Java.0, C.0, Pascal.0 and SQL.0. Then, we seeded each of these grammars with different kinds of ambiguous extensions. Examples of ambiguity introduced by us are:

- Dangling-else constructs: Pascal.3, C.2, Java.3
- Missing operator precedence: SQL.1, SQL.5, Pascal.2, C.1, Java.4
- Syntactic overloading[3]: SQL.2, SQL.3, SQL.4, Pascal.1, Pascal.4, Pascal.5, C.4, C.5, Java.1, Java.5
- Nonterminals nullable in multiple ways: C.3, Java.2

For each of these grammars we measure[4]:

1. Amber/Cfg Analyzer run-time and memory usage,
2. Filtering run-time with precisions LR(0), SLR(1), LALR(1) or LR(1),
3. Amber/Cfg Analyzer run-time and memory usage after filtering.

[3] Syntactic overloading happens when reusing terminal symbols. E.g. the use of commas as list separator and binary operator, forgetting to reserve a keyword, or reuse of juxtapositioning.

[4] Measurements done on an Intel Core2 Quad Q6600 2.40GHz PC with 8Gb DDR2 memory.

Observing only a marginal difference between measures 1 and 3 would invalidate our experimental hypothesis. Observing the combined run-times of measure 2 and 3 being longer than measure 1 would invalidate our second hypothesis.

To help explaining our results we also track the size of the grammar (**number of production rules**), the number of harmless productions found with each precision (**rules filtered**), and the number of tokens explored to identify the first ambiguity (**length**).

We have used AMBER version 30/03/2006[5] and CFG ANALYZER version 03/12/2007[6]. To experiment with the NU TEST algorithm and our extensions we have implemented a prototype in the Java programming language. We measured CPU user time with the GNU `time` utility and measured memory usage by polling a process with `pid` every 0.1 seconds.

3.2 Experimental Results

Results of Filtering Prototype All measurement results of running our filtering prototype on the benchmark grammars are shown in Table 1. As expected, every precision filtered a higher or equal number of rules than the one before. Columns 3 to 6 show how much production rules could be filtered with each of the implemented precisions. We see that LR(0) on average filtered respectively 76%, 12%, 19% and 16% of the productions of the SQL, Pascal, C and Java grammars. SLR(1) filtered the same or slightly more, with the largest improvement for the Java grammars: 19%. Remarkably LALR(1) never filtered more rules than SLR(1). LR(1) improved over SLR(1) for 12 out of 20 ambiguous grammars. On average it filtered 78% for SQL, a remarkable 64% for Pascal, and 21% for Java.

Columns 7 to 10 show the run-time of the filtering tool, and columns 11 to 14 show its memory usage. We see that the LR(0) and SLR(1) precisions always ran under 9 seconds and used at most 168Mb of memory. SLR(1) was slightly more efficient than LR(0), which can be explained by the fact that an SLR(1) position graph is generally more deterministic than its LR(0) counterpart. They both have the same number of nodes and edges, but the SLR(1) reductions are constrained by lookahead, which results in a smaller position pair graph.

An LALR(1) position automaton is generally several factors larger than an LR(0) one, which shows itself in longer run-time and more memory usage. The memory usage of the LR(1) precision became problematic for the C and Java grammars. For all variations of both grammars it needed more than 4Gb. Therefore we ran it on the C and Java grammars that we filtered first with the SLR(1) precision, and then it only needed around 3Gb. Here we see that filtering with a lesser precision first can be beneficial for the performance of more expensive filters.

On average the tool uses its memory almost completely for storing the position pair graph, which it usually builds in two thirds of its run-time. The other

[5] downloaded from `http://accent.compilertools.net/`
[6] downloaded from `http://www2.tcs.ifi.lmu.de/~mlange/cfganalyzer/`

	Time				Length			
Grammar	**Orig**	**LR0**	**SLR1**	**LR1**	**Orig**	**LR0**	**SLR1**	**LR1**
SQL.1	28m26s	**0.0s**	0.0s	-	15	9	9	-
SQL.2	0.0s	0.0s	0.0s	-	7	7	7	-
SQL.3	0.0s	0.0s	0.0s	-	6	6	6	-
SQL.4	0.0s	0.0s	0.0s	0.0s	9	9	9	9
SQL.5	1.3s	**0.0s**	0.0s	0.0s	11	9	9	9
Pascal.1	0.3s	0.1s	0.4s	0.4s	9	9	11	11
Pascal.2	0.0s	0.0s	0.0s	0.0s	7	7	8	9
Pascal.3	31.8s	2.9s	1.9s	**1.5s**	11	11	12	13
Pascal.4	0.0s	0.0s	0.0s	0.1s	8	8	9	10
Pascal.5	0.0s	0.0s	0.0s	0.1s	8	8	9	10
C.1	42.1s	0.1s	**0.0s**	-	5	5	5	-
C.2	>4.50h	>19.0h	>15.3h	-	>7	>11	>11	-
C.3	0.1s	0.0s	0.0s	-	3	3	3	-
C.4	42.0s	**0.1s**	0.1s	-	5	5	5	-
C.5	19m09s	0.7s	**0.5s**	-	6	6	6	-
Java.1	>25.0h	15.6h	6m53s	**6m30s**	>12	13	14	13
Java.2	0.0s	0.0s	0.0s	0.0s	1	1	1	1
Java.3	1h25m	6m55s	6m46s	**6m23s**	11	11	14	13
Java.4	17.0s	3.6s	**0.5s**	0.5s	9	9	11	10
Java.5	0.1s	0.1s	0.1s	0.1s	7	7	10	9

Table 2: Running AMBER on filtered and non-filtered grammars.

one third is used to filter the graph. If we project this onto the run-times of Schmitz' C tool [9], it should filter all our grammars with LR(0) or SLR(1) in under 4 seconds, if extended.

Results of Amber Table 2 shows the effects of grammar filtering on the behavior of AMBER. Columns 2 to 5 show the time AMBER needed to find the ambiguity in the original grammars and the ones filtered with various precisions. There is no column for the LALR(1) precision, because it always filtered the same number of rules as SLR(1). For LR(1) we only mention the cases in which it filtered more than SLR(1). AMBER's memory usage was always less than 1 Mb of memory.

In all cases we see a decrease in run-time if more rules were filtered, sometimes quite drastically. For instance the unfiltered Java.1 grammar was impossible to check in under 25 hours, while filtered with SLR(1) or LR(1) it only needed under 7 minutes. The C.2 grammar still remains uncheckable within 15 hours, but the LR(0) and SLR(1) filtering extended the maximum string length possible to search within this time from 7 to 11. The decreases in run-time per string length for this grammar are shown in Figure 3.

This confirms our first hypothesis. But to test our second hypothesis, we also need to take the run-time of our filtering tool into account. The left part of Figure 4 shows the combined computation times of filtering and running AMBER, compared to only running AMBER on the unfiltered grammars. Not all SQL grammars are mentioned because both filtering and AMBER took under 1 second in all cases. Also, timings of filtering with LR(1) are not mentioned

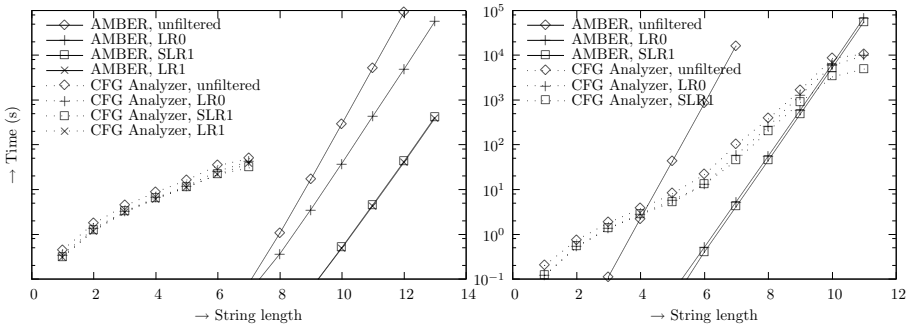


Figure 3: Run-time of Amber and Cfg Analyzer on grammars Java.1 (syntax overloading) left and C.2 (dangling-else) right, corrected for added terminals.

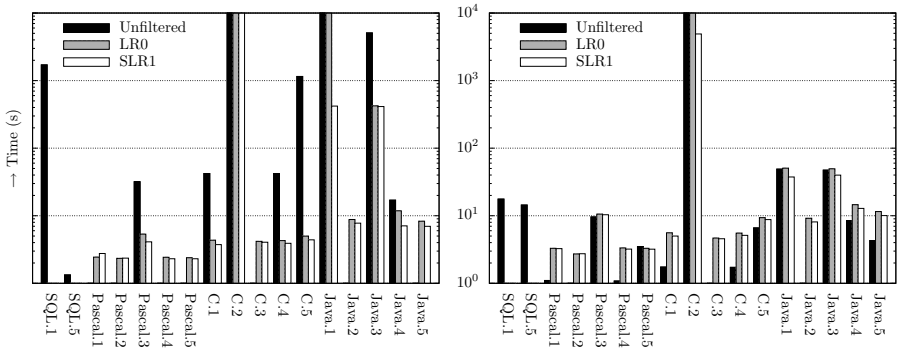


Figure 4: Added run-time of grammar filtering and ambiguity checking with Amber (left) and Cfg Analyzer (right).

because they are obviously too high and would reduce the readability of the graph. Apart from that, we see that the short filtering time of LR(0) and SLR(1) do not cancel out the decrease in run-time for grammars SQL.1, SQL.5, Pascal.3, C.1, C.4, C.5, Java.3 and Java.4. Add to that the effects on grammars C.2 and Java.1 and we get a significant improvement for 10 out of 20 ambiguous grammars. For the other 10 grammars we don't see improvements because Amber already took less time than it took to filter them.

Columns 6 to 9 show the string lengths that Amber had to search to find the ambiguity in each grammar. Only for SQL.1 and SQL.5 we see a decrease in string length due to filtering, in all other cases the needed length remains the same or even goes up. This is explainable by the extra terminals that are introduced in the grammar reconstruction to distinguish the production rules of the new start symbol.

	Time				Length			
Grammar	**Orig**	**LR0**	**SLR1**	**LR1**	**Orig**	**LR0**	**SLR1**	**LR1**
SQL.1	17.6s	**0.1s**	0.1s	-	11	5	5	-
SQL.2	0.4s	0.1s	0.1s	-	3	3	3	-
SQL.3	0.4s	0.0s	0.1s	-	3	3	3	-
SQL.4	1.4s	0.1s	**0.0s**	0.0s	5	5	5	5
SQL.5	14.4s	0.3s	0.4s	**0.2s**	11	9	9	9
Pascal.1	1.1s	0.9s	0.9s	**0.3s**	3	3	3	3
Pascal.2	0.5s	0.4s	0.4s	0.1s	2	2	2	2
Pascal.3	9.6s	8.1s	8.2s	**0.8s**	7	7	7	7
Pascal.4	1.1s	0.9s	0.9s	**0.3s**	3	3	3	3
Pascal.5	3.5s	0.9s	0.9s	**0.3s**	4	3	3	3
C.1	1.7s	1.3s	**1.3s**	-	3	3	3	-
C.2	3.00h	2.77h	**1.36h**	-	11	11	11	-
C.3	0.7s	0.5s	0.5s	-	2	2	2	-
C.4	1.7s	1.3s	1.3s	-	3	3	3	-
C.5	6.6s	5.1s	**4.9s**	-	5	5	5	-
Java.1	48.9s	42.4s	**30.6s**	38.7s	7	7	7	7
Java.2	0.5s	0.4s	0.4s	0.4s	1	1	1	1
Java.3	47.2s	41.2s	33.1s	**29.7s**	7	7	7	7
Java.4	8.4s	6.4s	6.3s	**6.0s**	4	4	4	4
Java.5	4.3s	3.3s	3.2s	**3.1s**	3	3	3	3

Table 3: Running Cfg Analyzer on filtered and non-filtered grammars.

Results of Cfg Analyzer Table 3 shows the same results as Table 2 but then for Cfg Analyzer. Again we see a decrease in run-time in almost all cases, as the number of filtered rules increases, but less significant than in the case of Amber. We also see that Cfg Analyzer is much faster than Amber. It was even able to check the SLR(1) filtered C.2 grammar in 1 hour and 22 minutes. Cfg Analyzer's memory usage always stayed under 70Mb, except for C.2: it used 1.21Gb for the unfiltered grammar, 1.31Gb for the LR(0) filtered one, and 742Mb in the SLR(1) case.

We see a decrease in required string length for the same grammars as Amber. However, there was never an increase in string length and Cfg Analyzer always needed smaller lengths than Amber. This is because Cfg Analyzer searches all parse trees of all non-terminals simultaneously, whereas Amber only checks those of the start symbol.

The right part of Figure 4 shows the combined run-times of our filtering tool and Cfg Analyzer. Here we see only significant improvements for grammars SQL.1, SQL.5, C.2, Java.1 and Java.3. In all other cases Cfg Analyzer took less time to find the first ambiguity than it took our tool to filter a grammar.

3.3 Analysis and conclusions

We saw that filtering more rules resulted in shorter run-times for both Amber and Cfg Analyzer. Especially Amber profited enormously for certain grammars. The reductions in run-time of Cfg Analyzer were smaller but still significant. This largely confirms our first hypothesis.

We conclude that the SLR(1) precision was the most beneficial for reducing

the run-time of Amber and Cfg Analyzer, while requiring only a small filtering overhead. In some cases LR(1) provided slightly larger reductions, but these did not match up against its own long run-time. Filtering with SLR(1) resulted in significant decreases in run-time for Amber on 10 of the 20 ambiguous grammars, and for Cfg Analyzer on 5 grammars. In all other cases the filtering did not contribute to an overall reduction, because it took longer than the time the tools initially needed to check the unfiltered grammars. Nevertheless, this was never more than 9 seconds. Therefore our second hypothesis is confirmed for the situations that really matter.

3.4 Threats to validity

Internally a bug in our implementation would invalidate our conclusions. This is unlikely since we tested and compared our results with other independently constructed tools (NU test [9], Cfg Analyzer and Amber) for a large number of grammars and we obtained the same results. Our source code is available for your inspection at `http://homepages.cwi.nl/~basten/ambiguity/`. Also note that our Java version is slower than the original implementation in C. An optimized version would eliminate some of the overhead we observed while analyzing small grammars[7].

As for external validity, it is entirely possible that our method does not lead to significant decreases in run-time for any specific grammar that we did not include in our experiment. However, we did select representative grammars and the ambiguities we seeded are typical extensions or try-outs made by language engineers.

About the application of our method to scannerless grammars, such as used by SDF [11], we do not have any information. Assuming that the average token length is about 8 in a language like Java, then to let ambiguity detection methods scale to a scannerless grammars would mean to scale to 8 times the currently maximally feasible sentence length. Also, it remains to be seen if our method applied to scannerless grammars would have a similarly positive effect since such grammars are quite different.

4 Conclusions

We proposed to adapt the approximative NU test to a grammar filtering tool and to combine that with the exhaustive Amber and Cfg Analyzer ambiguity detection methods. Using our grammar filters we can conservatively identify production rules that do not contribute to the ambiguity of a grammar. Filtering these productions from the grammar lead to significant reductions in run-time, sometimes orders of magnitude, for running Amber and Cfg Analyzer. The result is that we can produce precise ambiguity reports in a much shorter time for real world grammars.

[7] We are thankful to Arnold Lankamp for his help fixing efficiency issues in our Java version.

References

[1] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental SAT solver. In *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP'08)*, volume 5126 of *LNCS*, July 2008.

[2] H. J. S. Basten. Filtering parse trees of unambiguous strings. Technical report, CWI, 2009. To be published. Available at `http://homepages.cwi.nl/~basten/ambiguity/`.

[3] H. J. S. Basten. The usability of ambiguity detection methods for context-free grammars. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eigth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *ENTCS*, 2009.

[4] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. In Miroslav Balík and Jan Holub, editors, *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, July 2007.

[5] B. S. N. Cheung and R. C. Uzgalis. Ambiguity in context-free grammars. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 272–276, New York, NY, USA, 1995. ACM Press. `http://doi.acm.org/10.1145/315891.315991`.

[6] S. Gorn. Detection of generative ambiguities in context-free mechanical languages. *J. ACM*, 10(2):196–208, 1963. `http://doi.acm.org/10.1145/321160.321168`.

[7] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[8] S. Schmitz. Conservative ambiguity detection in context-free grammars. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *ICALP'07: 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *LNCS*, 2007.

[9] S. Schmitz. An experimental ambiguity detection tool. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, Braga, Portugal, March 2007.

[10] F. W. Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net, 2001. See `http://accent.compilertools.net/Amber.html`.

[11] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 143–158, London, UK, 2002. Springer-Verlag.

[12] Java grammar, GCC, `http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/java/parse.y?rev=1.475`.

[13] C grammar, `ftp://ftp.iecc.com/pub/file/c-grammar.gz`.

[14] Pascal grammar, `ftp://ftp.iecc.com/pub/file/pascal-grammar`.

[15] SQL grammar, GRASS, `grass-6.2.0rc1/lib/db/sqlp/yac.y` from `http://grass.itc.it/grass62/source/grass-6.2.0RC1.tar.gz`.

Tear-Insert-Fold grammars

Adrian Johnstone and Elizabeth Scott

Royal Holloway, University of London
Egham, UK

Abstract

Context Free Grammars (CFGs) are simple and powerful formalisms for defining languages (sets of strings) whose semantics are specified hierarchically — the meaning of a string is determined by terminals and the meanings of substrings. This hierarchy is captured in the derivation tree corresponding to the string.

Derivation trees usually contain more structure than is strictly required to determine the semantics of the string so in practice a simplified or *abstract* syntax tree is used as an internal representation of a concrete text. Indeed, much of the work of a compiler or source-source translator may be described in terms of stepwise transformation of such trees, culminating in a final traversal during which the translated text is output.

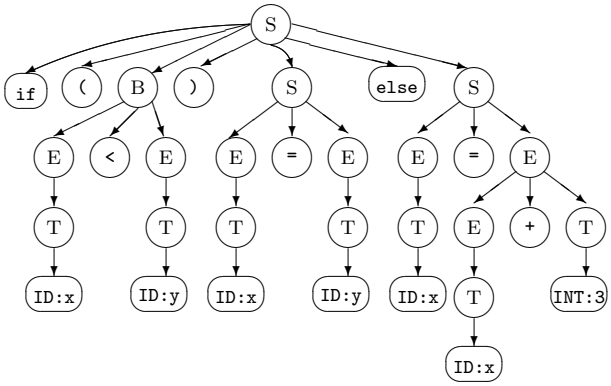
This paper introduces Tear-Insert-Fold (TIF) grammars which add tree-manipulation annotations to standard CFGs. These annotations allow typical abstract forms to be constructed directly from the grammar for the concrete syntax and provide a convenient and concise specification of the relationship between the set of derivation trees and the set of abstract trees for a parser. More significantly, for any TIF grammar Γ_0 there is a TIF grammar Γ_1 whose derivation trees are the abstract trees produced by Γ_0. Γ_1 may itself be used to automatically generate a parser and so by using conventional semantic actions and further TIF annotations, a complete multi-phase translator may be specified using a *chain* of TIF grammars.

1 Introduction

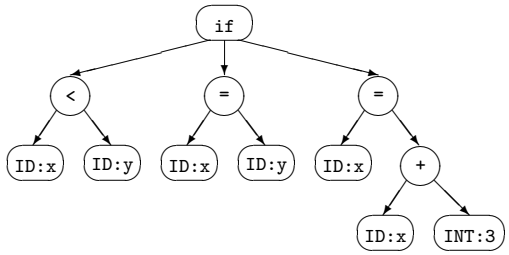
Context free grammars are simple and powerful formalisms for defining languages (set of strings) whose semantics are specified hierarchically, i.e the meaning of a string is determined by terminals and the meanings of substrings. For example, the meaning of `if(` B `)` $S1$ `else` $S2$, where B, $S1$ and $S2$ are substrings, is defined via a rule that is identified by the keywords `if-else` and parameterised by the meanings of B, $S1$ and $S2$. This hierarchical nature is captured in the derivation tree corresponding to the string.

The concept of abstract syntax (as distinct from the concrete syntax used in language definitions such as the Algol-60 report) was first proposed by John McCarthy in his 1962 paper to the IFIP congress [3]. Because a grammar must generate all the strings in the required language, and because the structure of the language must be such that the substrings on which the semantics depends can be identified, the derivation tree usually contains more structure than is required to determine the semantics of the string. In practice, then, a syntactically simpler structure is usually used for semantic evaluation which is often referred to as *abstract syntax*.

For example, the derivation tree for the string `if(x<y) x=y else x=x+3` might have the form



while the simpler abstract syntax tree



contains all the information required for semantic analysis. Looking at these two trees we can see that some nodes, such as `if` and `=` have effectively been overlaid on their parents. We call this sort of tree transformation a *fold*: we imagine that a node in the tree has been folded under, or folded over its parent; if the folded node has children they are dragged up into the children of the parent node. When node A is folded under node B, node A is deleted from the tree; when node A is folded over node B then node B is relabelled with the label from A. In the example there are several instances of chains E–T–`ID`; in these cases we think of T as being folded over E, giving T–`ID` and then `ID` being folded over T.

It can, of course, be the case that a desired abstract syntax has a more complicated relationship with the concrete syntax. For example, many programming languages permit an `if` statement to have an optional `else` clause. A corresponding semantic analyser may then have to include two different sections to deal with the two forms of `if` statement. It may be desirable to unify the two forms by adding an `else` clause which executes an empty statement. Thus as well as removing nodes from a tree we may also wish to insert them to achieve uniformity.

Programming languages typically also retain formally redundant constructs, for example the same flow of control can be specified with either the standard `for` loop or the standard `while` loop. We expect these constructs to be included in the language for the convenience of the user, but it is possible to have a 'core' language whose semantics are formally defined directly, with the remaining constructs defined in terms of the core ones. It would be useful to transform the derivation tree associated with a non-core construct into the corresponding tree of core constructs. In effect we want to be able to remove and reorder subtrees. We think of this reordering process as a combination of *tearing* subtrees out and then *inserting* them elsewhere under the same parent node. We also allow completely new subtrees to be constructed and inserted so that, for instance, optional phrases in the concrete syntax can have their default values directly added to the tree; in this way `if-then` statements can be normalised to `if-then-else` statements.

This paper introduces Tear-Insert-Fold (TIF) grammars which add four tree-manipulation annotations to standard CFGs: fold-under (`^`); fold-over (`^^`); tear(`^^^`) and insert `[ ]`. These annotations allow typical abstract forms to be constructed directly from the grammar for the concrete syntax and provide a convenient and concise specification of the relationship between the set of derivation trees and the set of abstract trees for a parser. More significantly, for any TIF grammar Γ_0 there is a TIF grammar Γ_1 whose derivation trees are the abstract trees produced by Γ_0. Γ_1 may itself be used to automatically generate a parser and so by using conventional semantic actions and further TIF annotations, a complete multi-phase translator may be specified using a *chain* of TIF grammars.

1.1 The rôle of abstract syntax

Abstract syntax has become a core part language design and implementation. Language sensitive environments and language sensitive environment generators are typically based around an abstract syntax tree (AST) structure which may be edited and annotated with directions to the 'unparser' that presents concrete views to the user of the environment. Multi-pass translators, in particular optimising compilers, typically start with parsers that return an AST rather than a full derivation tree.

For such a venerable and central concept in computer language related research, abstract syntax has a remarkably sparse literature. It would be easy to list several hundred papers that use or specify abstract syntaxes, but very few

authors have attempted to formally define the relationship between concrete and abstract syntax. To the compiler writer, this is particularly disappointing given that in practice the AST is the interface between parsers and attribute evaluators: the two parts of the compilation process that *have* yielded to formal analysis to such an extent that parsers and evaluators may be automatically generated from grammar based specifications. It would be convenient if the interface between these two could also be automatically generated.

From the point of view of the theoretician, this lack of formal linkage is *not* surprising because the concrete and abstract syntaxes perform such different rôles. McCarthy's proposal was that the abstract syntax should be "analytic rather than synthetic, it tells how to take a program apart, rather than how to put it together" [3]. This is a nice distinction, but most authors now describe abstract syntax as being a reduced cousin of concrete syntax with some nonterminals and terminals discarded and occasionally with more drastic restructuring. Compiler writers often simply view abstract syntax as a stripped down form of the concrete syntax. Semantics researchers typically focus on an abstract syntax and dismiss the concrete details as a pragmatic technical detail to be grafted on later [4]. Language sensitive environment tools based on abstract syntax have some built-in notion of AST into which programs written in particular concrete syntaxes must be mapped [2].

Although many in the semantics community would argue that this route leads to poor designs we must accept that proceeding *from* a concrete syntax to an abstract syntax is the natural approach for an engineer implementing a translator for an existing language, or for a new language in which the traditional procedure has been adopted of defining a language by adjusting a grammar so that it passes through the locally available parser generator, and then adding semantic routines by decorating the grammar with attributes and actions.

Our position is that we should encourage *specification* of abstract trees using our TIF operators rather than just using *ad hoc* tree construction semantic actions. The translator designer uses a single notation, the annotated Tear-Insertion-Fold (TIF) grammar, to specify both the language being parsed and the intermediate form to be generated. The problem of grammar maintenance is eased because as a grammar for the source language is modified during debugging there is no need to apply separate modifications to the intermediate form specification. Our method allows us to automatically derive a new grammar which describes the resulting intermediate form trees; this grammar can itself be annotated to generate a next stage walker, so in fact the single TIF notation can be used both for conventional string parsing and for tree rewriting stages. This opens up the possibility of specifying complete multi-phase translators as a *chain* of TIF grammars.

In the rest of this paper, we first look at TF grammars which provide the tear and fold operations; then in Section 3 we examine various forms of insertion leading to the full definition of a TIF grammar. In Section 4 we show how to construct a *TIF Transformed Grammar* (TTG) from a TIF grammar. In Section 5 we discuss applications and open issues.

2 Grammars, trees and TF grammars

A *context free grammar* (CFG) consists of a set $\mathbf{N}$ of nonterminals, a set $\mathbf{T}$ of terminals, a start symbol $S \in \mathbf{N}$, and a set of rules of the form $X ::= \alpha$, where $X \in \mathbf{N}$ and $\alpha \in (\mathbf{T} \cup \mathbf{N})^*$, the set of strings of elements of T and N. The empty string is denoted by ϵ.

A *derivation step* is of the form $\delta X \gamma \Rightarrow \delta \alpha \gamma$ where $\delta, \gamma \in (\mathbf{T} \cup \mathbf{N})^*$ and $X ::= \alpha$ is a rule in the grammar. A *derivation* is a sequence of derivation steps

$$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow u \in T^*,$$

u is a *sentence* of the grammar, and the *language generated by the grammar* is the set of all sentences.

A *derivation tree* is a tree in which the children of each node are ordered. The interior nodes of a derivation tree are labelled with nonterminals in the grammar, the leaf nodes are labelled with terminals, the root node is labelled with the start symbol, and the children of an interior node labelled X are labelled, in order, with the symbols in α, for some rule $X ::= \alpha$. If the leaf nodes of a derivation tree are read in left to right order they yield a sentence of the grammar.

Each derivation of a sentence corresponds to a unique derivation tree but a derivation tree usually corresponds to several derivations of a sentence. If there are two or more trees which correspond to derivations of the same sentence then the grammar is said to be *ambiguous*.

In the previous section we discussed restructurings of a derivation tree by removing and inserting subtrees and by removing individual nodes. We now describe a mechanism for taking a user-annotated grammar and using it to generate a particular form of abstract syntax tree which we call a *restructured derivation tree*, RDT.

2.1 Tears and folds

We begin with a CFG in which the symbols on the right hand sides of rules have optionally been annotated with `^`, `^^` or `^^^`, subject to the restriction that each right hand side may contain at most one symbol annotated with `^^`. We call this a *tear-fold grammar* (TF). (In the next section we shall extend the definition to include insertions.)

We shall consider the following grammar, Γ_1, as a working example.

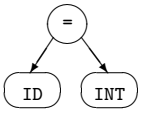
```
S ::= if^^ (^ B )^ S else^ S  |  ID =^^ E  | T^^^ a
B ::= E <^^ E
E ::= T +^^ E | T^^
T ::= ID^^ | INT^^
```

(The inclusion of the alternate `T^^^ a` is simply to illustrate tears, whose main use is in combination with insertions, see below.) The notion of a derivation in a TF (and in a TIF) is identical to the standard notion; only the structure of the

corresponding tree is different. Informally, the node corresponding to a symbol `x^^^` is deleted together with all of the subtree of which it is the root, the node corresponding to a symbol `x^` is deleted and its children become children of its parent, while the node corresponding to a symbol `x^^` replaces its parent node. In the example grammar, for the derivation

$$S \;\Rightarrow\; ID = E \;\Rightarrow\; ID = T \;\Rightarrow\; ID = INT$$

the node labelled `S` is replaced by the node labelled `=`, the node labelled `E` is replaced by the node labelled `T` which is then finally replaced by the node labelled `INT`, resulting in the abstract tree



We define a TF grammar formally as an attribute grammar whose attributes define the RDT.

2.2 TF grammars

Syntax-directed definitions [1] are grammars that have, at the end of each grammar rule, semantic rules which are used to calculate the values of attributes of grammar symbols. In our case, each nonterminal, `x`, has three attributes: `x.node`, which is a tree node created by some function `newNode("y")` which constructs and returns a new tree node labelled `y`, `x.label` which is a string and `x.kids` which is a sequence of tree nodes. If `x` is a terminal we will always have `x.node=newNode("x")`, `x.label = "x"` and `x.kids` = $\emptyset$ so we will leave these definitions out of the semantic rules.

For each grammar rule $X ::= x1\ x2\ \ldots\ xn$ we give semantic rules to define, in the RDT , the node corresponding to X, its label and its children. If we want the standard derivation tree construction for this rule we set `X.label = "X"` and `X.kids = (x1.node, x2.node, .., xn.node)`. If we wish to model the TF in which the rule is annotated with `xi^` we replace `xi.node` with `xi.kids` in the rule for `X.kids`. If we wish to model the TF in which the rule is annotated with `xi^^` we replace `xi.node` with `xi.kids` in the rule for `X.kids` and set `X.label = xi.label`. If we wish to model the TF in which the rule is annotated with `xi^^^` simply we leave `xi.node` out of the rule for `X.kids`.

In this exposition we use the common attribute grammar convention of giving symbols which are repeated in an alternate instance numbers. Then the formal syntax-directed definition which corresponds to the TF , Γ_1, in Section 2 is

```
S ::= if ( B ) S1 else S2
                { S.node = newNode("if"),  S.label = "if",
                  S.kids = (B.kids, S1.kids, S2.kids) }
S ::= ID = E    { S.node = newNode("="),  S.label = "=",
                  S.kids = (newNode("ID"),  E.node) }
```

```
S ::= T a       { S.node = newNode("S"),  S.label = "S",
                  S.kids = newNode("a") }
B ::= E1 < E2   { B.node = newNode("<"),  B.label = "<",
                  B.kids = (E1.node,  E2.node) }
E ::= T + E     { E.node = newNode("+"),  E.label = "+",
                  E.kids = (T.node, E.node) }
E ::= T         { E.node = newNode(T.label),  E.label = T.label,
                  E.kids = T.kids }
T ::= ID        { T.node = newNode("ID"),  T.label = "ID",
                  T.kids = ( ) }
T ::= INT       { T.node = newNode("INT"),  T.label = "INT",
                  T.kids = ( ) }
```

We shall use the annotated version of a TF rather than the equivalent syntax-directed definition, and we define the RDT to be the tree constructed by evaluating the attribute rules and taking as the root node the node `S.node` constructed by the semantic rule defined by the first derivation step.

3 Insertions and TIF grammars

Annotating a grammar as discussed above can remove nodes which are not required for semantic specification. However, as we have mentioned in the introduction, it may also be useful to insert nodes into a derivation tree, or tear subtrees and reinsert them to effect reordering of a node's children.

We now extend our grammar annotations using brackets `[ ]` to denote insertions. Our formalism permits a language designer to provide a library of preconstructed trees, each tree being identified by its root node. These nodes can then be made available for inclusion in the semantic rules.

Informally, given an annotated rule of the form

```
X ::= a [ libNode3 ] b
```

the parser parses the string ab but it adds an additional child, and its subtree, to the node corresponding to X, between the nodes for a and b. The subtree is identified in the library as `libNode3`.

This operation is defined formally in the attribute grammar by simply adding the name of the root node to a list of children. For example,

```
X ::= a b       { X.node = newNode("X"),  X.label = "X",
                  S.kids = (newNode("a"), libNode3, newNode("b") }
```

We call a grammar annotated with a selection of `^`, `^^`, `^^^` and `[]` operators a *tear-insertion-fold* (TIF) grammar.

The library of trees can contain anything the designer chooses to specify, however in practice we have no immediate applications for a library of general insertion trees. We focus on two special cases, the insertion of single nodes and the insertion of the derivation tree of a specific string.

If the tree to be inserted is a singleton node then instead of putting it into a library we can simply specify the label of the node directly in the TIF grammar. For example, when the substring ab is parsed by the rule

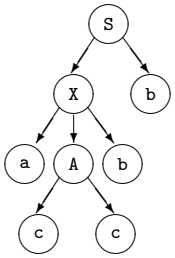
```
X ::= a [ "c" ] b
```

the subtree produced has root node labelled X with three children labelled a, c and b.

If the the subtree to be inserted is the RDT corresponding to the derivation of a string u from a nonterminal A then we can specify this as using the annotation `[A(u)]`. For example, given

```
S ::= X b
X ::= a [ A(cc) ] b
A ::= c A | c^^
```

if the string abb is parsed the corresponding RDT will be



Library insertions and the second of the above special cases have limited application at the moment and we shall not consider them further in this paper. However, we note that they are related to the modifications made to Tomita's GLR parsing algorithm which result in the RNGLR algorithm [6]. Tomita's algorithm failed on grammars with hidden left recursion, and the initial correction [5] for this involves substantially increased searching in the case of grammar rules whose right hand ends can derive ϵ (right nullable rules). The RNGLR solution is to extend the definition of an LR reduction item to include items of the form $X ::= \alpha \cdot \beta$, if $\beta \stackrel{*}{\Rightarrow} \epsilon$. The corresponding parser then performs the reduction when α has been matched, implicitly matching β to ϵ. This gives a correct recogniser but the parser needs a library of ϵ derivation trees which must be added to the tree construction to get the correct outcome. An alternative approach could be to rewrite the grammar so that each nullable nonterminal has an additional non-nullable version and each right nullable rule $X ::= \alpha\beta$ is replaced with a non-right-nullable version $X ::= \alpha\beta'$ and the truncated rule $X ::= \alpha$. A TIF version of this grammar can then be constructed with an annotated version `X::=`α `[`$\beta(\epsilon)$`]` of the latter rule. A standard LR parse table for this grammar can then be used in place of the RN-table, and the additional ϵ subtrees can be constructed on demand during the parse. We do not believe that this approach is preferable to RNGLR in practice but does provide a different method of comparing RNGLR with other GLR implementations such as SDF [8].

3.1 Copy insertions

Library-based insertions are static in the sense that they depend only on the grammar rule used and not on the particular string being parsed. To, for example, transform a tree based on a `for` loop into an equivalent tree based on a `while` loop we need to insert subtrees which do depend on the string being parsed. We can achieve this by inserting copies of subtrees that have already been constructed. We would normally expect such an insertion to be accompanied by a tear, so that the position of the subtree is moved and not just copied, but this is not necessary.

Copies and corresponding insertion attributes must be located within the same grammar rule, and we simply use the required grammar symbol in the insertion.

The following attribute grammar rule defines a C-style `do` loop, but the corresponding RDT is a tree for the corresponding `while` loop.

```
S ::= do S1 while ( E )
       { S.node = newNode("S"),  S.label = "S",
         S.kids = (S1.node, newNode("while"), newNode("("),
                                  E.node, newNode(")"), S1.node) }
```

In the TIF grammar we need to give symbols associated with copy insertions distinguishing names. We do this using a colon, `X:n1`. Then the equivalent TIF grammar rule for the above attribute rule is

```
S ::= do^ S:n1 while ( E ) [ S:n1 ]
```

We can also use TIF grammars to move from one programming language idiom to another. For example, the following example parses a C-style variable declaration and builds an RDT for the corresponding Pascal-style declaration.

```
S ::= typeName:n1^^^ [ "var" ] idName [ ":" ] [ typeName:n1 ]
```

4 TIF transformed grammars (TTGs)

Once we have an abstract syntax we want to use it for semantic analysis. It is conventional to write a program which takes the abstract syntax and performs semantic analysis. The problem is that if we change the syntax during the design of the language then we have to rewrite the semantic analyser. Our goal is to produce a context free grammar whose derivation trees are the RDTs of a given TIF grammar (subject to a systematic node label convention). In certain cases the context free grammar will have a larger set of derivation trees, but every RDT of the TIF grammar will be a derivation tree of the CFG. We call such a grammar the TIF transformed grammar, TTG , for the given TIF. The TTG can then be used to walk the RDT and perform semantic actions. In this way when the original CFG is changed a new semantic analyser can be automatically generated.

In this paper we only consider BNF grammars, although we shall use both the multiple alternate and factoring conventions, so `X::=a (B b | C )` is considered to be shorthand for two rules `X::= a B b`, `X ::= a C`. We shall also only consider single node insertion and copy insertions, although general library insertions could be incorporated in a straightforward way by writing, for each library tree, grammar rules which generate precisely that derivation tree.

Insertions and tears are relatively simple to deal with, but folds require more careful treatment because children of annotated nodes must be substituted for their parent, and `^^` annotations change the label of a node. We say that `C` *step substitutes* into `A` if there is a rule of the form `A::=`α `C^` β or `A::=`α `C^^` β. We say that `C` *substitutes* into `A` if there is a sequence $\mathtt{A}_0, \mathtt{A}_1, \ldots, \mathtt{A}_n$ such that $n \geq 1$, $\mathtt{A}_0 = \mathtt{A}$, $\mathtt{A}_n = \mathtt{C}$ and $\mathtt{A}_i$ step substitutes into $\mathtt{A}_{i-1}$, $1 \leq i \leq n$.

We only consider TIFs in which no nonterminal A substitutes into itself. In this case we say that the TIF has no self substitution. TIFs with self substitution do not have equivalent unannotated CFGs, a situation which which shall also discuss briefly at the end of this paper.

We call TIFs with no self substitution whose insertions are restricted to single node and copy insertions, *plain* TIFs.

4.1 Informal approach

We begin with an informal discussion of TTGs which correspond to plain TIFs then we will describe formally how to construct a TTG from a plain TIF.

Dealing with tears is trivial: we simply leave them out of the rules for the TTG. Insertions are also easy to deal with, we just remove the brackets and the identifying name. So for the TIF rule

```
X ::= a Y:t^^^ b Y Z^^^ d^ [ Y:t ] a
```

the corresponding TTG rule is

```
X ::= a b Y Y a
```

Now we focus on folds. To use the standard derivation tree construction to generate, from a TTG , the RDTs generated by a TIF , we need the TTG to have some nonterminal names which are terminal names TIF. For example, the (EBNF) grammar

```
S ::= + | INT | ID     + ::= (ID | INT)  ( + | ID | INT)
```

generates as derivation trees the RDTs generated by the grammar

```
S ::= E      E ::= T +^^ E | T^^    T ::= ID^^ | INT^^
```

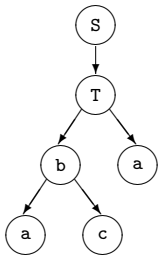
Of course, this approach does not generalise. If the original grammar has several instances of a symbol then there may need to be several different corresponding nonterminals in the TTG. For example, the RDTs generated by Γ_2

```
S ::= T     T ::= a b^^ c | E a | b      E ::= b^^ d
```

are a subset of the derivation trees generated by the grammar

```
S ::= T  | b   T ::= b a | b    b ::= a c    b ::= d
```

but the latter grammar also generates, for example, the tree



We can create new nonterminals for each instance `x^^`, then derivation trees in the TTG correspond to RDTs in the original TIF modulo the correspondence between the new nonterminal names and the original terminal names. So

```
S ::= T  | b1   T ::= b2 a | b    b1 ::= a c    b2 ::= d
```

generates derivation trees which correspond to the RDTs generated by Γ_2.

It is the nature of context free grammars that two rules with the same left hand side can always be used at any point in a derivation, so it is possible to use the same nonterminal to correspond to several instances of `x^^`, if they occur in rules which have the same left hand side. So we shall use nonterminals to distinguish different versions of the original names. For example, the RDTs generated by

```
S ::= E      E ::= ID +^^ ID  | +^^ ID  | ID
```

correspond to the derivation trees generated by

```
S ::= E | +E    +E ::= ID ID | ID     E ::= ID
```

when the labels $+_E$ are associated with the labels `+` in the RDTs.

Annotated terminals are simply removed from the right hand sides where they occur, but annotated nonterminals have to be replaced by substituting the right hand sides of their rules. For example, for the TIF

```
S ::= E       E ::= a T^  | b T^^ a     T ::= c | S
```

a corresponding TTG is

```
S ::= E | TE    E ::= a c | a S     TE ::= b c a | b S a
```

4.2 Formal TTG construction

We illustrate the process with a slightly modified version, Γ_1', of Γ_1 in which the rule `S::=T^^^ b` is replaced with `S::=T:t^^^ b [ T:t ]`.

Step 1 We replace rules of the form `X ::=`α `[ Y:t ]` β with rules `X::=`α `Y` β until there are no further square brackets, and we remove all the symbols annotated with `^^^` from all the rules.

For Γ_1', we replace `S::=T:t^^^ b [ T:t ]` with `S::= b  T`.

Step 2 Replace each rule of the form `X::=`α `x^^`β `y^^`γ with `X::=` α `x^`β `y^^`γ until all rules have at most one `^^` annotation.

Step 3 For each nonterminal `X` and symbol `x` such that there is a rule of the form `X::=` α `x^^`β create a new nonterminal $\mathtt{x}_X$. Then for each nonterminal `X` create a set $base_0$`(X)` as follows. For each rule `X::=`α, if α does not contain a terminal annotated with `^^`, add `X::=`α' to $base_0$`(X)` where α' is obtained from α by removing all the terminals annotated with `^`. If α contains `a^^` add $\mathtt{a}_X$`::=`α' to $base_0$`(X)` where α' is obtained from α by removing `a^^` and all the terminals annotated with `^`.

For the grammar Γ_1' we have

$base_0$`(S) = { if`$_S$ `::= B S S` `=`$_S$`::= ID E` `S ::= b T` `}`
$base_0$`(B) = { <`$_B$ `::= E E }` $base_0$`(T) = { ID`$_T$ `::=` INT$_T$`::= }`
$base_0$`(E) = { +`$_E$ `::= T E` `E ::= T^^ }`

Step 4 Starting with $i = 0$ construct sets $base_{i+1}$`(X)` from $base_i$`(X)` by substituting for the nonterminals annotated with `^^` as follows. For each rule `X::=` α `Y^^`β in $base_i$`(X)` and for each rule `Y::=`γ in $base_i$`(Y)` if γ contains a symbol `W^^` add `X::=` $\alpha\gamma\beta$ to $base_{i+1}$`(X)` otherwise add $\mathtt{Y}_X$`::=` $\alpha\gamma\beta$ to $base_{i+1}$`(X)`. For each rule $\mathtt{x}_Y$`::=`γ in $base_i$`(Y)` add $\mathtt{x}_X$`::=` $\alpha\gamma\beta$ to $base_{i+1}$`(X)`.

For the grammar Γ_1' we have

$base_1$`(S) = { if`$_S$ `::= B S S` `=`$_S$`::= ID E` `S ::= b T` `}`
$base_1$`(B) = { <`$_B$ `::= E E }` $base_1$`(T) = { ID`$_T$ `::=` INT$_T$`::= }`
$base_1$`(E) = { +`$_E$ `::= T E` ID$_E$ `::=` INT$_E$`::= }`

Step 5 Let K be the integer such that none of the sets $base_K$`(X)` contains any rules of the form `Z::=` α `Y^^`β. Starting with $i = K$ construct sets $base_{i+1}$`(X)` from $base_i$`(X)` by substituting for the nonterminals annotated with `^` as follows. For each rule `Z::=` α `Y^`β in $base_i$`(X)` and for each rule `W::=`γ in $base_i$`(Y)` add `Z::=` $\alpha\gamma\beta$ to $base_{i+1}$`(X)`.

Note: we can carry out the substitutions in Steps 4 and 5 in any order as long as we do not replace a rules `Z::=`α `Y^`β while there is still a rule of the form `W::=`γ `V^^`δ in $base_i$`(Y)`.

Let $base$`(X)` denote the sets constructed at the end of Step 5. For the grammar Γ_1, $base_1$`(X)`=$base$`(X)` for all `X`.

Step 6 For each nonterminal X from the original grammar let $X_{new} = X_1 \mid \ldots \mid X_m \mid w_1 \mid \ldots \mid w_k$ where $X_1, \ldots, X_m$ are the left hand sides of nonempty rules in $base(X)$ and $w_1, \ldots, w_k$ are the symbols such that there is an 'empty' rule of the form $(w_i)_X ::=$ or $w_i ::=$ in $base(X)$.

For any string α let α_{new} be the result of replacing each nonterminal X in α with X_{new}.

Create a new start symbol `S` and rule `S'::=`S_{new}. The rules of the TTG are the rules `W::=`α_{new}, where `W::=`α is a nonempty rule in some $base(X)$.

For the grammar Γ_1' we have

`S' ::= if`$_S$ `| =`$_S$ `| S  S ::= b T`
`if`$_S$ `::= B ( if`$_S$ `| =`$_S$ `| S )( if`$_S$ `| =`$_S$ `| S )`
`=`$_S$ `::= ID ( +`$_E$ `| ID | INT )`
`<`$_B$ `::= ( +`$_E$ `| ID | INT ) ( +`$_E$ `| ID | INT )`
`+`$_E$ `::= (ID | INT) ( +`$_E$ `| ID | INT )`

4.3 Example

```
S ::= S +^^ S  |  +^^ S | a^^ | A b A^ | A^^ b
A ::= c^ | +^^ d
```

Step 3 :

$base_0$`(S) = { +`$_S$ `::= S S | S  a`$_S$ `::=  S ::= A b A^ | A^^ b }`
$base_0$`(A) = { A ::=  +`$_A$ `::= d }`

Step 4 :

$base_1$`(S) = { + `$_S$ `::= S S | S  a`$_S$ `::=  S ::= A b A^  A`$_S$ `::= b`
`+`$_S$ `::= d b }`
$base_1$`(A) = { A ::=  +`$_A$ `::= d }`

Step 5 :

`S' ::= +`$_S$ `| S | A`$_S$ `| a`
`+`$_S$ `::= ( +`$_S$ `| S | A`$_S$ `| a)( +`$_S$ `| S | A`$_S$ `| a)`
`| ( +`$_S$ `| S | A`$_S$ `| a) | d b`
`S ::= ( +`$_A$ `| A ) b | ( +`$_A$ `| A ) b | b d`
`A`$_S$`::= b  +`$_A$`::= d`

Notice, this grammar has `A` as a terminal.

5 Concluding remarks and open issues

We have implemented the TIF operators in our ART parser generator tool which uses the GLL [7] algorithm to create generalised parsers and tree traversers. We are now beginning to build experience of using this formalism as an engineering tool. For the construction of simple abstract trees and many source-to-source transformations the tool is comfortable and convenient. ART can also produce TTGs, and those TTGs are written in ART's TIF syntax, so that they can be input to a second stage parser generation. (In detail, tree walkers can use a simplified version of GLL, so ART can be switched to generate string parsers or tree walkers.)

An immediate consequence of the locality of the TIF operators and our ability to create multi-phase translators is that we can take a TIF grammar and decompose it. For instance, we can have an initial grammar which has the semantic actions and TIF annotations for say, expressions but which leaves the other rules unannotated. The TTG will have copies of the unannotated rules as well as the new rules describing the rearranged expression trees. This second stage grammar can then be annotated to describe, say, statement trees and semantics. Thus we have separation of concerns with separate phases to handle different parts of the original concrete grammar. ART also provides modularity control constructs which, in combination with the multi-phase support, allow multiple programmers to be working independently on different parts of a translator.

In practice we expect multi-phase translators to also feature some larger scale reworking of the TTG before it is input to the next stage. In particular, it is common to simplify the TTG by combining rules or even deleting rules when ambiguous grammars are constructed. In the accompanying Tools Paper we give a simple example in which a constant expression is inserted as a default, and that results in an ambiguous TTG in which the constant expression and a fully general expression 'overlap'. Since the constant expression lanaguage is a subset of the fully general expression language, the TTG rule for the constant expression may be safely deleted. *Note to referees: we'll rework this section if the tools paper is not accepted for LDTA10*

An interesting question is: what useful tree transformations are there that *require* multi-phase grammars? It is clear that some transformations cannot be performed in a single pass simply because there has to be a convention governing the order in which the fold operators are applied, and as a result fold-over (`^^`) operators do not propagate through fold-under (`^`) operators. Such transformations thus require multi-phasing however, we do not, at this stage, have an application which requires them.

There are some theoretical issues where we have a choice of approach, and we shall experiment with the alternatives in engineering applications. For instance, if a TIF contains a rule of the form `X::=a X^^ b |` ϵ or `X::=a X^ b |` ϵ (the self substitution mentioned in Section 4), then it is not possible to find a BNF CFG whose derivation trees include all the RDTs of the TIF. The infinite set of trees whose parent is labelled `X` and which have n children labelled `a` followed by

n children labelled `b`, for every value of $n \geq 0$, cannot be specified as derivation trees of a BNF grammar. We could use an EBNF grammar `X::=a*b*` but this depends on the definition of a derivation tree from an EBNF grammar, and in any case this will also generate a large class of trees which are not RDTs of the original TIF.

Our current proposal is that for TIF grammars with self substitution the corresponding TTG should be a TIF rather than a CFG. This TTG would have the property that the only annotations are immediate self substitution, i.e. rules of the form `X ::=`α `X^^`β or `X ::=`α `X^`β.

Finally, in this paper we have not considered tree nodes having attributes. In particular, when we perform a copy-insertion we end up with two copies of a sub-tree, and it is not clear whether each sub-tree should have its own attributes or whether they should share. In practice, both are likely to be needed: an identifier might have its symbol table entry as an attribute, and it is likely that both sub-trees would want to share the symbol table entry. On the other hand attributes representing local computations, such as the sequence number of a variable, would need different attributes.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] Miren Begoña Albizuri-Romero. Internal representation of programs in GRASE. *SIGLAN notices*, 20(8):41–50, 85.

[3] John McCarthy. Towards a mathematical science of computation. In *Information processing 62, Proc. IFIP congress 62*, pages 21–28, Amsterdam, 1962. North-Holland.

[4] Peter Mosses. *Action Semantics*. Cambridge University Press, 1992.

[5] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 60–75. Kluwer Academic Publishers, The Netherlands, 1991.

[6] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[7] Elizabeth Scott and Adrian Johnstone. GLL parsing. In Jurgen Vinju and Torbjorn Ekman, editors, *LDTA'09 9th Workshop on Language Descriptions, Tools and Applications*, also in Electronic Notes in Theoretical Computer Science, 2009.

[8] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

Embedding a Web-Based Workflow Management System in a Functional Language

EXPERIENCE PAPER

Jan Martin Jansen

Faculty of Military Sciences, Netherlands Defence Academy, Den Helder, the Netherlands

Rinus Plasmeijer, Pieter Koopman, and Peter Achten

Institute for Computing and Information Sciences (ICIS), Radboud University Nijmegen, the Netherlands

Abstract

Workflow management systems guide and monitor tasks performed by humans and computers. The workflow specifications are usually expressed in special purpose (graphical) formalisms. These formalisms impose severe restrictions on what can be expressed. Modern workflow management systems should handle intricate data dependencies, offer a web-based interface, and should adapt to dynamically changing situations, all based on a sound formalism. To address these challenges, we have developed the iTask system, which is a novel workflow management system. We *entirely embed* the iTask specification language in a modern general purpose *functional* language, and *generate* a complete workflow application. In this paper we report our experiences in developing the iTask system. It not only inherits state-of-the-art programming language concepts such as generic programming and a hybrid static/dynamic type system from the host language Clean, but also offers a number of novel concepts to generate complex, real-world, multi-user, web based workflow applications.

1 Introduction

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *specification* plays a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified as a structured and ordered collection of tasks that are assigned to available resources at run-time. In

many WFMSs, the workflow specification only determines the framework for the workflow application, i.e. a *partial* workflow *application*. In other WFMSs one has to provide much details in the workflow specification. In both approaches substantial coding is required to complete the workflow application. In general, this results in complex distributed, multi-user and heterogeneous applications that are hard to maintain.

In this paper, we report on our experience in designing, building, and deploying the iTask system [12], which is a novel WFMS based on state-of-the-art programming language concepts with firm roots in functional programming. We developed the iTask system, because of a number of perceived issues with contemporary WFMSs. Their complex nature makes it very hard to correctly create a complete application from the partial application that is generated by them. Furthermore, contemporary WFMSs use special purpose (mostly graphical) specification languages to enable the rapid development of a workflow framework. Unfortunately, these formalisms often offer limited expressiveness. First, *recursive definitions* are commonly inexpressible, and there are only limited ways to make *abstractions*. Second, workflow models usually only describe the *flow of control*. Data involved in the workflow is mostly maintained in databases and is extracted or inserted when needed. Consequently, workflow models cannot easily use this data to parameterize the flow of work. This results in more or less pre-described workflows that cannot be dynamically adapted. Third, these dedicated languages usually offer a fixed set of *workflow patterns* [1]. However, in the real world work can be arranged in many ways. If it does not fit in a (combination of) pattern(s), then the workflow specification language probably cannot cope with it either. Fourth, and related, is the fact that functionality that is not directly related to the main purpose of the special purpose language is hard to express. To overcome this limitation, one either extends the special language or interfaces with code written in other formalisms. In both cases one is better off with a well designed general purpose language.

For the above reasons, the iTask system is a *domain specific language* that is *embedded* in a textual, formal general purpose *programming language* as a workflow specification language. This allows us to address all computational concerns within the workflow specification and provides us with general recursion. We use a *functional* language, because it offers a lot of expressive power in terms of modeling domains, use of powerful types, and functional abstraction. We use the *pure* and *lazy* functional programming language Clean, which is a state-of-the-art language that offers fast compiler *and* interpreter technology, generic programming features [2], a hybrid static/dynamic type system [16], which are paramount for generating systems from models in a type-safe way. Workflows modeled in the iTask system result in complete workflow applications that run on the web distributed over server and client side [14]. Clean and the iTask system can be found at `http://clean.cs.ru.nl/` and `http://itask.cs.ru.nl`.

The remainder of this paper is organized as follows. We present the iTask system in Sect. 2 and give a case study in Sect. 3. We discuss our experience in Sect. 4 and 5. Related work is discussed in Sect. 6. We conclude in Sect. 7.

2 Overview of the iTask system

The iTask system is a scientific prototype of a WFMS. It is also a real-world application that deploys and coordinates contemporary web technology. The main reason for using web technoloy is that WFMSs are by nature distributed, multi-user, and heterogeneous software systems. The iTask system is a library made in the functional programming language Clean. The specifications that serve as input to the iTask system are expressed as a domain specific language embedded in Clean. We have adopted the practice in the functional programming community to provide a library offering a set of *combinator functions* and *primitive functions* to allow for compositional, higher-order, parameterized model specifications.

In order to give an impression of the combinators that a workflow engineer can use, Fig. 1 shows a few of the combinator functions and types that constitute the iTask domain specific language (for reasons of presentation, the types have been slightly simplified).

```
:: Task a          // Task is an opaque, parameterized type constructor

// Sequential composition:
(>>=) infixl 1  :: (Task a) (a → Task b)    → Task b   | iTask a & iTask b
return          :: a                        → Task a   | iTask a

// Splitting-joining any number of arbitrary tasks:
anyTask         :: [Task a]                 → Task  a  | iTask a
allTasks        :: [Task a]                 → Task [a] | iTask a

// Task assignment to workers:
class   (@:) infix 3 w :: w (String,Task a) → Task a   | iTask a
instance @: User, String
```

Figure 1: A snapshot of the iTask combinator functions.

A task is an expression of the opaque (hidden), parameterized type `Task a`. Here, `a` is a type parameter that can be instantiated with any conceivable first order type. It represents the type of the value that is produced by the task. Hence, a task (expression) of type `Task a` is a task that, once it has been performed, produces a value of type `a`.

Tasks can be combined *sequentially*. The infix combinator `>>=` and `return` function are the standard *monad* combinators [11]. Task `t >>= f` first performs task `t`, which eventually produces a value of type `a`. This value can be used by the *function argument* `f`, which can compute any new kind of task expression based on that information. The type demands that `f` eventually produces a value of type `b`, which is also the final result of `t >>= f`. The task `return v` only produces value `v` without any effect.

Any number of tasks `ts` = $[\mathtt{t}_1 \ldots \mathtt{t}_n]$ ($n \geq 0$) can be performed in parallel

and synchronized (also known as *splitting* and *joining* of workflow expressions): `anyTasks ts` and `allTasks ts` both perform all tasks `ts` simultaneously, but `anyTasks` terminates as soon as *one* task of `ts` terminates and yields its value, whereas `allTasks` waits for completion of *all* tasks and returns their values.

Tasks can be assigned to workers. The expression `w @: (l,t)` assigns task `t` to worker `w`. Here `l` is a descriptive label (like the subject field in an e-mail message). The infix operator `@:` is overloaded in the identification value of the worker, which can be a value of type `User` (a predefined iTask type), or by means of the user name (`String` value).

A more detailed description of these combinators is out of scope of this paper, but in Sect. 3 we give a complete example of a small, yet realistic and complex workflow that uses many of the above combinators. The crucial points are that first, all combinator functions are parameterized and statically type checked with the data that flows along the tasks. Second, tasks can inspect this data and change the control flow accordingly. Third, there is no limit on the type of the data that is passed along, provided that suitable generic functions (see Sect. 5) are available. This is expressed by means of the type class context restrictions (`| iTask ...`). Fourth, several combinators to express iteration are included in the iTask library. However, because the iTask system is a library embedded in Clean, the workflow engineer can define new combinators and even define recursive workflows if desired.

In addition to combinators that combine task expressions in new ways, the workflow engineer also needs primitive iTask functions. Fig. 2 shows some.

```
// Worker interaction:
enterInformation   :: question          →Task a      | html question & iTask a
updateInformation  :: question a        →Task a      | html question & iTask a
showMessage        :: message           →Task Void   | html message
chooseTask         :: question [Task a] →Task a      | html question & iTask a

// Worker administration:
chooseUsersWithRole:: question String   →Task [User] | html question
```

Figure 2: A snapshot of the iTask primitive combinator functions.

The archetypical primitive iTask combinator is `enterInformation q` which, when performed, presents the current worker with a form to create a new value of type `a`. Here, `q` is a guiding prompt for the worker. Fig. 3 gives an example of a form for the type `Person`. `updateInformation q v` is similar, except that the value `v` acts as initial content of the form. The `showMessage` combinator displays a message to the user. With `chooseTask` the user can choose a task to be performed from a list of tasks. In order to dynamically delegate work to users in the system, a workflow needs to have access to the worker administration. With the combinator function `chooseUsersWithRole` the user is given a list of current workers, and she can make a selection.

The overview of the iTask combinators here is just a selection enabling us

```
:: Person = { firstName   :: String
            , surname     :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Male | Female

enterPerson :: Task Person
enterPerson = enterInformation "Enter Information"
```

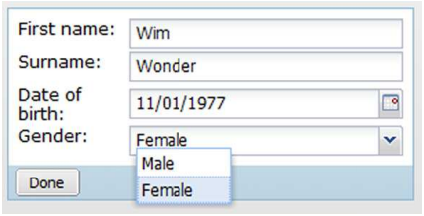


Figure 3: A standard form editor generated for type `Person`.

to present the example used in Sect. 3. There are many more combinators that we cannot discuss here due to lack of space: combinators for the dynamic creation and control of workflow *processes*, combinators to raise and handle *exceptions* (stop a running workflow, inform all collaborators and start an alternative workflow), and combinators which allow to *change* workflows *at execution time* (replace a workflow on-the-fly by another workflow yielding a result of the same type). These features are necessary to handle realistic workflow cases.

Finally, iTask is embedded in Clean. This provides the workflow engineer with many abstraction techniques that are common practice in functional programming: tasks can be polymorphic, use higher-order functions, can be parameterized, and even higher-order workflows can be created (tasks that have tasks as parameter or result). This yields a high degree of reusability and customization. As a final example, iTask provides a core combinator function, `parallel` that is used in the system to define many other split-join combinators such as `anyTask` and `allTasks` that were shown earlier. Its type signature is:

```
parallel :: ([a] → Bool) ([a] → b) ([a] → b) [Task a] → Task b | iTask a & iTask b
```

`parallel c f g ts` performs all tasks within `ts` simultaneously and collects their results. However, as soon as the predicate `c` holds for any current collection of results, then the evaluation of `parallel` is terminated, and the result is determined by applying `f` to the current list of results. If this never occurs, but all tasks within `ts` have terminated, then `parallel` terminates also, and its result is determined by applying `g` to the list of results.

3 Ordering example

To demonstrate the expressive power of iTask, we present an *ordering* example. The code presented below is a complete, executable, iTask workflow. The workflow has a recursive structure and monitors intermediate results in a parallel and-task. This case study is hard to express in traditional workflow systems. The overall structure contains the following steps (see `getSupplies` below): first, an inventory is made to determine the required amount of goods (`getAmount`) (e.g. vaccines for a new influenza virus); second, suppliers are asked in parallel how

much they can supply (`inviteOffers`); third, as soon as sufficient goods can be ordered, these orders are booked at the respective suppliers (`placeOrders`).

```
getSupplies :: Task [Void]                                                    1.
getSupplies = getAmount >>= inviteOffers >>= placeOrders                      2.
```

Determining the required amount of goods proceeds in a number of steps:

```
getAmount :: Task Amount                                                      3.
getAmount                                                                     4.
= chooseTask "Decide how much we need"                                        5.
      ["Decide yourself"   @>> enterInformation "Enter the required amount"   6.
      ,"Let others decide" @>> determineOthers]                               7.

determineOthers :: Task Amount                                                8.
determineOthers                                                               9.
=                 chooseUsersWithRole "Select institutes:" "Institute"        10.
   >>= λusers →  allTasks [  user @: ("Amount request", getAmount)            11.
                          \\ user ← users                                     12.
                          ]                                                   13.
   >>= λothers → updateInformation "Enter required amount" (sum others)       14.
```

First, with `chooseTask` the user can choose to enter the amount herself or to ask others to determine this amount. `@>>` is used to give a task a (displayable) label. In `determineOthers`, with the task `chooseUsersWithRole` (line 10) a set of users (of type `User`) which fulfil a certain role, in this case institutes, is selected by the user. Each of the selected institutes on their turn may enquire other institutes *recursively* in parallel (using the `allTasks` combinator) how many goods they need (lines 11-13). The recursive call `getAmount` has as effect that each of the chosen institutes can ask other institutes for the same thing, and so on. Given the amount determined by others, an institute may alter the final amount it wants to have (line 14). `Amount` is a non-negative `Int`:

```
:: Amount :== Int                                                             15.
```

Once the amount of goods is established, the workflow can continue by inviting offers from a collection of candidate suppliers:

```
inviteOffers :: Amount → Task [(Supplier,Amount)]                             16.
inviteOffers needed                                                           17.
=              chooseUsersWithRole "Select suppliers:" "Supplier"             18.
  >>= λsups →  parallel enough (maximum needed) id                            19.
                [sup @:("Order request", updateInformation prompt needed      20.
                                         >>= λa → return (sup,a))             21.
                \\ sup ← sups                                                 22.
                ]                                                             23.
where enough as = sum (map snd as) >= needed                                  24.
      prompt    = "Request for delivery, how much can you deliver?"           25.
```

This collection is determined first (line 18). Each supplier can provide an amount (line 20). This is again done in parallel (line 19-23). The termination criterium is the `enough` predicate which is satisfied as soon as the sum of

provided offers exceeds the requested amount (line 24). The canonization function `maximum` is discussed below. Hence, the result of this task is a list of offers. Each offer is a pair of a supplier and the amount of goods that it offers to deliver. A supplier is just a user:

```
:: Supplier :== User                                                              26.
```

The total number of offered goods can differ from the required number of goods. The function `maximum` makes sure that not too many goods are ordered.

```
maximum :: Amount [(Supplier,Amount)] → [(Supplier,Amount)]                       27.
maximum needed offers   = [(sup,exact) : rest]                                    28.
where                                                                             29.
    [(sup,_) : rest]    = sortBy (λ(_,a1) (_,a2) → a1 > a2) offers                30.
    exact               = needed - sum (map snd rest)                             31.
```

With the correct list of offerings, we can place an order for each supplier. This can be expressed directly with `allTasks`:

```
placeOrders :: [(Supplier,Amount)] → Task [Void]                                  32.
placeOrders offers                                                                33.
=   allTasks [sup @: ("Order placement", showMessage ("Please deliver " <+ a))    34.
             \\ (sup,a) ← offers                                                  35.
             ]                                                                    36.
```

The overloaded infix operator `<+` converts its right-hand argument to a string and glues it to the given left-hand argument. It is part of the iTask system.

In order to complete the case study, the `getSupplies` workflow needs to be passed to the iTask run-time system as a workflow that returns `Void`:

```
Start :: *World → *World                                                          37.
Start world  = startEngine [workflow] world                                       38.
where                                                                             39.
    workflow = { name     = "Ordering example"                                    40.
               , label    = "Collect ordering info and make the order"            41.
               , roles    = []                                                    42.
               , mainTask = getSupplies >>= λ_ → return Void                      43.
               }                                                                  44.
```

4 Experience with the iTask language

iTask is a prototype language. We have investigated its expressiveness by means of constructing examples as well as larger case studies, for instance a conference management system [13]. The next step is to investigate its use in demanding environments that concern crisis management situations, in a project with the Netherlands Defense Academy. In this section we report on our experience in using the iTask specification language.

iTask is built on a single, powerful, concept

In iTask, everything is constructed as (a combination of) a task. The notion of a task and the combinators we use have a clear semantics [7]. A task represents work that needs to be performed, and abstracts over the way the task is composed out of sub-tasks and the order in which these sub-tasks are being evaluated. No matter how complex a task may be, for the programmer a task remains a unit of work returning a value of type (`Task a`) once the task as a whole is terminated. The result of a task can be used as input for other tasks. The coordination of tasks is defined by means of combinators.

A task represents work that needs to be performed. This work can be anything that is required by the workflow case, such as connecting to a legacy information system, calling a web service, or arbitrary foreign code. For instance, for access to information stored in standard information systems, we have developed a systematic conversion between an information model defined in e.g. ORM (Object Role Model) and Clean data type definitions. This enables the automatic conversion between values of these types and the corresponding values stored in a relational database [8], without the need for explicit SQL programming. As another example, for the type `GoogleMap`, the basic task `enterInformation`

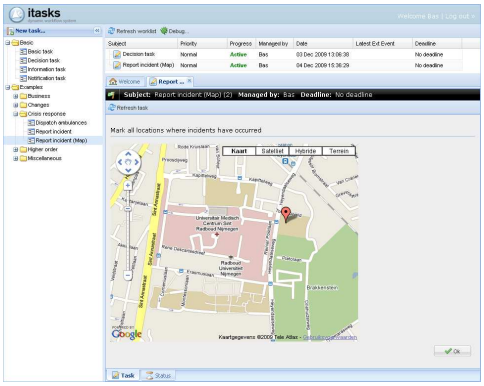


Figure 4: An iTask for manipulating a map

will show a standard Google Map in which the end user can scroll and place markers (Fig. 4). User manipulations of the map are automatically kept track of and are reflected in the `GoogleMap` data structure. No extra effort is needed in the workflow specification other than using the type.

In this way, everything can be considered to be a task. An iTask specification uses combinators to coordinate tasks, and hence one can use the iTask language as a web coordination language as well.

iTask is a declarative language

We want the specification of a workflow to be declarative and hence to abstract from details as much as possible. Given an iTask workflow specification, the iTask system automatically generates all required web forms, handles all user data entry, storage of intermediate results, task distribution to specified workers, and handles all coordination. Also the precise way information is displayed in the browser is not specified in the workflow, but delegated to the client. To further enable abstraction over lay-out, we offer several primitives in the iTask library for basic interaction steps. For instance, in addition to `enterInformation`, there are basic primitives like `enterChoice` and `enterMultipleChoice`. The advantage of having different primitives for such basic interaction steps is that the workflow specification becomes more readable while the representation and lay-out can again be delegated to the client. Due to abstraction, the workflow engineer can concentrate on specifying the workflow. This promotes rapid prototyping of workflow applications.

iTask is more than Clean

iTask is an embedded domain specific language and inherits all language aspects of its host Clean. In particular, these are the strong type system, higher-order functions, lazy and strict evaluation, and the module system. All computational and algorithmic concerns can be dealt with in the Clean language. iTask is also more than Clean because workflows are inherently sequential, distributed, multi-user, concurrent systems and the Clean standard supports neither of those. Also, to model realistic workflow cases, one needs to address exceptions and dynamic change. Again, these concepts are absent in native Clean (see also Sec. 5). Each of the required concepts of the embedded language are challenging to add to native Clean. Nevertheless, this experiment shows that it is possible to embed a workflow language in a host that offers entirely different concepts.

iTask has higher-order tasks

A task in Clean of type `Task a | iTask a` effectively works for all first order types `a`. In particular, it works for the type `Task` itself, which means that tasks can be higher order: the result of a task might be a task which can be dynamically and interactively constructed. In this way meta programming (doing tasks that have as goal to define new tasks) can be accomplished. A task thus created can be given as argument to other tasks which can decide to evaluate it or to use it in the construction of an even more complex task. It is very unlikely that an ad-hoc domain specific workflow language has the ability to deal with advanced notions such as higher functions and tasks, and this feature is therefore missing in all commercial workflow systems. Embedding a workflow language in a language like Clean really pays off here.

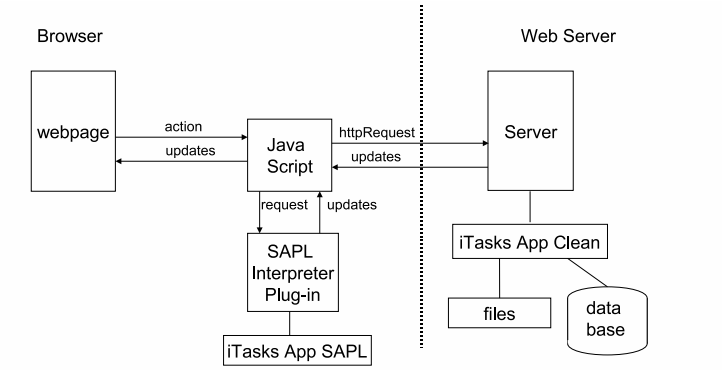


Figure 5: The architecture of an iTask application

5 Experience with Clean as host language

In this section we focus on our experience with using Clean as host language and implementation vehicle to embed iTask. An iTask specification results in a web application. The architecture of this web application is given in Fig. 5.

Smart combinators

iTask is a workflow language and is hence inherently sequential, distributed, multi-user, and concurrent. It needs to handle exceptional situations and dynamically changing workflows. The host language Clean offers no native support for these concepts. When developing such a language in the traditional way, one would develop a grammar, semantic rules, perhaps a type system, a compiler and/or interpreter, code generator, and so on. This is a huge amount of work. In this project we have taken a different route: when designing a language, one needs to define the semantic rules. Semantic rules can be represented in a natural way by means of functions. If one takes care in designing these rules in a compositional way, then these form a set of *smart combinator* functions. In this way one can obtain a compositional language implementation almost for free. This decreases the implementation effort of a new language significantly.

The combinators have several obligations in the iTask system. First, the combinators yield the current status (and hence GUI) at any moment during execution. For example, the iTask system can evaluate the expression `t >>= f` even if task `t` is not finished yet. The iTask system does this by creating a default value of the proper type for the whole expression `t >>= f`. In this way the status of all tasks defined in a workflow can be inspected, but only the values of the finished tasks are taken into account. Second, a new workflow is calculated by the combinators given the finished tasks. Third, each combinator stores its current state in memory and uses it for handling the next event from the participating workers.

Smart tasks

The iTask language is a declarative language. This implies that we want to generate as much boilerplate code as can be possibly done from an iTask specification. In iTask this has been realized by using the generic programming features of Clean [2]. Tasks require the availability of a collection of generic (kind indexed, type driven) functions. These generic functions are used to generate all kinds of functionality automatically, such as the generation of web forms, the handling of user updates of such forms, the storage and retrieval of information, the serialization and de-serialization of data and functions. The generic functions are predefined in the iTask library. To use them for a certain type, however, one needs instances for that type for all the generic functions being used. As a result a task can be applied to values of *any* type, as long as instances for this type have been defined for all generic functions the task is depending on. The Clean compiler is able to generate instances for these generic functions for (almost) any (non opaque) type fully automatically. Clean is special in this respect. In Haskell e.g. generic functions can be constructed using special pre-processors like template Haskell.

It should be noted that a great deal of the facilities for which we have used generics in our project can be done in a programming language that offers introspection and code generation facilities. One significant advantage of using generics is its firm integration with the static type system of Clean.

Smart serialization

An iTask application is a web application that runs on the server side. This application must handle every possible user request from any possible web browser that connects with the application. After an event is handled, the web application terminates and is started all over again by the web server when new user events arrive. Hence, an iTask application needs to fully recover its previous state to compute the proper response. Conceptually, this amounts to reconstructing the *task tree* that reflects the current state of computation of the workflow. The *nodes* of a task tree are formed by the combinators in the task that is being computed, and the *leaves* of a task tree are the primitive tasks. Evaluation of a workflow amounts to *rewriting* this task tree as dictated by the combinators. The task tree can become very big. Hence, a naive implementation of task tree rewriting for iTask applications is not realistic. Instead, we have incorporated a number of optimizations that are required to obtain an efficient and scalable implementation. We briefly discuss two of the most important optimizations.

The first optimization is based on the observation that most rewrites affect only a local part of the task tree. Hence, for these rewrites it is not necessary to reconstruct the entire task tree, but only the sub task tree that can be affected. Because an iTask application terminates after handling an event, we need to be able to store and read any sub tree that is currently being rewritten. Tasks and combinators are implemented as state transition functions, hence we need to be able to store functions. Clean offers a hybrid type system, and statically typed

expressions can be turned into a dynamically typed expression (of static type `Dynamic`) and the other way around. Dynamics can be stored to disk and it is even possible to read in a dynamic stored by some other Clean application.

The second optimization is based on the observation that many computations do not *have to be done* at the server side, but can also be done on the web client side. Hence, clients need to be able to run tasks, which amounts to running Clean code. To implement this, the Clean compiler generates *two* executable instances from a single source. The first instance is a Clean executable that runs on the server, and the second instance is a SAPL program to be executed by the SAPL interpreter [6] that is running as a Java applet at the client side. At run-time it can be decided where to execute what. Any function or task can be shifted from server to client. For this purpose we again use dynamics in Clean to serialize functions and expressions as SAPL programs at the server side and interpret them at the client side. For details we refer to [14].

6 Related work

The WebWorkFlow project [5] shares our point of view that a workflow specification is regarded as a web application. WebWorkFlow is an object oriented workflow modeling language. *Objects* accumulate the progress made in a workflow. *Procedures* define the actual workflow. Their specification is broken down into *clauses* that individually control *who* can perform *when*, what the *view* is, what should be *done* when the workflow procedure is applied, and what further workflow procedures should be *processed* afterwards. Like in iTask, one can derive a GUI from a workflow object. The main difference is that iTask is embedded in a functional language, but this has significant consequences: iTask supports higher-order functions in both the data models and the workflow specifications; arbitrary recursive workflows can be defined; reasoning about the evaluation of an iTask program is reasoning about the combinators instead of the collection of clauses.

Brambilla *et al*[4] enrich a domain model (specified as UML entities) with a workflow model (specified as BPMN) by modeling the workflow activities as additional UML entities and use OCL to capture the constraints imposed by the workflow. The similarity with iTask is to model the problem domain separately. However, in iTask a workflow is a function that can manipulate the model values in a natural way, which enables us to express functional properties seamlessly (Sect. 3). This connection is ignored in [4] and can only be done ad-hoc.

Pešić and van der Aalst [10] base an entire formalism, ConDec, on linear temporal logic (LTL) constraints. Frequently occurring constraint patterns are represented graphically. This approach has resulted in the DECLARE tool [9]. In iTask a workflow can use the rich facilities of the host language for computations and data declarations – such facilities are currently absent in DECLARE.

Andersson *et al*[3] distinguish high level *business models* (value transfers between *agents*), low level *process models* (workflows in BPMN), and medium level *activity dependency models* (activities for value transfers of business mod-

els). Activities are *value transfer*, *assigning* an agent to a value transfer, value *production*, and *coordination* of mutual value transfers and activities. Activities are modeled as nodes in a directed graph. The edges relate activities in a way similar to [4] and [10]: they capture the workflow, but now at a conceptual level. A *conformance relation* is specified between a process model and an activity dependency model. Currently, there is no tool support for their approach. The activity dependency models provide a declarative foundation to bridge the gap between business models and process models. One of the goals of the iTask project is to provide a formalism that has sufficient abstraction to accomodate both business models and process models.

Vanderfeesten *et al*[15] have been inspired by the *Bill-of-Material* concept from manufacturing, recasted as *Product Data Model* (PDM). A PDM is a directed graph. Nodes are product data items, and arcs connect at least one node to one target node, using a functional style computation to determine the value of the target. A tool can inspect which product data items are available, and hence, which arcs can be computed to produce next candidate nodes. This allows for flexible scheduling of tasks. Similarities with the iTask approach are the focus on tasks that yield a data item and the functional connection from source nodes to target node. We expect that we can handle PDM in a similar way in iTask. iTask adds to such an approach strong typing of product data items (and hence type correct assembly) as well as the functions to connect them.

7 Conclusions

In this paper we report on our experience in using the lazy, pure, functional language Clean as embedding language to specify and create web-based workflow iTask applications. Although the iTask combinator language is embedded as a library in Clean, it is by no means a *shallow* embedding, i.e. the meaning of the embedded language is not a straightforward extension of the host language. The result *is* a new language for defining workflow applications. This new language provides the workflow engineer with concepts to seamlessly merge data flow with control flow (exemplified by the >>= combinator), use higher-order tasks (tasks that can create, manipulate, and pass around tasks), in a compositional way. The evaluation order of the workflow is controlled by the iTask combinators and dictated by the needs of the workflow engineer (by using sequential and generalized parallel split-join patterns as well as recursion). It is important to observe that this evaluation order is very different from the lazy evaluation order of the host language *and* that one can add new combinators within iTask to capture other evaluation orders when needed. The iTask system is very general and serves as a coordination language to control and unify all tools that are used to realize the system. Specifications inherit the terseness of their host language.

We have used many state-of-the-art programming language techniques to obtain this result: *generic programming* to handle boilerplate code generation (including foreign code) in a type-directed way, *dynamic types* to handle arbitrary (higher-order) data structures which origin need not be the source program

itself, and *higher-order functions* which permeate through the entire design, implementation, and resulting language. The entire system is statically typed. Although the boilerplate code generation aspects can be realized in other programming languages that support some form of inspection, we have shown in this project that the task of embedding a language (however alien) is one that fits functional programming languages like a glove.

References

[1] Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Ana Barros. Workflow patterns. QUT technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.

[2] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, September 2002.

[3] Birger Andersson, Maria Bergholtz, and Ananda Edirisuriya. A Declarative Foundation of Process Models. In Oscar Pastor and João Falcão e Cunha, editors, *Proceedings 17 Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005*, volume 3520 of *LNCS*, pages 233–247. Springer-Verlag, 2005.

[4] Marco Brambilla, Jordi Cabot, and Sara Cornai. Automatic Generation of Workflow-Extended Domain Models. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings Model Driven Engineering Languages and Systems, 10th Int'l Symposium, MoDELS 2007*, volume 4735 of *LNCS*, pages 375–389. Springer-Verlag, 2007.

[5] Zef Hemel, Ruben Verhaaf, and Eelco Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*, pages 113–127. Springer-Verlag, 2008.

[6] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In Henrik Nilsson, editor, *Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.

[7] Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In Sven-Bodo Scholz, editor, *Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, 2009. To appear in Springer LNCS.

[8] Bas Lijnse and Rinus Plasmeijer. Between types and tables - Using generic programming for automated mapping between data types and relational databases. In Sven-Bodo Scholz, editor, *Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, 2009. To appear in Springer LNCS.

[9] Maja Pešić. *Constraint-based workflow management systems: shifting control to users*. PhD thesis, Technical University Eindhoven, 8, October 2008.

[10] Maja Pešić and Wil van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *Proceedings of the 1st Business Process Management Workshop on Dynamic Process Management, DPM'06*, volume 4103 of *LNCS*, pages 169–180. Springer-Verlag, 2006.

[11] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th International Symposium on Principles of Programming Languages, POPL'93*, pages 71–84, Charleston, SC, USA, January 1993. ACM Press.

[12] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP'07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.

[13] Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort. An iTask case study: a conference management system. In *Selected Lectures of the 6th International Summer School on Advanced Functional Programming, AFP'08*, volume 5832 of *LNCS*, Center Parcs "Het Heijderbos", The Netherlands, 19-24, May 2008. Springer-Verlag.

[14] Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP'08*, pages 56–66, Valencia, Spain, 15-17, July 2008.

[15] Irene Vanderfeesten, Hajo Reijers, and Wil van der Aalst. Product based workflow support: dynamic workflow execution. In Z. Bellahsène and M. Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering, CAiSE'08*, volume 5074 of *LNCS*, pages 571–574, Montpellier, France, 2008. Springer-Verlag.

[16] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *Selected Papers of the 14th International Symposium on the Implementation of Functional Languages, IFL'02*, volume 2670 of *LNCS*, pages 101–117. Springer-Verlag, September 2003.

Specifying Generic Java Programs: two Case Studies

A. Giorgetti[1,2], C. Marché[3,4], E. Tushkanova[2,1], and O. Kouchnarenko[1,2]

[1]LIFC, Univ. of Franche-Comté, Besançon F-25030
[2]INRIA Nancy - Grand Est, Villers-lès-Nancy F-54600
[3]INRIA Saclay - Île-de-France, Orsay F-91893
[4]LRI, Univ. Paris-Sud, Orsay F-91405

Research paper

Abstract

This work investigates the question of modular specification of *generic* Java classes and methods. We propose extensions to the *Krakatoa Modeling Language*, a part of the Why platform for proving that a Java or C program is a correct implementation of some specification. The new constructs we propose for the specification of generic Java programs are presented through two significant examples: the specification of the generic method for sorting arrays which comes from the `java.util.Arrays` class in the Java API, and the specification of the `java.util.HashMap` class defining a generic hash map and its use for memoization. The main ingredient is the notion of *theories* and the instantiation relation between them. We discuss soundness conditions and their verification.

1 Introduction

The program verification problem is to verify the Hoare triple, $\{P\}\ S\ \{Q\}$, where P and Q are assertions and S is a program statement. P is called the precondition and Q the postcondition: if the precondition is met and the statement terminates, then it should establish the postcondition.

The Why platform [4] is a set of tools for deductive verification of Java source code. From a source program annotated by specifications, the Why platform extracts verification conditions and transmits them to provers like *SMT solvers* (Simplify, Z3, CVC3, Yices, Alt-Ergo, etc.) or *proof assistants* (Coq, Isabelle/HOL, PVS, etc.) For Java, these specifications are given in the Krakatoa Modeling Language (KML) [7], similar to the popular Java Modeling Language (JML) [2, 5].

A new feature starting from Java 1.5 is *generic* classes and methods. These are not yet supported neither by JML nor KML. We aim at addressing the question of *modular* specification of generic Java classes and methods by extending KML with generic specification constructs. In order to identify these constructs, we consider examples of generic Java code.

This paper presents a complete generic specification of two significant examples. The former is the specification of the generic method for sorting arrays which comes from the `java.util.Arrays` class in the Java API. The latter is the specification of the `java.util.HashMap` class and its use for memoization. This proposal clearly shows that the following two extensions are required:

- add *type parametricity* in the specification language;
- add a notion of *instantiation* of *theories* used to model programs.

The paper is organized as follows. Section 2 presents an excerpt of the constructs for specifying Java programs. We focus on the KML-specific features which allows to specify algebraic-style data types [6] and theories, for which we propose an extension. Section 3 presents new specification constructs for specifying a generic method for sorting arrays. Section 4 presents additional constructs needed when specifying generic hash maps. Section 5 discuss the verification process.

2 Specification Language

A specification language is a formal language used during requirement analysis and system design. There are algebraic-based specification languages like CASL, Z, B ; and program-oriented ones like JML for Java, Spec# for C#. KML is a specification language for Java, inspired from JML, while sharing many features with the ANSI/ISO C Specification Language [1]. The principal difference between JML and KML is that the main application of JML is *runtime assertion checking*, whereas KML is designed for automated or assisted *deductive verification*, by statically producing verification conditions. For this reason, KML was designed to allow algebraic-style specifications because they are suitable for theorem proving.

2.1 Basic standard features

Specifications are given as annotations in the source code, in a special style of comments after `//@ ...` or between `/*@` and `*/`.

Method *contracts* are made of a *precondition* and a set of *behaviors*, under the general form

```
/*@ requires R;
    behavior b:
      assigns L; ensures E; */
```

where b is the behavior name, R and E are logical assertions. R is a *precondition*, E is a *postcondition* and L is a set of memory locations, that may be modified by the method. The precondition R is supposed to hold in the method pre-state, for any value of its arguments. It must be checked valid by the caller. The postcondition E must be established by the method's code at the end of its execution. In that formula, `\result` denotes the returned value.

A *class invariant* is declared at the level of class members. It has the form

`//@ invariant` id`:` p`;`

where p is a property that must be established by each constructor, and preserved by each method of the class. A *model field* is a field introduced in the specification with the keyword `model`, and is related to concrete fields with an invariant. Its type must be a logic type. The model fields are used to provide abstract specifications to functions whose concrete implementation must remain private.

Inside Java code, an assertion

`//@ assert` p`;`

specifies that the property p is true at the corresponding program point. The construct `\at`(e, L) refers to the value of the expression e in the state at label L. There exist predefined labels, e.g. `Old` and `Here`. `\old`(e) is in fact syntactic sugar for `\at`$(e,$`Old`$)$. The label `Here` is visible in all statement annotations, where it refers to the state where the annotation appears. It refers to the pre-state in a method precondition (`requires` clause), and to the post-state in a method postcondition (`ensures` clause). The label `Old` is visible in `ensures` clauses and refers to the pre-state of the method's contract. More details can be found in [7].

2.2 Logical specifications

KML does not allow pure methods to be used in annotations but it permits to declare new logic functions and predicates. They must be placed at the global level, i.e. outside any class declaration, and respectively have the form

`//@ logic` t $id(t_1\ x_1, \ldots, t_n\ x_n)$ `=` e`;`
`//@ predicate` $id(t_1\ x_1, \ldots, t_n\ x_n)$ `=` p`;`

where e has type t, and p is a proposition. The types t, $t_1, \ldots, t_n$ can be either Java types or purely logic types. The logic types of mathematical integers and reals are built-in and respectively denoted `integer` and `real`.

Logic functions and predicates can also be *hybrid*. It means that they depend on some memory state. More generally, they can depend on several memory states, by attaching them several labels. The general form of a hybrid function and predicate definition respectively is

`//@ logic` t $id\{L_1, \ldots, L_k\}(t_1\ x_1, \ldots, t_n\ x_n)$ `=` e`;`
`//@ predicate` $id\{L_1, \ldots, L_k\}(t_1\ x_1, \ldots, t_n\ x_n)$ `=` p`;`

where $L_1, \ldots, L_k$ are memory state labels on which the function or predicate depends, and t, $t_1, \ldots, t_n$, e and p are as before.

A predicate may also be defined by an *inductive* definition of the form

```
/*@ inductive P{L_1, ..., L_k}(t_1 x_1, ..., t_n x_n) {
      case c_1 : p_1;
      ...
      case c_m : p_m;
    } */
```

where $c_1, \ldots, c_m$ are identifiers and $p_1, \ldots, p_m$ are propositions. The semantics of this definition is that P is the least fixpoint of the cases, i.e. the smallest predicate (in the sense that it is false the most often) satisfying the propositions $p_1, \ldots, p_m$. To ensure existence of a least fixpoint, it is required that each of these propositions is of the form

```
\forall y_1, ..., y_m, h_1 ==> ... ==> h_l ==> P(t_1, ..., t_n)
```

where P occurs only positively in hypotheses $h_1, \ldots, h_l$.

Finally, a set of types, functions and predicates can be declared *axiomatically* by an algebraic-style axiomatization. Such a set forms a block of declarations called a *theory*. It has the following general form

```
theory Th {
    type id;
    logic t id{L_1, ..., L_k}(t_1 x_1, ..., t_n x_n);
    predicate id{L_1, ..., L_k}(t_1 x_1, ..., t_n x_n);
    axiom id_1 : p_1;
    axiom id_2 : p_2;
    ⋮
}
```

Arbitrarily many types, functions, predicates and axioms can be given. Notice that functions and predicate are only given by their profiles. Unlike inductive definitions, there is no syntactic conditions which would guarantee theories to be consistent. It is up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

3 Specification of a Generic Sorting Function

An array sorting function is a routine that modifies the order of elements in a given array. The resulting array must satisfy two properties:

1. The array elements are a permutation of the elements at the beginning.
2. The elements are in increasing order w.r.t. some ordering relation.

Filliâtre and Magaud [3] study several algorithms for sorting, and both specify and prove them correct with the Why tool, but only on the particular instance of an array of integers and the usual "less-than" order. The second condition is specified by a predicate `(sorted t i j)` which expresses that array `t` is sorted in increasing order between the bounds `i` and `j`. The first condition is specified by a predicate `(permut t t')` where `t` and `t'` are permutations of each other. They describe many ways to define such a predicate, but the best solution is to express that the set of permutations is the smallest equivalence relation containing the transpositions, i.e. exchanges of two elements. The predicate `(exchange t t' i j)` is defined for two arrays `t` and `t'` and two indexes `i` and `j` and the predicate `(permut t t')` is defined inductively for the following properties: reflexivity, symmetry and transitivity. The proofs are performed within the Coq proof assistant [9].

A selection sorting algorithm is written in Java by Marché [7] with a similar specification in KML. It is also specific to integers and the usual less-than order. The proof is done fully automatically with SMT solvers (namely Simplify and Alt-Ergo provers).

We present new specification constructs for specifying a generic method for sorting arrays, where the array elements are of any type T and the ordering is given as a parameter, under the form of a comparison function on T. As we will see, it is also important to study how generic specification can be used by *client* code, because it has to be instantiated.

3.1 Generic sorting in Java

The class `java.util.Arrays` defines a generic sorting method with the profile:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

In this method `<T>` is a type parameter and the syntax `<? super T>` denotes an unknown type that is a supertype of `T` (or `T` itself). The `java.util.Comparator<T>` interface imposes a total ordering on some collection of objects.

```
interface Comparator<T> {
  public int compare(T x, T y);
}
```

`T` is the type of objects that may be compared by this comparator. The method `compare` compares its two arguments for some order. It returns a negative integer, zero, or a positive integer when the first argument is respectively less than, equal to, or greater than the second one.

The simple program given on Figure 1 illustrates an instance of use of this `sort` method. In the `main` method, `intc` is an instance of the class `IntComparator` (Figure 2), which implements the interface `java.util.Comparator<T>` instantiated with the class `Integer` which wraps a value of the primitive type `int` in an object.

The client code ends with a simple assertion which we expect to be able to prove, as a consequence of the generic specification we will provide.

```
class Main {
 public static void main(String[] args) {
  IntComparator intc = new IntComparator();
  Integer[] b = {new Integer(2),new Integer(1),new Integer(3)};
  java.util.Arrays.sort(b,intc);
  //@ assert b[0].value <= b[1].value;
 }
}
```

Figure 1: Main class where the sorting method is called

```
class IntComparator implements Comparator<Integer> {
   public int compare(Integer x, Integer y) {
       if (x.intValue() < y.intValue()) return −1;
       if (x.intValue() == y.intValue()) return 0;
      return 1;
   }
}
```

Figure 2: Implementation of the comparison method on integers

3.2 Specifying the permutation behavior

Following [7], we introduce a hybrid predicate, defined inductively on Figure 3. The new and simple extension we need is the addition of a type parameter *T* to denote the type of array elements. The first behavior of the `sort` method is then given below.

```
/*@ behavior permuts: ensures Permut<V>{Old,Here}(a,0,a.length-1); */
public static <V> void sort(V[] a, Comparator<? super V> cmp);
```

```
predicate Swap<T>{L1,L2}(T a[], integer i, integer j) =
 \at(a[i],L1) == \at(a[j],L2) && \at(a[j],L1) == \at(a[i],L2) &&
 \forall integer k; k != i && k != j ==> \at(a[k],L1) == \at(a[k],L2);

inductive Permut<T>{L1,L2}(T a[], integer l, integer h){
 case Permut_refl{L}: \forall T a[], integer l h;
   Permut<T>{L,L}(a, l, h);
 case Permut_sym{L1,L2}: \forall T a[], integer l h;
   Permut<T>{L1,L2}(a, l, h) ==> Permut<T>{L2,L1}(a, l, h);
 case Permut_trans{L1,L2,L3}: \forall T a[], integer l h;
   Permut<T>{L1,L2}(a, l, h) &&
     Permut<T>{L2,L3}(a, l, h) ==> Permut<T>{L1,L3}(a, l, h);
 case Permut_swap{L1,L2}: \forall T a[], integer l h i j;
   l <= i <= h && l <= j <= h &&
      Swap<T>{L1,L2}(a, i, j) ==> Permut<T>{L1,L2}(a, l, h);
}
```

Figure 3: The permutation predicate

```
theory ComparatorTheory<T> {
  predicate eq{L}(T x, T y);
  axiom eq_ref{L}: \forall T a; eq{L}(a,a);
  axiom eq_sym{L}: \forall T a b; eq{L}(a, b) ==> eq{L}(b,a);
  axiom eq_trans{L}: \forall T a1 a2 a3;
    eq{L}(a1, a2) && eq{L}(a2,a3) ==> eq{L}(a1,a3);

  predicate sto{L}(T x, T y);
  axiom sto_irref{L}: \forall T a; ! sto{L}(a,a);
  axiom sto_antisym{L}: \forall T a1 a2;
    ! (sto{L}(a1,a2) && sto{L}(a2,a1))
  axiom sto_totality{L}: \forall T a1 a2;
    eq{L}(a1,a2) || sto{L}(a1,a2) || sto{L}(a2,a1);
  axiom sto_trans{L}: \forall T a1 a2 a3;
    sto{L}(a1,a2) && sto{L}(a2,a3) ==> sto{L}(a1,a3);

  predicate to{L}(T x, T y) = eq{L}(x,y) || sto{L}(x,y);

  predicate sorted{L}(T[] a, integer l, integer h) =
    \forall integer i; l <= i <h ==> to{L}(a[i],a[i+1]);
}
```

Figure 4: General theory for `Comparator`s

```
interface Comparator<U> /*@ <Th instantiating ComparatorTheory<U> > */ {

  /*@ ensures (\result == −1 <==> Th.sto(x,y)) &&
              (\result == 0 <==> Th.eq(x,y)) &&
              (\result == 1 <==> Th.sto(y,x));    */
  public int compare(U x, U y);
}
```

Figure 5: Specification of the `Comparator` interface

3.3 Specifying the sorting behavior

Unlike the permutation behavior, specifying the sorting behavior is significantly more complex in the generic case than in the non-generic case. The first step is to define a general theory for types equipped with an ordering relation. Figure 4 shows a theory named `ComparatorTheory` which defines two predicates `eq` for equality and `sto` for a strict total order. Equality is reflexive, symmetric and transitive. The strict total order satisfies four properties: irreflexivity, antisymmetry, totality and transitivity.

The `Comparator` interface should take some comparison theory as a parameter. This is shown on Figure 5. The `Comparator` interface thus has two parameters: a `Java` type `U` and a theory `Th`. The syntax `Th instantiating ComparatorTheory<U>` says that `Th` is an instance of the general theory defined in Figure 4.

```
/*@ behavior sorts:
      ensures th.sorted(a,0,a.length−1); */
public static <V> /*@ <W> <th instantiating ComparatorTheory<W> > */
  void sort(V[] a, Comparator<? /*@ as W */ super V> /*@ <th> */ cmp) {
}
```

Figure 6: Specification of the generic sorting method

3.4 Specifying the sorting method

Figure 6 presents the specification of the method `sort` which tells that the array is sorted.

The sorting method takes an array `a` and a comparator `cmp` as parameters. It is not only parameterized by the type `V` but also by the type `W` which denotes the super type of `V` on which the comparator operates, and by a theory which can be any instance of the general `ComparatorTheory` on `W`. Notice the new `as` keyword added to relate the anonymous Java type denoted by `?` and the explicit name `W` we need to introduce for it in the specification.

The comparator type is itself instantiated with the Java type `W` and the theory `th`. In the method postcondition the predicate `sorted` is then qualified with this theory.

3.5 Instantiating the sorting method

To deal with our client program, we need more annotations. First we need to consider a specified version of the `java.lang.Integer` class. Figure 7 shows an excerpt of it annotated in KML. Notice that the private field is visible in the annotations of the public methods. In JML, the field should be annotated with modifier `spec_public` to allow that. In KML, private fields are automatically visible in annotations.

We then need to specify the `IntComparator` class of Figure 2. For that, we first provide a theory which instantiates the general comparator theory, in

```
public final class Integer extends Number implements Comparable {
    private int value;

    /*@ assigns this.value;
        ensures this.value == v; */
    public Integer(int v) { this.value = v; }

    /*@ assigns \nothing;
        ensures \result == this.value; */
    public int intValue() { return this.value; }
}
```

Figure 7: Annotated `Integer` class

```
theory IntComparatorTheory instantiates ComparatorTheory<Integer> {
  predicate eq{L}(Integer x, Integer y) = \at(x.value == y.value, L);
  predicate sto{L}(Integer x, Integer y) = \at(x.value < y.value,L);
}
```

Figure 8: Theory for integer comparison

```
class IntComparator
  implements Comparator<Integer> /*@ <IntComparatorTheory> */ {

  public int compare(Integer x, Integer y) {
    ...
  }
}
```

Figure 9: Specification of the `IntComparator` class

Figure 8. The `instantiates` declaration generates verification conditions, this is discussed below in Section 5.

The class `IntComparator` shown in Figure 9 implements the instantiation of the `Comparator` interface where the `Java` type is the `Integer` class and the theory is the comparison theory for this class, defined in Figure 8.

Finally, notice that when checking the specific call to method `sort` in client program, the "implicit" parameters `V`, `W` and also the theory `Th` must be guessed. The value of `V` comes as usual with the `Java` typing, the value of `W` comes from the type of the `Comparator`. Then the theory `Th` must be inferred from the theory argument of `cmp` and checking whether it really instantiates a convenient comparator theory.

The specifications are now complete, and should allow to prove the final assertion of our client program (see Section 5).

4 Specification of a Generic Hash Map

We present additional constructs needed when specifying generic *hash maps*. These are data types which build finite mappings from indexes of some type *key* to values of some other type *data*. Finding the value associated to a given index is made efficient by use of classical hashing techniques.

Among the CeProMi (`http://www.lri.fr/cepromi`) collection of challenging examples, there is a simple but illustrating example of use of hash maps: a method for computing *Fibonacci numbers*: $F(0) = 0$, $F(1) = 1$, and $F(n+2) = F(n+1) + F(n)$ for $n \geq 0$. To avoid the exponential complexity of the naive recursive algorithm, we apply the general technique of *memoization*. (This is just for illustration, since there exist other efficient ways to compute Fibonacci numbers.)

A `Java Fib` class with a `fib` method computing Fibonacci numbers with memoization is shown on Figure 10.

```
import java.util.HashMap;

class Fib {
    HashMap<Integer,Long> memo;

    Fib() { memo = new HashMap<Integer,Long>(); }

    public long fib(int n) {
        if (n <= 1) return n;
        Integer n_obj = new Integer(n);
        Long x = memo.get(n_obj);
        if (x == null) {
            x = new Long(fib(n−1)+fib(n−2));
            memo.put(n_obj,x);
        }
        return x.longValue();
    }
}
```

Figure 10: Java source for Fib class

4.1 Specification of the Fibonacci sequence

A mathematical definition of the Fibonacci sequence as a theory is given below.

```
theory Fibonacci {

  logic integer math_fib(integer n);
  axiom fib0: math_fib(0) == 0;
  axiom fib1: math_fib(1) == 1;
  axiom fibn: \forall integer n; n >= 2 ==>
       math_fib(n) == math_fib(n-1) + math_fib(n-2);
}
```

The expected behavior of the `fib` method is specified as follows.

```
/*@ requires n >= 0;
  @ assigns \nothing;
  @ ensures \result == math_fib(n); */
long fib(int n);
```

Notice that issues related to arithmetic overflow are ignored. We just assume for simplicity that computations are made on unbounded integers.

4.2 Theories for hashable objects and hash maps

The first step is to define a theory which provides a predicate for testing equality, and a hash function. This theory is given on Figure 11. The essential part of this theory is the axiom `hash_eq` which specifies the expected property for the hash function: two equal objects must have the same hash code.

```
theory HashableTheory<T> {

  predicate eq{L}(T x, T y);
  axiom eq_refl{L}: \forall T a; eq{L}(a,a);
  axiom eq_sym{L}: \forall T a b; eq{L}(a, b) ==> eq{L}(b,a);
  axiom eq_trans{L}: \forall T a1 a2 a3;
    eq{L}(a1, a2) && eq{L}(a2,a3) ==> eq{L}(a1,a3);

  logic integer hash{L}(T x);
  axiom hash_eq{L}: \forall T x,y;
    eq{L}(x,y) ==> \at(hash(x) == hash(y),L);
}
```

Figure 11: Theory of hashable objects

```
theory Map<K><Th instantiating HashableTheory<K> > {

  type t<V>;
  logic <V> V acc{L}(t<V> m, K key);
  logic <V> t<V> upd{L}(t<V> m, K key, V value);

  axiom <V> acc_upd_eq{L}: \forall t<V> m, K key1 key2, V value;
    Th.eq{L}(key1,key2) ==> \at(acc(upd(m,key1,value),key2) == value,L);

  axiom <V> acc_upd_neq{L}: \forall t<V> m, K key1 key2, V value;
    ! Th.eq{L}(key1,key2) ==>
      \at(acc(upd(m,key1,value),key2) == access(m,key2),L);
}
```

Figure 12: Theory of maps

We then provide a theory for maps, as shown on Figure 12. This theory is parameterized by both a type `K` for the keys and a theory for equality and hashing of `K` objects. The type of data is not given as a parameter to the theory itself, but as a parameter `V` of the type of maps. This allows to use the same theory of maps for several instances of `V`. This theory is indeed the classical *theory of arrays* which is a typical theory supported by SMT provers. It is defined by a function `acc` to access the element indexed by some key, and a function `upd` which provide a so-called *functional update* of a map, returning a new map in which the element associated to some key is changed. The behavior of these two functions is axiomatized by the two axioms in Figure 12, which make an essential use of the equality predicate on keys. It has to be noticed that specifying the proper equality relation on keys is one of the issues in this specification, and our proposal of use of theories is an answer to this issue.

Building a hash map for type `K` of keys should be allowed only if `K` implements methods `equals` and `hashCode` in a way compatible with some theory instantiating `HashableTheory<K>`. This condition is expressed by the `Hashable`

```
interface Hashable /*@ as T */
          /*@ <Th instantiating HashableTheory<T> > */ {

    /*@ requires (o instanceof T);
      @ ensures \result == true <==> Th.eq(this,(T)o); */
    boolean equals(Object o);

    //@ ensures \result == Th.hash(this);
    int hashCode();
}
```

Figure 13: Interface for `Hashable` objects

```
class HashMap<K,V> /*@ <Th instantiating HashtableTheory<K> >
                    @ constraint: K implements Hashable<Th> */
{
 //@ theory M = Map<K>(Th);
 //@ model M.t<V> m;

  /*@ requires x instanceof K;
    @ ensures \result != null ==> \result == M.acc(m,(K)x) ;
    @*/
  V get(Object x);

  /*@ requires k != null;
    @ assigns m;
    @ ensures m == M.upd(\old(m),k,v);
    @*/
  void put(K k, V v);
}
```

Figure 14: Specification of the `HashMap` class

interface for hashable objects reproduced in Figure 13. The specification of the generic `java.util.HashMap` class is shown on Figure 14. A *constraint* is posed on the type of keys to relate it with a proper `HashableTheory`. Notice the use of local naming of a particular instance of a theory: the name `M` is given to the theory of `Map`s instantiated on the type of keys and on its theory of equality and hashing. This naming mechanism is certainly a construction that we should offer in practice.

4.3 Instantiating generic hash maps

The generic `HashMap` class being specified, we can use it in the `Fib` class. The first step is to provide an instance of the theory of equality and hashing on `Integer`s. This is done on Figure 15. A proper implementation of the `Integer` class is then given on Figure 16.

In order to prove the `fib` method behavior, it is mandatory to provide a

```
theory HashableInteger instantiates HashableTheory<Integer> {

  predicate eq{L}(Integer x, Integer y) = \at(x.value == y.value, L);
  logic integer hash{L}(Integer x) = \at(x.value, L);
}
```

Figure 15: Theory of equality and hashing of `Integer`s

```
class Integer implements Hashable /*@ <HashableInteger> */ {

    boolean equals(Object o) {
        if (o instanceof Integer)
            return this.value == ((Integer)o).value;
        return false;
    }

    int hashCode() { return this.value; }
}
```

Figure 16: Implementation of hashable `Integer`s

```
class Fib {
  HashMap<Integer,Long> /*@ <HashableInteger> */ memo;

  /*@ invariant memo_fib: \forall Integer x, Long y;
    @   y == memo.M.acc(memo,x) && y != null ==>
    @     y.value == math_fib(x.value);
    @*/

    /*@ requires n >= 0;
      @ assigns \nothing;
      @ ensures \result == math_fib(n);
      @*/
    long fib(int n) {
        ...
    }
}
```

Figure 17: Class invariant of the `Fib` class

class invariant which, informally, states that for any pair (x, y) stored in the `memo` map, $y = F(x)$. The class invariant for the `Fib` class can be written as in Figure 17.

5 Verification Conditions for Soundness

The soundness conditions for the first case study are concerned with the theory `IntComparatorTheory` which instantiates the theory `ComparatorTheory`-

<Integer> (Figure 8) and the interface IntComparator which implements the interface Comparator<U> (Figure 9). In fact, the definitions of the predicates eq and sto given in IntComparatorTheory should satisfy the axioms given in ComparatorTheory<T> when the type variable T is instantiated with Integer. This condition is easily discharged by SMT provers. On the other hand, the method compare defined in the IntComparator class should satisfy the specification of the method compare declared in the interface Comparator<U> when the theory parameter Th is instantiated with IntComparatorTheory. This condition is again easily proved by SMT provers (up to the question of null pointers that is not addressed here).

Similarly, soundness conditions for the second case study are concerned with the instantiates declaration (Figure 15). It should be proved that the given definitions satisfy the axioms given in HashableTheory<T> (Figure 11), when the type variable T is instantiated with Integer.

Hopefully, the given constructions in the specifications should allow to prove formally the expected behavior of the fib method. However, there remains a problem known as the issue of hidden side-effects [6]. This final missing element should be investigated further: the contract says that there is no side-effects at all, whereas in reality the private memo hash map can be modified.

6 Conclusion

To formally specify generic methods and classes, it is necessary to extend the existing specification languages. In the case of generic Java, we address this question for the KML specification language. In KML it is possible to model programs with logical specifications, called *theories*. We first generically specify two typical examples of generic Java code, namely a sorting method and a class for hash maps. From these examples, it appears mandatory not only to add type parametricity in theories, but also to provide a notion of parametricity of theories and a corresponding notion of theory instantiation. We proposed here a complete set of constructs achieving this goal.

This is on-going work and it clearly remains to formalize the proposed constructions, to express what are the necessary verification conditions in general, and to show a soundness result. We have already implemented the definition and importation of theories within the Why platform. Also, a relation has to be established between our approach and similar approaches based in refinement techniques [8].

Another future work is to apply a similar approach to the formal specification of C programs, i.e by extending the ACSL language [1]. An issue is that C's type system is even weaker than Java generics, for example the qsort() function in the C standard library is made generic via the use of void pointers, and function pointer to pass the comparison function as argument. In fact, void* plays the same role as the ? in Java, which means that the annotation language will need to annotate each occurrence of void* by a regular type variable.

Acknowledgments

This work is supported by the INRIA CeProMi "Action de Recherche Collaborative" (ARC). We would like to thank C. Paulin, A. Paskevych, W. Urribarrí, A. Tafat, R. Bardou and J. Kanig for helpful discussions and suggestions.

References

[1] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[3] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[4] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.

[5] G. T. Leavens and Y. Cheon. Design by Contract with JML. Available from `http://www.jmlspecs.org`, 2006.

[6] C. Marché. Towards modular algebraic specifications for pointer programs: a case study. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[7] C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009. `http://krakatoa.lri.fr/ws/`.

[8] A. Tafat, S. Boulmé, and C. Marché. A refinement methodology for object-oriented programs. `http://www.lri.fr/cepromi/index.php/Publications`, 2009.

[9] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.

Language Description for Frontend Implementation

Anya Helene Bagge

Bergen Language Design Laboratory
Dept. of Informatics, University of Bergen, Norway

December 2009

Abstract

For a language to be useful, it requires a robust and reliable implementation. Writing and maintaining such an implementation is a hard task, particularly for experimental or domain-specific language projects where resources are limited. This paper describes an implementation approach based on modular specifications of syntax and static semantics. Specification is done in a language description DSL, MetaMagnolia, from which compiler frontends can be automatically generated.

1 Introduction

The semantics of programming languages is commonly divided into static semantics, specifying the class of well-typed, valid programs; and dynamic semantics, specifying the behaviour or meaning of programs. A compiler has to consider both kinds, with the compiler frontend being responsible for rejecting programs that are invalid according to the static semantics, and delivering valid programs to the rest of the compiler in a suitable internal representation.

While language syntax has traditionally been formally specified, specifications of formal semantics are much rarer, with Standard ML [7] being the most famous example. Defining semantics is generally seen as much harder than syntax definition, and the payoff is less – while production-quality parsers can be generated from a grammar, compilers are still mostly built by hand, with compiler generation mostly being used as a prototyping tool.

Formal descriptions of semantics have several advantages, though, since it opens up the possibility of formal reasoning about the semantics, and gives a foundation for reasoning about programs in the language.

In this paper, we will introduce *MetaMagnolia*, a domain-specific language used for specifying the syntax and static semantics of the Magnolia Programming Language [2, 3]. Our goal is to provide for formal, yet pragmatic specification of compiler frontends, so that we may generate high-quality, usable compiler components,

rather than just prototypes. Being able to specify other languages than Magnolia is of secondary concern for now, we would rather have a useful formalism and tool for Magnolia, than a half-finished general tool. However, we hope that the same approach can be used for other languages, and possibly be extended to more general applicability.

Magnolia is a statically typed, modular language with support for overload – a feature that typically complicates specification of static semantics. The overload resolution code is clearly the most complicated part of the old Magnolia compiler, so being able to specify this nicely is important.

Magnolia is also designed to support experimentation and language extension [2], so being able to make custom language definitions with extra syntax and extended semantics is desirable. In particular, we would like to be able to tweak the existing semantics when creating language variants – for example, changing overloading preferences.

We will start by discussing syntax description in the next section, then move on to our static semantics formalism in Section 3, error handling in Section 4, before discussing related work, future directions and conclusion in Section 5. Some auxiliary definitions and rules are provided in the appendix.

2 Syntax and Sugar

The Magnolia language presented to the user is layered on top of a simplified core language (*Magnolia Core*), where unnecessary variation has been eliminated. This has the benefit of reducing the number of constructs that must be dealt with in the language specification and in the compiler. In order to support language experimentation and evolution, Magnolia is designed as an extensible language [2] – it is thus natural to design the extension facility in such a way that it can also be used to implement the default user-level language (*Magnolia Base*).

The compiler frontend translates between the layered language, and the core language. The optimisation and code generation phases of the compiler work solely on the core language. Magnolia Core differs from Magnolia Base in the following respects:

- All names are fully qualified and unique (i.e., no overloading).
- Variables in Core are always explicitly typed (e.g., `x:int` instead of just `x`).
- Language constructs have no optional parts (e.g., the `else` clause of an `if` must be specified, even when empty).
- Superfluous constructs are removed (e.g., operator calls are mapped to function calls)

Thus, translation to Core consists of

- Qualifying all names and resolving overloaded names.
- Qualifying all variables with type information.
- Removing optional clauses and other syntactic sugar.

```
language module Magnolia
syntax mix Magnolia/Core + Magnolia/BaseExt
expression syntax
 +⟦ n ⟧          <-> n : ?              // Variables
  ⟦ e1 o e2 ⟧    <-> _o_(e1, e2)     // Infix operators
  ⟦ o e1 ⟧       <-> o_(e1)             // Prefix operators
  ⟦ e[es*] ⟧     <-> index(e, es*)   // Indexing operator
statement syntax
 +⟦ if e then s1* end ⟧     <-> if e then s1* else end
```

Figure 1: Fragment of language definition module for Magnolia Base. The `syntax mix` declaration references external SDF syntax definitions. Syntax rules preceded by + introduce new syntax, the others use externally defined syntax from `Magnolia/BaseExt`.

The semantic analysis part of the frontend, discussed in Section 3, takes care of name and type qualification, while the desugaring part, discussed here, deals with removing syntactic sugar.

The syntax of Magnolia Core is specified in SDF2 [13], and is split into multiple independent files, so that new language variants can be built by combining fragments of the Core language. Syntax extensions can be done by adding SDF definitions, or directly as MetaMagnolia syntax rules.

2.1 Language Definitions

Language variants are specified in language definition files, as shown in Figure 1. External syntax SDF syntax definitions are pulled in and combined by the `syntax mix` declaration. The new language (hereafter referred to as the 'sugar language') is related to the Core syntax by syntax rules, which define a translation based on the *banana algebra* approach of Andersen and Brabrand [1].

The syntax rules form a *constructive catamorphism* (or *banana*) from the sugar language to the core language. Catamorphisms [6] are a generalisation of list folding from functional programming, working on any recursive data type – in this case, abstract syntax trees. For every term constructor in the input language, there is a rule that maps to a similarly-sorted term in the output language. Subterms are processed inductively, by implicit recursion. While this translation scheme is less powerful than one based on unrestricted program transformation or attribute grammars, it offers several benefits:

Guaranteed Completeness: It is possible to statically verify that every construct in the input language is mapped to the output language.

Safety: The translation will always terminate, and always maps between syntactically valid terms.

Starting with	Non-terminal sort	Starting with	Non-terminal sort
d	Declaration	*o*	Operator
e	Expression	*p*	Procedure
f	Function	*s*	Statement
m	Parameter mode	*t*	Type
n	Name / Identifier	*v*	Variable

Figure 2: Meta variables and their types. Variable names are constructed from an initial letter, optionally followed an *s*, a series of digits and a series of primes. Identifiers that do not match this scheme are literal identifiers.

Efficiency: The translation can be implemented very efficiently.

As we gain more experience, we may make desugaring rules slightly more powerful, if needed, as long as we can still ensure the safety and efficiency of the desugaring process.

2.2 Syntax Rules

The left hand sides contain patterns in the sugar language (enclosed in brackets), while right hand sides contain patterns in the core language, possibly with sugar language subterms to be processed recursively (in brackets). For example,

+⟦ **if** *e* **then** s_1***** **end** ⟧ <-> **if** ⟦ *e* ⟧ **then** ⟦ s_1***** ⟧ **else end**

Variables are typed according to a fixed naming scheme (see Figure 2), and are displayed here in italics. A star or plus indicates list matching. Brackets around variables on the right hand side are optional, since the variables are already known to in the input language. The type (non-terminal sort) of the right hand side is required to be the same as on the left hand side (e.g., mapping expressions in the input language to expressions in the output language).

Rules can be left-to-right (*desugaring rules*, as above), right-to-left (*ensugaring rules*) or both ways. A plus in front of a pattern indicates that the grammar should be extended according to that pattern – this is done by mapping variables to non-terminals, and non-variables to literal tokens. Grammar productions too complicated to fit into this simple scheme, such as infix operators with precedence, must be specified externally in SDF (for now).

Ensugaring rules are intended to be used to pretty-print code, or to generate specialised versions of semantic rules, tailored to an specific input language.

2.3 Parsing and Desugaring

Based on the language definition, the MetaMagnolia compiler will generate an SDF syntax definition, importing any syntax definitions referenced by the language definition, and adding productions for any syntax defined in rules.

The SDF syntax definition is then used to generate a parse table and a regular tree grammar for the parse tree, which can later be used to perform format checking on parse trees.

Syntax rules are straight-forwardly translated to Stratego [4] rules for desugaring and ensugaring, and a custom frontend module is built which parses input programs, applies desugaring and feeds the code to the next step in the compiler pipeline.

3 Specification of Static Semantics

Static semantics encompasses specification of type checking, and of other compile time checks to determine the validity of an input program. Classic specification of type checking, e.g., as presented by Pierce [11], consists of providing *type judgements*, that succeed or fail depending on whether a term is well-typed or not. For example, the following type judgement states that, given an environment Γ, if f has the type $T \to U$ and e has the type T then $f(e)$ is well-typed, with type U.

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash e : T}{\Gamma \vdash f(e) : U} \tag{1}$$

Such rules are unsuitable for direct use for semantic analysis in a compiler frontend for a language with overloading and named scopes; we also need information from name and overload resolution – we would need to know which particular f the programmer is referring to. Type checking and semantic analysis should therefore yield both a decision on program validity, and an annotated program tree. While name resolution could be done separately from type checking, overload resolution is tied in with typing, as the type of an expression will depend on the result of overload resolution, and overload resolution will depend on the types of arguments. It therefore makes sense to combine these aspects of static semantics into a single set of rules.

We have the following requirements for a static semantics formalism for Magnolia:

Pragmatic: Should support the generation of a real, high-quality compiler frontend with good performance and good error reporting – not just serve as a rapid prototyping tool.

Formal: Should have a formal basis, so properties of the language can be verified, and so semantic analysis can be statically check for completeness and correctness.

Modular: Should support the addition of new language constructs, and modification of existing constructs.

Friendly: The formalism should be concise and be familiar to people with previous exposure semantic formalism.

3.1 Static Semantic Rules

We use a formalism based on the familiar *structual operational semantics* (SOS) [12], but with some changes and extensions to make it more convenient for complex programming languages.

Static semantics is described by *resolution rules* of the form $pat \Rightarrow pat'$, where each rule may have zero or more *premises*, written above a dividing horizontal line.

Rule patterns are terms over Magnolia Core, specified in concrete syntax, using the same variable naming scheme as before (Figure 2). The typing of the resolution rules is shown in Figure 3.

A premise can be either a *resolution* `pat ==> pat'` ("*pat resolves to pat'*", invoking another semantic rule); a *transition* `pat --X-> pat'` ("*X transforms pat to pat'*" or "*pat X-es to pat'*"); an *operator call* `op(pat1,pat*) = pat'`; or a *predicate* `pred(pat*)`. A rule succeeds if it left hand side pattern matches and all its premises succeed.

Rules are evaluated in a *context*, containing information about name bindings (i.e., the environment in a traditional SOS setting) and other useful information. The context is an abstract data type, only accessed through operations such as name lookup, name binding, scoping, etc. The context is implicitly propagated to all premises of a rule (Appendix, PROP1–PROP3), and may be modified only upon applying a resolution rule.

For example, the non-overloaded function application rule from (1) would look like this in our scheme:

```
construct funapp:
    ?(f) = fun f'(_:t) : t'        e ==> e', t
    ---------------------------------------
              f(e) ==> f'(e'), t'
```

In the first premise, the lookup operation `?` is applied to the function name f, yielding a function declaration; the second resolves e to an annotated expression e_1 and a type t_1. The overall transition resolves $f(e)$ to an annotated expression $f'(e_1)$ and its type t'. The `construct` keyword introduces rules relevant to a particular construct.

With the context made explicit, the rule would look something like:

```
construct funapp:
    ?(f, ctx) = fun f'(_:t) : t'        e, ctx ==> e', t
    ------------------------------------------------
                  f(e), ctx ==> f'(e'), t'
```

In keeping with our pragmatic approach, we place the following restrictions on premises, enabling semantic rules to be easily translated to rewrite rules:

- The left hand side of a premise may not refer to unbound variables. Unbound variables occurring on the right hand side gets bound if the premise holds (already bound variables much match the result).
- All variables in predicate and operator arguments must be bound.
- Premises are processed in sequence.

Also, rules with overlapping patterns are tried in the order they appear in.

Rules are placed in a language definition module, with the rules for each construct preceded by a `construct` declaration. In the future, for the `construct` declarations to provide more information than just semantic rules – for example, we could also provide syntax specification, desugar rules for language extension, and even language

$$e \Rightarrow (e', t') : Ctx \times Expr \rightarrow Expr \times Type$$
$$s \Rightarrow s' : Ctx \times Stat \rightarrow Stat$$
$$d \Rightarrow d' : Ctx \times Decl \rightarrow Decl$$

Figure 3: Typing of transition rules for expressions, statements and declarations. The context (*Ctx*) is implicitly propagated and does not appear in the patterns.

user documentation. Also, by naming the constructs and analysis rules, we make it possible to refer to them later on, for example to replace the rules for a particular construct in an language extension, or provide additional rules, etc.

3.2 Resolving Expressions

For resolving expressions, we have the following considerations:

- Magnolia allows function overloading, so we need access to the types of subexpressions when resolving an expression.
- Any names used must be updated to the unique identifier of the relevant function, type or variable.

Expression resolution will therefore yield both an updated expression, and its type.

Let's start with a simple rule for variables:

```
construct variable:
    n --?-unique-> var n':t' = _;
    ----------------------------
        n:? ==> (n':t', t')
```

In Core syntax, a variable consists of a name and a type. For an unresolved variable, the type is unknown, hence `n:?` in the pattern above. Name resolution is performed by a transition, applying first the lookup operator `?`, which yields a list of declarations. In the case of variables, we don't allow overloading, so we apply `unique` which accepts exactly one declaration, failing otherwise. The result is matched against a variable declaration, picking out the resolved name and type.

A more complicated example is function application:

```
construct function_application:
                e* ==> (e'*,t'*)
    n --?-applicables(t'*)-best-unique-> (f, t'')
    ---------------------------------------------
          n(e*) ==> (f(e'*), t'')
```

Here we have a function name `n` and a list of arguments `e*`. We start by resolving the arguments, getting a list of resolved arguments and a list of their types. Then we do a

name lookup of `n`, giving us a list of declarations. We now need to find the declarations that match the particular argument types – this is done by the `applicables` filter, which yields list of weighted pairs of identifiers and return types. Finally, since we may be faced with more than one applicable candidate, we find the `best` candidate according to the weights, and ensure that we have a `unique` result.

Applicability is checked by a filter, applying `applicable?` to each element of the declaration list, and removing those for which it fails. A function f with formal argument types t_1* is applicable with respect to an actual argument list t_2* if t_1* is compatible with t_2*:

operator applicable?:

$$\texttt{compatible}(t_1\texttt{*},\ t_2\texttt{*}) = x$$

---

function $f(n_1\texttt{*}:t_1\texttt{*}):t = e$; --applicable?$(t_2\texttt{*})$-> $(f,\ t)$ weight x

The compatibility check yields a weight, x, that indicates how 'good' the match is (number of type conversions, generic type matching, etc.). Note the argument list matching convenience syntax – $n* : t*$ matches a list of name/type pairs $n_1 : t_1 \cdots n_k : t_k$.

Now, type compatibility can get interesting, if we go beyond just checking for equality. For instance, assume we allow implicit conversions from one type to another, by defining a function with the same name as the target type. In this case, we would like the implicit conversion made explicit, so we should let the compatibility check yield a modified argument list. The applicability and function application rules would have to be changed accordingly. The rule for checking a single argument would look like (with 0 and -10 being the weights given for no conversion and implicit conversion, respectively):

$$\texttt{compatible?}(t,\ t,\ e) = e \texttt{ weight } 0$$

$$\texttt{not}(\texttt{equal}(t_1, t_2))\quad \texttt{nameOf}(t_1) = n\quad n(e_2) \texttt{ ==> } (e_2',\ t_2)$$

$$\texttt{compatible?}(t_1,\ t_2,\ e_2) = e_2' \texttt{ weight } -10$$

Similarly, to support generics, we would have `compatible` yield the list of type argument bindings that was required for argument list compatibility.

3.3 Resolving Statements

Statement resolution consists mostly of processing the constituent parts, and combining the result. For example,

construct if:

$$e \texttt{ ==> } (e',\ t')\quad \texttt{truthType}(t')\quad s_1\texttt{* ==> } d_1'\texttt{*}\quad s_2\texttt{* ==> } s_2'\texttt{*}$$

---

if e **then** s_1* **else** s_2* **end** ==> **if** e' **then** d_1'* **else** s_2'* **end**

$$\begin{aligned}
\textbf{declare}(d*) &: Decl*, Ctx \rightarrow Ctx \\
\textbf{loadModule} &: Name \rightarrow Decl \\
\textbf{uniqueId}(t*) &: Type*, Ctx \rightarrow Name \\
\textbf{uniqueId} &: Ctx \rightarrow Name \\
\textbf{set}(n, x) &: Name, \tau, Ctx \rightarrow Ctx
\end{aligned}$$

Figure 4: Auxiliary operators and context operations.

The `if` statement is resolved by resolving the condition, checking that it has a truth value type, then resolving the branches.

Procedure calls follow the same pattern as function application; resolve the arguments, lookup the procedure name and find the best candidate that matches the argument list.

construct procedure_call:

$$\frac{\begin{array}{c} e* \texttt{ ==> (}e'*\texttt{, }t'*\texttt{)} \\ n \texttt{ --?-callables(}t'*\texttt{)-best-unique-> } p \end{array}}{\textbf{call } n(e*); \texttt{ ==> } \textbf{call } p(e'*);}$$

The `let` construct is more interesting, as it involves modifying the context:

construct let:

$$\frac{e \texttt{ ==> (}e'\texttt{, }t'\texttt{)} \qquad s* \texttt{ ==}\textbf{declare}(\textbf{var } n \texttt{ : } t' \texttt{= } e'\texttt{;}) \texttt{=> } s'*}{\textbf{let var } n \texttt{ : ? = } e\texttt{; } \textbf{in } s* \textbf{ end} \texttt{ ==> } \textbf{let var } n \texttt{ : } t' \texttt{= } e'\texttt{; } \textbf{in } s'* \textbf{ end}}$$

The `declare` operation adds a declaration to the context used to resolve the statement list `s*` (see Figure 4 for this and other auxiliary operations).

3.4 Resolving Declarations and Modules

Declarations should be mapped to resolved declarations (i.e., with constituent parts resolved, and the declaration name made unique). For example,

construct function_declaration:

$$\frac{\begin{array}{c} t \texttt{ ==> } t' \qquad t_1* \texttt{ ==> } t_1'* \qquad n \texttt{ --uniqueId(}t_1'*\texttt{)-> } f \\ e \texttt{ ==declareArgs(}n_1*\texttt{,}t_1'*\texttt{)=> } e' \end{array}}{\texttt{fun } n(n_1*\texttt{:}t_1*\texttt{) : } t \texttt{ = } e\texttt{; ==> fun } f(n_1*\texttt{:}t_1'*\texttt{) : } t' \texttt{ = } e'\texttt{;}}$$

where `uniqueId` constructs a unique function identifier based on the name and the parameter list, and `declareArgs` adds declarations for all the parameters to the context

used to resolve the function body:

```
operator declareArgs
    ctx --declareArgs([],[])-> ctx'

       ctx --declare(var n : t = _;)-> ctx'
    ----------------------------------------
    ctx --declareArgs([n|n'*],[t|t'*])-> ctx'
```

This resolution style disallows recursive functions – to allow recursion, the function declaration would also have to be added to the context of the body.

This specification style also means that all operations must be declared before use – a more user-friendly style could be achieved by splitting the declaration rules into multiple passes, one dealing solely with declarations, and the next dealing with declaration bodies.

A *module* is a named list of declarations. We add the name of the current module to the resolution context, for use in creating unique names for declarations.

```
construct module:
  n --uniqueId(n')-> n' d* ==set(currentModule,n')=> d'*
  ------------------------------------------------------
              module n d* ==> module n'd'*
```

Declaration lists are resolved such that a declaration is visible in the context of the following declarations. Importing of modules is dealt with by adding all the declarations of the imported module to the context.

```
construct declList:
  [] ==> []

  n --loadModule-> d    d ==> module n'd'*   ds ==declare(d'*)=> ds'
  ------------------------------------------------------------------
                  [import n|ds] ==> [import n'|ds']

  d ==> d'   ds ==declare(d')=> ds'
  ---------------------------------
        [d|ds] ==> [d'|ds']
```

Processing of a program starts with loading a main module, and then resolving that.

4 Handling Errors

Failing with just a "typechecking failed" message is clearly not a satisfactory solution for a compiler frontend. We would like error messages to inform the user of *what*

went wrong, *where* it when wrong (location and code sample), and provide related information like a list candidates for function overloading.

One possibility for dealing with errors is to introduce fallback rules, that are tried when the main resolution rules fail. For example, we could have a rule that reported failure in variable lookup (with `/fail` in the construct name indicates a fallback rule):

```
construct variable/fail:
                n --?-> []
    ---------------------------------------
    n:? ==> fail("Undeclared variable <n>")
```

Of course, plenty of other things could go wrong when resolving a variable – we might find that the name refers to something which is not a variable, for example. Making one rule for each case is tedious, and gives us a specification where most of the rules are actually concerned with error handling. On the other hand, a single error message for all variable errors will provide the user with poor feedback:

```
construct variable/fail:
    n:? ==> fail("Error resolving variable <n>")
```

Another way to deal with error reporting is to trigger error messages as soon as possible. This can be done by extending the lookup transition chain with more steps, that check for errors and report as necessary:

```
construct variable:
    n --?-nonEmpty-vars-unique-> var n':t' = _;
    ---------------------------------------------
                n:? ==> (n':t', t')
```

Where `nonEmpty` is defined as

```
      []  --nonEmpty-> fail("Unknown name")
[d | ds]  --nonEmpty-> [d | ds]
```

`vars` is filter that picks variable declarations, and `unique` is:

```
      []   --unique-> fail("Not a variable")
      [d]  --unique-> d
      ds   --unique-> fail("Ambiguous variable name, candidates are: <ds>")
```

This allows us to differentiate between "nothing found by that name" and "no variable found by that name".

Now, `unique` is likely to be reused in for more than just variable resolution, so having the error messages refer directly to variables may be a bad idea. What we can do, is to make relevant information available through the context. Each `construct` declaration provides us with a name for the construct being processed, which is clearly useful in forming error messages. We will therefore extend the definition of resolution rules so that information about the current construct and the current program term being resolved (including location information) is added to the context (Appendix, Constr-

DEF). This information can be accessed when building error messages:

```
[]   --unique-> fail("<curTerm> is not a <curConstruct>")
[d]  --unique-> d
ds   --unique-> fail("Ambiguous name in <curConstruct>,
                      candidates are: <ds>")
```

A further refinement is to let the lookup operator ? save its result in the current context, so that this information is available further down the transition chain.

4.1 How to Treat Failure

While we now have a fair idea of how to construct meaningful error messages, there is still the problem of what to do when an error is encountered. Just writing out the message and aborting the whole resolution process is a possibility, but this is sub-optimal for programmers, who will have to discover and fix type errors one at a time. It is bettter to find all errors in the program first, before aborting the compilation.

The scheme used in our old compiler is to output the error message and simply give a dummy 'unknown' result for the resolution. E.g., for any expression:

construct expression/fail:

```
e ==> (e, ?)
```

We can avoid cascading errors by simply withholding error messages when any sub-resolution gives an 'unknown' result.

5 Discussion

The restricted format of the desugaring rules means that it should be possible to comb-ing desugaring and resolving into a single pass. This would also be advantageous from the user's perspective as error messages will be related directly to the input code rather than code in the Core language. The idea here is to use the desugaring rules in the reverse direction on the static semantic rules to obtain rules tailored to a particular input language.

Dinesh [5] deals with error handling by having type checking of a term yield *true* if the type-correct and yielding a structure error message otherwise. These messages propagate up through the program, so that the overall type checking results in either *true* or a message. This assumes of course that we're just dealing with type checking and not overload resolution, but the idea of propagating messages up the program tree instead of outputting directly is useful, particularly since we may want to create rules like "if this resolves, do something, otherwise do something else", without getting spurious error messages from checking whether something resolves or not.

Mosses and New [10] introduce I-MSOS, an approach to MSOS with implicit propagation of auxiliary entities. Our context-propagation approach can be seen as a limited version of this. It may turn out that our scheme is too restrictive, and we may have to introduce more entities than just the context, and allow more powerful propagation patterns. In particular, it may be useful to allow semantic rules to insert new declarations

into the module top-level list of declarations, something which is not possible in our approach.

The main inspiration for our static semantic rules is SOS, *structural operational semantics* [12, 9], and the modular variant, MSOS [8]. While traditional SOS explicitly deals with environments, stores and other auxiliary entitites, MSOS attaches labels to the transitions, indicating how the auxiliary entities are propagated throughout a rule. This means that rules can be reused for different languages which have different sets of auxiliary entities. We have less need for many different kinds of auxiliary entities, preferring to model this using a single *context*. Instead we need to do a wide range of processing, such as name lookup, filtering and overload resolution, which is conveniently specified on transition arrows. Thus labels on arrows in MSOS have an entirely different meaning than operators on arrows in MetaMagnolia.

5.1 Conclusion

MetaMagnolia is a formalism for specifying the syntax and static semantics of programming languages (in particular, the Magnolia language). Syntax is described either in the SDF formalism, or directly as MetaMagnolia rules; desugaring rules are then written to map the syntax into a core language. Static semantic rules for the core language are specified in an SOS-like formalism, which allows complex language features like overloading to be specified using series of transitions.

The syntax and semantic rules are compiled into SDF [13] and Stratego [4] code, giving a custom compiler frontend for the particular language definition. This is intended to replace the old hand-written Magnolia frontend as soon as the tools are mature enough.

Acknowledgements This research is partially funded by the Research Council of Norway. Magnolia is developed at the Bergen Language Design Laboratory.

References

[1] Jacob Andersen and Claus Brabrand. Syntactic language extension via an algebra of languages and transformations. In Torbjörn Ekman and Jurgen J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.

[2] Anya Helene Bagge. Yet another language extension scheme. In *SLE '09: Proceedings of the Second International Conference on Software Language Engineering*. Springer, 2009. To appear.

[3] Anya Helene Bagge and Magne Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In Torbjörn Ekman and Jurgen J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.

[4] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[5] T.B. Dinesh. Type-checking revisited: Modular error-handling. In *In Proceedings of the Workshop on Semantics of Specification Languages*, pages 216–231. Springer-Verlag, LNCS., 1993.

[6] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, revised edition, 1997.

[8] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[9] Peter D. Mosses. Programming language description languages: From Christopher Strachey to semantics online. In *Formal Methods: State of the Art and New Directions*, pages 247–271. Springer, 2009. To appear.

[10] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(4):49–66, 2009.

[11] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[12] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

A Auxiliary Rules for MetaMagnolia

Implicit propagation is desugared according to these rules, with the context being distributed over the rule definition:

$$\frac{h_1 \cdots h_n}{\pi \overset{\omega}{\Rightarrow} \pi'} \quad \rightarrow \quad \frac{h_{1,\rho} \cdots h_{n,\rho}}{\pi \overset{\omega}{\Rightarrow}_\rho \pi'} \qquad \text{(PROP1)}$$

$$\frac{h_1 \cdots h_n}{\pi \overset{\chi}{\rightarrow} \pi'} \quad \rightarrow \quad \frac{h_{1,\rho} \cdots h_{n,\rho}}{\pi \overset{\chi}{\rightarrow}_\rho \pi'} \qquad \text{(PROP2)}$$

$$\pi \overset{\chi}{\rightarrow}_\rho \pi' \quad \rightarrow \quad \pi, \rho \overset{\chi}{\rightarrow} \pi', \rho' \qquad \text{(PROP3)}$$

Resolution is just a particular kind of transition:

$$\frac{\pi, \omega(\rho) \xrightarrow{\text{resolve}} \pi', \rho'}{\pi \overset{\omega}{\Rightarrow}_{\rho} \pi'} \quad \text{(Resolve)}$$

Operator calls and *predicates* are simply sugar for transformations, with predicates discarding the result:

$$\frac{\pi_1 \xrightarrow{op(\pi*)} \pi'}{op(\pi_1, \pi*) = \pi'} \quad \text{(Op-Call)}$$

$$\frac{\pi_1 \xrightarrow{pred(\pi*)} \_}{pred(\pi_1, \pi*)} \quad \text{(Pred-Call)}$$

Transition chains:

$$\frac{\pi, \rho \xrightarrow{\chi_1} \pi', \rho' \qquad \pi', \rho' \xrightarrow{\chi*} \pi'', \rho''}{\pi, \rho \xrightarrow{\chi_1 \chi*} \pi'', \rho''} \quad \text{(Trans-Chain)}$$

Construct resolution definition:

$$\texttt{construct C} \ \left(\frac{h_1 \cdots h_n}{\pi \overset{\omega}{\Rightarrow} \pi'}\right) \ \rightarrow \ \frac{\rho \xrightarrow{\text{set}(curConstruct,C),\text{set}(curTerm,\pi)} \rho' \qquad h_{1,\rho'} \cdots h_{n,\rho'}}{\pi \overset{\omega}{\Rightarrow}_{\rho} \pi'} \quad \text{(Constr-Def)}$$

DSL Tools: Less Maintenance?

Paul Klint[1,2], Tijs van der Storm[1,2], and Jurgen Vinju[1,2]

Software Analysis and Transformation Team
Centrum Wiskunde & Informatica[1]
Universiteit van Amsterdam[2]
Amsterdam, The Netherlands

Abstract

The current rise of DSLs is often attributed to the existence of modern DSL tools that help to mitigate cost of ownership. Our main research goal is to investigate that non-trivial hypothesis.

We present empirical results on the maintainability of six implementations of the same Domain Specific Language (DSL) using different languages (Java, JavaScript, C#) and DSL tools (ANTLR, OMeta, Microsoft "M"). Our evaluation largely confirms that the maintainability of DSL implementations is higher when constructed using DSL tools.

1 Introduction

Domain Specific Languages (DSLs) promise an increase in productivity, maintainability and reliability by providing notations tailored to certain problem domains [20, 28]. Solutions can be described at a higher level of abstraction, thus narrowing the gap between problem domain an solution domain.

There are many approaches for the development of DSLs. *Internal* (embedded) DSLs reuse the syntax and parser of a general purpose host language (e.g. Ruby, Scala). *External* DSLs, on the other hand, are often developed using DSL tools. Such tools include parser generators (e.g. ANTLR, Yacc), transformation systems (e.g., ASF+SDF, Stratego, TXL) or attribute grammar systems (e.g., JastAdd, Kiama, LISA). The goal of these tools is to lower the cost of DSL ownership by facilitating the construction of DSL implementations.

Just like software has to evolve to remain viable [10], DSLs are subject to maintenance activities as well [28, 29]. Especially DSL implementations are complex and unusual when compared to mainstream software applications. Therefore, maintainability (understandability) of DSL implementations seems to be a more critical factor than usual.

On the one hand, an explanation for the "fall of DSLs" in the nineties could be that the cost of maintenance of 4GL languages and other DSLs outweighed their benefits. On the other hand the current rise of DSLs is often attributed to the existence of more modern DSL tools that help to mitigate costs of DSL ownership [8]. Our main research goal is to investigate that non-trivial hypothesis.

For instance, a context-free grammar as input to a DSL tool is a complex and dense form of source code that needs to be subjected to disciplined engineering practices [15]. Grammar changes also naturally have a high impact on other components (e.g., checkers, evaluators, compilers) of a DSL implementation. Almost paradoxically, parser generators really do not (spontaneously) generate parsers. Rather they automate a part of parser construction using grammars—taking parser implementation to a higher level of abstraction. The principle intellectual effort, however, appears to remain with the developers and maintainers. Is then the benefit of parser generation as opposed to manual parser construction really significant compared to the total cost of parser ownership?

We present an initial empirical study on six implementations of the same DSL. Three implementations are developed from scratch, in Java, JavaScript and C# respectively. We call these implementations the "vanilla" implementations. The other three implementations are supported by DSL tools: ANTLR (Java) [22], OMeta (JavaScript) [32] and "M" of the Microsoft SQL Server Modeling Platform (C#) [21].

In this paper we focus on the difference between vanilla implementations and implementations using DSL tools. Our research question is:

> **Is the use of DSL tools beneficial for the maintainability of the resulting language implementation as compared to not using them?**

This paper represents the following contributions:

1. We published more than six implementations of the same non-trivial DSL as open source software[1]. The implementations all include a parser, checker and evaluator.
2. We present quantitative data (metrics) and qualitative analyses on six of these implementations in this paper.
3. To our knowledge, this is the first work investigating maintainability aspects of the use of DSL tools.
4. Our results largely confirm the hypothesis that the use of DSL tools increases maintainability.

We explicitly do not derive any recommendations from this study as to whether one should use a certain tool or not, since there are many aspects to the quality of DSL implementations; maintainability is just one of them. We consider performance, flexibility (e.g., error handling), and usability (e.g., debugging, testing etc.) as directions for future work.

2 A Little Language for Markup Generation

The DSL used in this study is WAEBRIC, which is a little language for XHTML markup generation. WAEBRIC is used for creating and maintaining (static) websites, by providing the user with abstraction facilities derived from functional programming. Sites can be built using reusable, functional building blocks. Figure 1 shows an example of how to generate a simple homepage.

This program consists of two modules. The first, `Homepage` imports the second, `Utils`. The WAEBRIC function `home` defines a very simple homepage. It invokes the

[1] `http://waebric.googlecode.com`

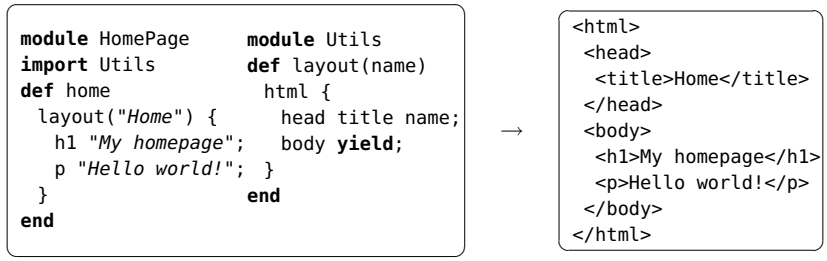


Figure 1: WAEBRIC program for a simple home page, including its output.

utility function `layout` with a single argument, the string literal "Home". Additionally a block (lines 4–7) is passed into the `layout` function. This block consists of two statements. The first produces an `h1` element and the second produces a `p` element. All valid XHTML tag names are built in "functions" that generate the corresponding elements.

The `layout` function, defined in the `Utils` module, produces a reusable skeleton for web pages, consisting of the root `html` element, containing a `head` and a `body` element. The block passed to the `layout` function is spliced into the `body` element using the built in **`yield`** statement (line 5). Note that nested within the `head` is a `title` element containing the `name` argument of `layout` (curly braces are optional if nesting occurs along a single spine).

WAEBRIC is a non-trivial language. In order to better appreciate the complexity of the implementations we present in Section 4, we highlight some distinguishing features here. As can be seen from the example in Figure 1, WAEBRIC is a modular language featuring function calls. Figure 2 shows how conditionals, iteration, and recursion are used. A limited form of *closures* is catered for in order to pass additional code blocks into functions (cf. the body of a page passed into the `layout` function, which is then "invoked" using **`yield`**). In addition to these features, WAEBRIC has support for **`let`** bindings for introducing local variables and functions.

Support for data is limited in WAEBRIC: strings, lists and records can be created and inspected, but not modified; there are no data operations except for string concatenation. For an example of how data is used, Figure 2 shows how to create a recursive menu from recursively structured data. The structure of the menu is provided to the function `menu` in the form of a record with two fields, title, and kids). An excerpt of the data structure is shown in function `the-menu` (line 1–3). The function `menu-item` processes a single element of the list of children by checking if a `kids` field is present and if so, recursing to `menu` for each child. Otherwise a `<li>` element is created with an anchor based on the `link` field.

Finally, a feature that can be considered a complicating factor for parsing WAEBRIC is string interpolation. Any string literal may contain an arbitrary number of embedded markup expressions. For instance, to annotate a snippet of text with a CSS class, one could use the following WAEBRIC statement:

```
echo "Embedding: <span(class="alert") "This"> is highlighted.";
```

In this example the markup statement `span(class="alert") "This"` is evaluated to the XHTML fragment `<span class="alert">This</span>` and then included in the result. The result of the **`echo`** statement thus will be:

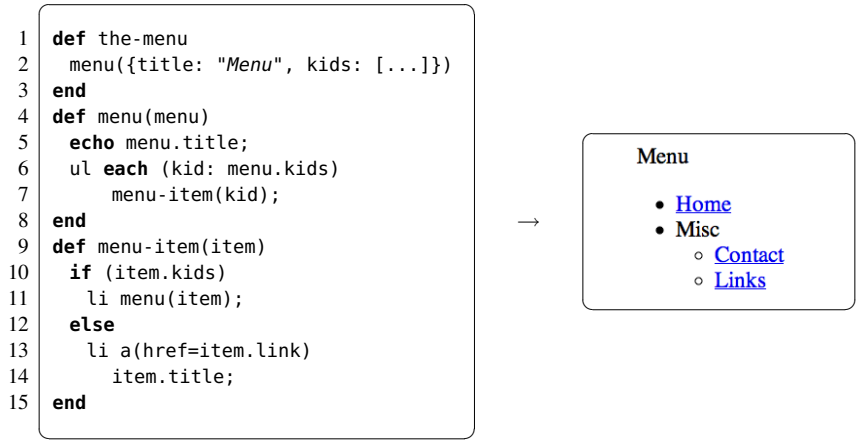


Figure 2: Recursive menus in WAEBRIC with output.

```
Embedding:  <span class="alert">This</span> is highlighted.
```

This feature is commonly used for embedding data (e.g. variables, parameters) into strings, or to embed (non-block-level) XHTML tags in mixed content.

The complication in parsing such interpolated strings is that, naturally, the embedded expressions are defined as ordinary, context-free non-terminals, whereas string literals are often dealt with directly in a scanning phase. To allow arbitrary nesting of embeddings, scanner and parser should synchronize.

WAEBRIC implementations are accompanied by a light-weight semantic analyzer which checks for common mistakes. Among the constraints that have to be checked, are the following: user defined functions are called with the right number of arguments, function definitions are unique (no duplicates), and referenced variables are either introduced as formal parameters of the enclosing function, or introduced by a `let` construct.

3 Evaluating Maintainability

We briefly introduce necessary concepts and methods for the evaluation of software maintainability here and explain our evaluation method for our six implementations of WAEBRIC. The results can be found in Section 4.

3.1 Introduction to Software Maintainability

Software maintenance is defined as the process encompassing all modifications made to a software system after its initial release. It is generally accepted that software maintenance is the most expensive phase in the software development life cycle. This is aptly summarized in the following fact from Robert Glass's *Facts and Fallacies of Software Engineering* [10]:

> FACT 41. Maintenance typically consumes 40 to 80 percent of software costs. It is probably the most important life cycle phase of software.

This means that a large part of a software's cost is spent on error correction, enhancement, or reacting to changes in the environment. Better maintainability of a software product can therefore be a great cost saver.

DSLs (and DSL tools in particular) are targeted at achieving increased productivity and enabling domain specific analysis, verification and optimization in a certain domain [20]. Nevertheless, this perspective emphasizes the *construction* of a software system, but not necessarily the maintenance afterwards. So, what is the primary difference between construction and maintenance? The difference is that in order to modify software, one has to understand it first. Moreover, as Glass states:

> FACT 44. Understanding the existing product is the most difficult task of maintenance.

It is not so much the modification of software itself that necessarily is expensive. What is expensive is to find out what has to be modified. Thus, in the following we will approach maintainability from an understandability point of view. The metrics that are introduced in the next section measure aspects of maintainability because they try to quantify how much work it might be to understand certain pieces of source code.

3.2 Maintainability Metrics

There are many metrics (see e.g. [7]) that aim to provide indicators for maintainability. These metrics focus on the complexity of understanding source code in terms of size and structure. A common indicator of size is the (non-comment) Lines of Code metric (NCLOC), which measures the lines of code in a system that do not consist of comments or blank lines.

Structural complexity is harder to quantify, but one well-defined metric is the McCabe Cyclomatic Complexity (CC) [19]. This metric is defined as the number of independent paths through a control flow graph of a program unit. The motivation is that to fully understand a function or method, one must understand all of its possible execution paths. CC can easily be computed by taking the number of decision points (e.g. while, if, etc.) in a function or method and add 1 to the result.

In our research we paid heed to the observations of [18] where the authors indicate a number of problems with prevailing maintainability metrics, such as Halstead Effort and the Maintainability Index (MI) . These metrics aim to quantify the maintainability of a software system by providing a single aggregated number over an entire software system. The authors, however, observe that these metrics are not suitable to do root-cause analysis: the formulae for computing both Halstead effort and the Maintainability index aggregate over program units using both averages and logarithms. As a consequence, the obtained metrics data is hard to trace back to source code properties.

As an alternative, [18] proposes to rank a software system for maintainability assessment according to the following rules:

- The size of a software system, measured in number of modules (e.g., files, classes etc.), units (e.g. methods) and non-comment lines of code (NCLOC), should be as low as possible.
- The percentage of NCLOC in units that have a cyclomatic complexity higher than a certain value X should be as low as possible.
- The percentage of duplicated code should be zero for a given minimal duplicated-fragment size.

	Unit	Decision point
General purpose	method, function	if, for, while, case, catch
Grammar	non-terminal	$\|$, $*$, $+$, ?

Table 1: Units and decision points in general purpose code and grammar code.

We have used these rules to quickly get an overview of maintainability of the six implementations. We have not formally measured the percentage of duplicated code. However, we will occasionally refer to instances of duplication that have been observed in the code. The cyclomatic complexity threshold *X* is discussed in more detail below, in Subsection 3.4.

3.3 Grammar Metrics

The metrics discussed above are usually defined for ordinary code, and not for the grammar formalisms used by ANTLR, OMeta and "M". However, Power and Malloy [25] have proposed a grammar-based interpretation of these metrics seeing each non-terminal as procedure. A non-terminal thus corresponds to a single code unit.

Correspondingly, the cyclomatic complexity of a grammar is derived from the number of decision points in it; in fact, it is defined to be equal to that number. In grammars, such decision points are represented by alternatives ($|$), optionals (?), and the closure operators for iteration ($*$, $+$). A summary of the correspondence is shown in Table 1.

Although [25] only defines the CC for a complete grammar, we have counted the CC per non-terminal. This allows us to reason about the ratio between simple and complex parts of a grammar, parallel to the way we evaluate complexity of the vanilla implementations.

If the production(s) for a non-terminal have *action code* attached, the CC of the code is additionally counted as CC of the non-terminal and the NCLOC of the code is added to the NCLOC of the module. A full understanding of a DSL implementation naturally includes understanding the control flow of action code. Also note that action code can conditionally influence choice of alternates or trigger backtracking behavior, directly influencing the control flow of the generated parser.

3.4 How we measured

We constructed generic (parse-tree-based) metric analyses using Rascal, a DSL for source code analysis and transformation [16]. All sources are parsed using simple partial (fuzzy) SDF grammars for the languages involved. The resulting parse trees are subsequently processed in Rascal. To compute NCLOC the white space nodes are inspected. CC is obtained through the use of production annotations. Annotations in the grammar mark conditional control flow. This allowed us to reuse the exact same analyses across different languages. Thus we eliminate the possibility that different tools (e.g., for Java and C#) count in different ways.

There is the issue of *preamble code* that will end up in the generated parser. It should contribute to the NCLOC metrics for grammar files. If the preamble contains methods these are counted as units as if they were put in ordinary source files and their cyclomatic complexity is measured accordingly.

	ANTLR	OMeta	"M"
Scan	Lexical grammar	Integrated	Lexical grammar
Parse	Context-free grammar		Context-free grammar
Technique	*LL(*)*	*PEG/Packrat*	*GLR*
AST	Generated	Reused	Generated
Check	Tree grammar	Grammar extension	Dispatching
Eval	Tree grammar	Reused	Dispatching

Table 2: Overview of implementation strategies in the DSL tool implementations

To compute the percentage of NCLOC that contributes to units with high complexity, the threshold is set at 6. This means that the percentages, described in detail below, apply to units with a CC greater than 6. A relatively low threshold acknowledges the intrinsic complexity of DSL implementations; they should be kept as simple as possible.

4 Quantitative Observations

Here we present facts acquired about the six WAEBRIC implementations.

4.1 Overview

We describe the differences between the various WAEBRIC implementations. Differences may be caused by the use of different general purpose languages, different developers that implemented WAEBRIC and different DSL tools applied. The implementations described in this paper have originally been described in three master's theses supervised by the second author [12, 30, 31].

In order to gain confidence in the fact that all six implementations actually implement the same language a test suite of 104 WAEBRIC programs was used to compare the resulting XHTML to the output of the reference implementation (built using ASF+SDF). All six implementations achieve 100% conformance on this test suite.

The vanilla implementations are quite similar. All implementations contain a handwritten lexer and parser. The parsers employ a recursive descent [1] strategy with variable look-ahead. The result of the parsing process is an abstract syntax tree (AST) consisting of nodes that are instances of classes in a statically typed hierarchy[2].

A common technique for tree traversal is the Visitor design pattern [9]. In the Java and C# implementation this pattern is applied for implementing the checker and interpreter. The Java script implementation uses explicit dispatch using `instanceof` execute actions for specific nodes.

In order to generate XHTML code from WAEBRIC programs two implementations depend on third-party libraries. First, the Java implementation uses the JDOM [14] XML library. Second, the JavaScript implementation uses John Resig's `env.js` [26] to simulate a browser environment, thus allowing internal DOM manipulation to be used for generating XHTML, even without support from an actual browser. C# uses a single custom XHTML class to generate XHTML from strings.

To provide some insight into how the DSL tool implementations are implemented, Table 2 briefly describes the implementation of each language component.

[2]JavaScript is a prototype-based language so there are no classes. "Statically typed" thus means that every node is an instance of a named object.

	Java			ANTLR			JavaScript			OMeta			C#			"M"		
Component	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N
Scan	6	71	491	–	–	–	13	68	530	–	–	–	10	61	**725**	–	–	–
Parse_c	13	79	958	–	–	18	16	166	**1724**	3	6	51	12	91	1366	2	4	70
Parse_g	–	–	–	1	52	133	–	–	–	1	66	144	–	–	–	1	80	**312**
AST	39	360	2045	–	–	–	22	130	612	22	130	612	81	464	**3647**	–	–	–
Check_c	7	17	183	–	–	–	6	93	**597**	7	27	261	9	22	334	9	26	430
Check_g	–	–	–	2	73	**294**	–	–	–	1	10	72	–	–	–	–	–	–
Eval_c	2	65	596	–	–	–	6	153	1074	–	–	–	7	86	1264	7	91	**1574**
Eval_g	–	–	–	1	29	**377**	–	–	–	–	–	–	–	–	–	–	–	–
Misc	6	28	**387**	2	3	186	3	2	82	7	19	222	2	2	84	2	2	74
Total	73	620	4660	6	157	1008	66	612	4619	41	258	1362	121	726	7420	21	203	2460

Table 3: Size metrics: number of files (#F), number of units (#U), and NCLOC (#N). The maximum in each row is in bold face.

The terms "lexical grammar" and "context-free grammar" mean that the DSL tool provides separate (grammatical) notations for specifying scanner and parser. Since OMeta employs a scannerless formalism these two components are integrated. The row beneath the *Parse* row describes the parsing algorithm that is used by the tools.

In the context of AST implementation, "Generated" means that the tool provides automatic AST generation; in all implementations these ASTs are dynamically typed (even if the tool could facilitate otherwise). "Reused"—only in the case of OMeta—entails that the implementation reuses the component in question from its vanilla counterpart.

For tree traversal (used in check and eval components) ANTLR provides a formalism of "Tree grammars". The OMeta checker uses grammar extension (inheritance) to implement visiting and matching behaviour; the evaluator is reused from the vanilla JavaScript implementation. Finally, "Dispatching" means that dynamic switch statements are used to traverse ASTs.

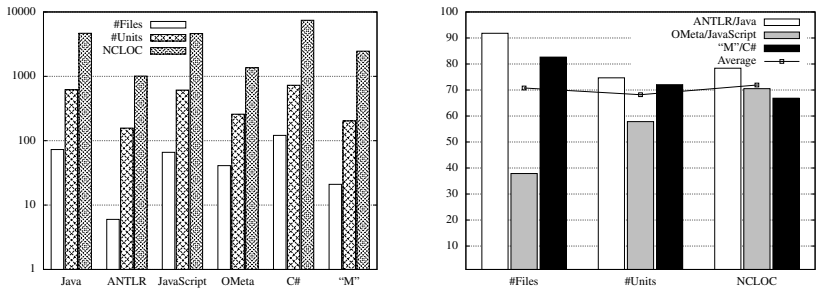
The table highlights some notable differences in the design between the non-vanilla implementations, some of which are caused by the tool that has been used (e.g., ANTLR providing tree grammars), and some of which can be ascribed to developer idiosyncrasies (e.g., implementing the Visitor pattern in an atypical way, or reusing the vanilla components).

4.2 Size

The results of our size measurements are shown in Table 3. The first column lists the various implementation components. To separate grammar code from ordinary code, three components (parse, check and eval) have an additional row containing the metrics computed from grammar code: C_c indicates the ordinary code in the component, whereas C_g indicates grammar code ($C \in \{\text{parse}, \text{check}, \text{eval}\}$).

Metrics for the scanning component are included in the grammar parsing row. Columns 2 till 7 represent each WAEBRIC implementation. Each implementation column is divided in three sub-columns, corresponding to number of files (#F), number of units (#U) and NCLOC (#N). An "–" indicates that the number for a certain component and implementation is zero.

Figure 3(a) graphically depicts the total size of each implementation in number of files, units, and NCLOC per implementation. Note that this has a logarithmic scale. Figure 3(b) shows the percentage of code reduction that has been obtained by using a DSL tool, in % per measured metric: $\%_{reduction} = 100 - (\#_{tool}/\#_{vanilla}) \times 100$, where



(a) Size totals

(b) Code reduction induced by DSL tool usage

Figure 3: Overview of size measurements per WAEBRIC implementation.

$\in$ {#Files, #Units, NCLOC}. The line indicates the average code reduction per metric.

Figure 3 immediately shows that the use of the DSL tools consistently reduces the number of files, units and NCLOC in each pair of WAEBRIC implementations. The average reduction in code size is around 70%. The percentage chart shows, for instance, that the reduction in NCLOC by using ANTLR is roughly 80%, which means that this implementation requires a mere one fifth of the NCLOC that are required in the Java vanilla implementation.

Parsing is the part that is affected most by the use of the tools, since all three tools are primarily parser generators. This can be easily understood since parser generators take away much of the boiler plate code required to write a parser, for instance iteration, backtracking and look-ahead. This is especially visible in the number of units metric (#U): one could say that one function per non-terminal does not seem to be true. More code units seem to be required than non-terminals, especially if we add the units for scanning and parsing. For instance, in the vanilla Java implementation, the number of units for scanning and parsing add up to 150, whereas these two phases implemented using ANTLR require only 52 non-terminals. For all implementation pairs the ratio between the number of units amounts to a factor of roughly 2 (C#/"M" = 152/80) or 3 (Java/ANTLR = 150/52, JavaScript/OMeta = 234/66).

AST code takes up a large portion of the vanilla code. Both ANTLR and "M" generate (dynamically typed) ASTs so this kind of code is completely absent from those DSL tool implementations; it is responsible for a large part of the code reduction. In the OMeta implementation the vanilla AST hierarchy is reused; we have included these numbers in the results for OMeta.

4.3 Structure

The cyclomatic complexity measurements are shown in Table 4. It shows the percentage of NCLOC that lives in units with a CC value higher than the threshold value 6. A lower percentage is better. The maximum in each row is in bold face.

At first sight, one would think that complexity is reduced: for instance, if we look at the percentages of the scanning and parsing component of the vanilla Java implementation, than we see that both percentages (31% and 22%) are higher than the ANTLR percentage (which combines scanning and parsing). "M" vs. C# shows a similar trend,

Component	Java	ANTLR	JavaScript	OMeta	C#	"M"
Scan	**31%**	–	9%	–	**31%**	–
Parse_c	**22%**	–	2%	–	17%	–
Parse_g	–	20%	–	**26%**	–	10%
AST	–	–	–	–	–	–
Check_c	–	–	8%	10%	**18%**	12%
Check_g	–	**23%**	–	–	–	–
Eval_c	13%	–	4%	–	27%	**33%**
Eval_g	–	**29%**	–	–	–	–
Misc	**32%**	–	–	–	–	–

Table 4: Percentage of NCLOC in units with cyclomatic complexity higher than 6. The maximum in each row is in bold face.

except the difference is much bigger (31% and 17% vs 10%). In both cases the difference can be accounted for by look-ahead code which heavily uses conditions.

In the case of OMeta, however, the picture is different: the grammar complexity is more than twice as large as the vanilla JavaScript scanner and parser. Inspection of the code leads to the conclusion that the vanilla JavaScript implementation uses extensive factoring in the scanner and parser thus reducing CC per method. In Table 3 this is confirmed by the large number of methods (234) in those components.

The AST components in the vanilla implementations have very low complexity. This can be understood from the fact that such hierarchies basically define an algebraic data type containing primarily getters, setters, and, possibly, an accept method for traversing using the Visitor design pattern. This code can be considered "boilerplate" code.

In terms of complexity static AST hierarchies as such can reduce the cyclomatic complexity of the code that uses them, for instance in the check and eval components. This is most visible in the C# vs "M" complexity results: the eval component of "M" has an atypical high complexity. This can be understood from the fact that "M" produces dynamically typed ASTs which can only be traversed using explicitly coded dispatch. If ASTs were to conform to a static hierarchy, the Visitor design pattern could have been used. Additionally, traversal for evaluation is not structure shy: you need to visit all node types. As a consequence 33% of NCLOC is spent in high complexity methods.

In the ANTLR implementation, the complexity of check and eval is relatively high because the real work is done in action code embedded in tree grammar productions. The CC of this code added to the CC of the productions themselves, thereby increasing the overall complexity.

Finally the utility code (Misc) in the vanilla Java implementation has a high complexity percentage due to a large `main` entry method with a high CC.

5 Qualitative Evaluation

After having presented the complexity of our implementations in terms of size and structure in the previous section, we return here to our research questions: does the use of DSL tools lead to implementations that are more maintainable?

Scanning and parsing. Considering the size metrics, the use of DSL tools clearly improves the maintainability of the implementations. In other words, one needs to understand less code. This certainly applies in the context of scanning and parsing, since the DSL tools used in this study are basically parser generators.

Using a DSL tool for scanning and parsing also reduces the structural complexity of parser implementations. DSL tools provide high-level primitives to describe the grammar of a language in a declarative way. As such they abstract over a lot of detail that has to be taken into account when writing scanners or parsers from scratch. The grammar codes represents the intrinsic complexity of parsing WAEBRIC, while the vanilla code exhibits overhead—accidental complexity.

However, the JavaScript/OMeta case strongly contradicts the above (Table 4), and we must conclude that using a DSL tool does not necessarily make the resulting implementation of the parser less complex. Even in a relatively complicated language as WAEBRIC, a DSL tool may increase structural complexity as compared to a well-factored vanilla implementation.

AST generation. Both ANTLR and "M" (and many other similar tools, see e.g., [11]) provide AST generation facilities. This obviates the need for maintaining AST boilerplate code. This is a very strong point in favour of DSL tools since AST hierarchies co-evolve with the grammar, which leads to the duplicate maintenance problem (if the grammar changes, the AST hierarchy has to change accordingly). This observation is compatible with the reduction in code size due to Abstract Data Type based code generators [2, 3].

AST traversal. The DSL tools used in this study do not provide extensive facilities for writing checkers and/or interpreters. Although both ANTLR and OMeta provide mechanisms to traverse AST nodes, the advantages of these features over the use of the Visitor pattern are less clear cut, especially in the case that structure-shy traversal is needed.

A case in point is the ANTLR implementation. Looking at the checker and interpreter we see that the complexity percentage of the checker and interpreter is 23%, and 29% respectively. With respect to size, the evaluator implemented with ANTLR is not even twice as large as the checker whereas in the other implementations, the evaluators are about 3 times (Java), 2 times (JavaScript), 10 times (C#), and 3 times ("M") as large as the checkers. One would expect evaluators to be larger than checkers since they tend to "do" more than checkers. The reason for this discrepancy between ANTLR and the rest is that ANTLR tree grammars have to be completely copied for each implementation component that has to traverse the AST. As a consequence, the checker cannot be implemented in a structure shy fashion. Our results show that the check component of ANTLR is even larger than in the vanilla implementation.

Additionally, this introduces a form of duplication that is especially hard to maintain: the tree grammars co-evolve with the normal grammar and with each other. The author of ANTLR is well aware of this limitation, as he notes in a footnote in [22] on page 193:

> Managing multiple copies of the same grammar but with different actions is an onerous task currently, and discussing the solutions is beyond the scope of this reference guide. [...]

An attempt at solving this problem in ANTLR is described in [23]. The current version of ANTLR also has a feature called *tree pattern matching* that can be used to implement, for instance, well-formedness checkers in a structure shy fashion [24].

Types and dispatch. Above we noted that AST generation in general is a good thing since it reduces code. However, in the ANTLR and "M" implementations discussed here, the generated ASTs were dynamically typed: the type of node is encoded as a field in a generic node object. This did not present a visible problem in the ANTLR implementation because tree grammars were used. In the "M" version this resulted in high complexity results for the check and eval components, because traversal had to be programmed using explicit dispatch (switch statements). A static AST hierarchy makes it possible to use the Visitor pattern. Such hierarchies could be easily generated from a grammar.

A default Visitor, possibly generated from a grammar too, has an additional advantage: it allows specialization to be used for implementing structure shy traversals. This can be done by inheriting from the default Visitor and overriding only the relevant visit methods. This way, explicitly coded dispatch as well as co-evolution of code that follows the structure of the grammar can be minimized.

A similar scheme is possible using OMeta, which has been applied in the implementation of the check component. There, the OMeta grammar that is used for parsing WAEBRIC is specialized (using "grammar inheritance") for implementing the check. Only the relevant productions are overridden with additional action code that is used for well-formedness checking of WAEBRIC programs.

5.1 Threats to Validity

This study is subject to a number of threats to validity. We distinguish internal validity from external validity. Internal validity is concerned with the validity of our conclusion with respect to our measurements. External validity concerns the question to what extent we may generalize the results.

The primary concern with respect to internal validity is the question of optimal implementation. In order to really derive valid data based on such implementation experiments the DSL tools in question should be used in the way they are intended to be used. The following actions have been undertaken in order to mitigate this risk. The implementations using ANTLR and OMeta have been reviewed by the respective tool authors and the "M" implementation has been reviewed internally at Microsoft Nederland. This at least gives some confidence that no obvious mistakes and/or idiosyncrasies exist in the implementations.

One issue of external validity is the representativeness of WAEBRIC as a DSL. Although the domain of WAEBRIC, XHTML generation, is very narrow, we think that WAEBRIC is a suitable language for experiments of this kind, for the following reasons. First, WAEBRIC has a non-trivial syntax, including expressions, statements and string interpolations. This already makes WAEBRIC more involved than many little languages around.

Second, WAEBRIC has many features that make it perhaps more like a programming language than a DSL proper: it includes the notions of conditional execution, looping, function call, closure, module import and recursion. This leads us to conclude that common aspects of programming languages have been exercised. Nevertheless, the issue whether this is representative for typical DSLs is uncertain, also because

DSLs are typically implemented using a compiler/code generator instead of an interpreter.

Finally, the fact that WAEBRIC has such a narrow domain has enabled that the developers of the implementation could exclusively focus on language implementation complexity; they have not been distracted by the complexity of the domain. In this case, the simplicity of the domain also makes it easier to test conformance of implementations

In this paper we have only investigated how properties of the source code might affect maintainability, but there are other factors that influence maintainability as well. First a tool has to be learned to be useful. This learning curve should not be underestimated, especially in the context of maintenance. Should the original developer(s) leave the company, someone else should learn the tool in order to keep the DSL implementation viable. This is even more a problem if documentation and/or support is non-existent or scarce.

Another aspect influencing maintainability is lack of IDE support. Code outliners, declare/use services, immediate feedback etc. may all make the maintenance less onerous. Additionally, without IDE support for debugging, error diagnosis may be hard. Especially, if the tool uses code generation, it can be difficult to relate static or dynamic errors back to their original location in the code.

A tool may introduce additional complexity by mixing domain specific notations with ordinary general-purpose language code, for instance by having action code in between arbitrary grammar productions. This also exacerbates debugging. IDE support, if any, is even more complicated, since now the composition of two languages has to be catered for.

5.2 Related Work

Empirical research on DSLs and DSL implementation is scarce. In [4] a case-study is presented between the use of a DSL vs an object-oriented framework in the domain of financial engineering. More recently, Hermans et al. [13] present a user study on the perceived value of a real life DSL in industry. These papers focus on the value of a DSL for its users and not on the (dis)advantages of specific implementation approaches. Van Dijk in his masters thesis focuses on changeability of a DSL-based web application, including the use of a DSL tool [6]. Changeability is another aspect of maintainability, next to understandability.

There is related work studying implementation approaches. For instance, [27] presents a case-study of post-design DSL embedding. This is primarily a qualitative experience report. More similar to our approach, is the work described in [17]. There, an empirical study of different implementation approaches is presented, comparing, for instance, embedding in Haskell, implementation using a language workbench (LISA), or using a parser generator (e.g. Yacc). The study combines computed metrics (size), user questionnaires for usability and runtime performance measurements to compare different implementations. Although this work has the same objectives as ours, its scope is much wider. We have focused exclusively on maintainability and properties of the source code.

Additionally, the example DSL used, Feature Description Language (FDL) [5], is much "smaller" than WAEBRIC: it is basically a notation for boolean propositions. The grammar of FDL contains 11 non-terminals, whereas our WAEBRIC grammars have between 133 (ANTLR) and 312 ("M") non-terminals. Finally, not all FDL implementations (by design) achieve the original DSL notation, whereas we required the implementations to have feature-complete parsers.

6 Conclusion

On the whole DSL tools do indeed increase maintainability of DSL implementations. Especially in terms of code size we have confirmed this hypothesis. Structurally the use of a DSL tool does not necessarily make the resulting implementation less complex, but it does usually remove the need for boilerplate structure. The structure of a parser remains intrinsically complex though, even when coded as a grammar.

Acknowledgements The Java and ANTLR implementations have been developed by Jeroen van Schagen. Nickolas Heirbaut developed the JavaScript and OMeta implementations. Jeroen van Lieshout worked at Microsoft Nederland to develop the C# and "M" versions. We additionally thank Jeroen Quakernaat and Dennis Mulder for their support at Microsoft Nederland. Finally, we thank Terence Parr and Alessandro Warth for respectively reviewing the ANTLR and OMeta implementations.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1st edition, 1986.

[2] M. Brand, P.-E. Moreau, and J. J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. *IEE Proceedings – Software*, 152:70–78, April 2005.

[3] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *J. Log. Algebr. Program.*, 59(1-2):35–61, 2004.

[4] A. v. Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39. Ilmenau Technical University, 1997.

[5] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.

[6] D. Dijk. Changeability in model driven web development. Master's thesis, Universiteit van Amsterdam, Master Software Engineering, 2009.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[8] M. Fowler. Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] R. Grimm. Declarative syntax tree engineering (or, one grammar to rule them all). TR 2007-905, New York University, November 2007.

[12] N. Heirbaut. Two implementation techniques for domain specific languages compared: OMeta/JS vs. JavaScript. Master's thesis, Universiteit van Amsterdam, 2009.

[13] F. Hermans, M. Pinzger, and A. van Deursen. Domain-specific languages in practice: A user study on the success factors. *Lecture Notes in Computer Science*, 5795/2009:423–437, 2009. Proceedings ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS) — Empirical Track.

[14] J. Hunter and B. McLaughlin. JDOM. http://jdom.org/.

[15] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[16] P. Klint, T. van der Storm, and J. Vinju. RASCAL: a domain specific language for source code analysis and manipulation. In *Proceedings of 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE, 2009.

[17] T. Kosar, P. E. M. López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[18] T. Kuipers and J. Visser. Maintainability index revisited. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007. Special session on System Quality and Maintainability (SQM).

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–321, 1976.

[20] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[21] Microsoft. "M" modeling language. http://msdn.microsoft.com/en-us/library/ee460940.aspx.

[22] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[23] T. Parr. The reuse of grammars with embedded semantic actions. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008.

[24] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009.

[25] J. Power and B. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[26] J. Resig. Bringing the browser to the server. http://ejohn.org/blog/bringing-the-browser-to-the-server/.

[27] A. Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. *Hawaii International Conference on System Sciences*, 9:281, 2002.

[28] A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[29] A. van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In D. Tamzalit, editor, *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, pages 41–49, Amsterdam, The Netherlands, March 2007.

[30] J. van Lieshout. Benchmarken van een dsl implementatie in "Oslo" en C#. Master's thesis, Universiteit van Amsterdam, 2009. In Dutch.

[31] J. L. van Schagen. Measuring the quality of domain-specific language implementation approaches: Java versus ANTLR,. Master's thesis, Universiteit van Amsterdam, 2009.

[32] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS'07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

Using DSLs for Developing Enterprise Systems

Margus Freudenthal

Cybernetica AS/University of Tartu
Tartu, Estonia

Abstract

This paper investigates the suitability of contemporary DSL tools in the context of enterprise software development. The main focus is on integration issues between the DSL tool, the DSL implementation and the rest of the enterprise system. The paper examines different scenarios for integrating DSLs into enterprise systems. A number of criteria for evaluating DSL tools are then extracted from these scenarios. These criteria are then used to evaluate five industry-strength DSL tools.

1 Introduction

Background and Motivation

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain[17]. Classical examples of DSLs are Unix Makefiles (build scripts), regular expressions (specifying text patterns), HTML (describing text layout) GraphViz (describing graphs). This paper investigates practical issues connected to using DSLs for developing enterprise information systems.

Before we can get into specifics, we need to define, what is "enterprise information system" (EIS). Finding authoritative definition for the term is quite difficult (if not impossible), but there seems to be rough consensus that the EIS is a system for integrating and coordinating business processes of a (usually large) organization. From the technical point of view, enterprise systems are characterised by the following properties.

- They are usually written in a language like Java or C# and make extensive use of frameworks (such as JEE and .NET) and middleware (application servers, enterprise service buses etc.).

- They may consist of set of interconnected modules that are built upon common architecture. Often there may be a benefit in applying the Software Product Lines[12] approach.
- They tend to be shallow, but wide. The application code in EIS itself does not have to be technically complex because the complex parts are implemented in the frameworks and application servers. Instead, most of the application code deals with implementing the concepts and processes of the organization.

The frameworks and middleware that are used to build EIS usually contain different graphical or XML-based DSLs for configuring the components. We call such DSLs "platform DSLs". Additional benefits can be gained from using DSLs for encoding business logic. Such DSLs are concentrated on a specific business domain, such as taxation, banking or medical domains.. This paper is mainly concerned with creating such application-level business DSLs.

When adopting DSL-based development process, one of the most important questions is tool support for developing DSLs. There exists a reasonable body of tools to assist in various aspects of creating DSLs. Examples of these are parser generators, code generators, transformation systems, IDE generators. The main question posed in this paper is: are the existing DSL tools suitable for use in enterprise software development?

Previous work

Published comparisons of DSL/DSM tools [15, 13, 11] have mostly discussed available functionality and ease of use. In particular, the set of tools included in comparison by Pfeiffer and Pichler[15] is similar to this paper. However, this paper differs from the Pfeiffer and Pichler paper in two important aspects. First, this paper only reviews tools that are suitable for industrial-strength software development. This excludes pure research prototypes and inactive open source projects. Additionally, this comparison includes tools that do not provide IDE functionality, but that can be used to create a working DSL implementation. Second, Pfeiffer and Pichler are mainly concerned by general characteristics of the tools (type of metamodel, type of code generator, etc.), whereas this paper targets the technical properties that are important when using the DSL tools for enterprise development.

den Haan[3] describes the development model and the roles that are involved in DSL-based software development. This model is reused in this paper and further developed to include roles related to deployment of DSL programs and product-line software development.

Contributions of this paper

This paper adds to the body of knowledge about domain-specific languages and enterprise information system development. The main contributions of this paper are the following.

- Section 2 discusses how the DSLs and DSL tools integrate with the rest of the EIS.
- Section 3 develops framework to evaluate DSL tools for suitability for enterprise software development.
- Section 4 presents brief overview of the evaluated tools.
- Section 5 presents the results of the evaluation.

Scope of this Work

Since the terms DSL and DSL tool can be used in quite a wide sense, we will first make matters more concrete by narrowing down the range of DSLs and DSL tools analyzed in this paper.

DSLs can take quite different forms and be implemented in quite different ways (see [16] for a taxonomy). This paper focuses on creating DSLs that

- use textual syntax;
- use custom syntax and a specially crafted parser (external DSLs) as opposed to being embedded into general-purpose programming language (internal DSLs);
- are integrated into the system as opposed to being implemented as a separate service (such as business rule engine). This paper focuses on languages and tools that implement light-weight components that can be natively called from the surrounding system;
- use active code generators[8] for generating code.

DSL tools are considered to be software programs that simplify some aspects of creating DSLs (such as parser generators, code generators, IDE generators) and that are advertised as DSL tools. Many software tools (including all the general-purpose programming languages) can, in principle, be used to implement DSLs. Parser libraries in high-level languages, such as Haskell or Prolog, can achieve the result comparable to using specialised parser generators[9]. In order to compare the tools on a more equal footing, this paper only looks at tools that are specifically aimed at developing DSLs.

2 Integrating DSLs into EIS

An important characteristic of an enterprise information system is that it is too big and complex to be completely described by a set of DSL programs. Instead, DSL-based EIS typically consists of components written in different DSLs and glued together by code manually written in general-purpose language, such as Java. Therefore, one important question is how the DSL code can be called from the rest of the EIS.

The biggest factor influencing the options for integrating DSL code and "glue" code is whether the DSL is interpreted or compiled. In general, the difference between interpretation and compilation can be fuzzy (for example, is Java compiled or interpreted?). In this paper, the line is drawn according to whether the DSL interpreter is part of the application code or part of the environment (virtual machine, hardware). For example, an application can be written in Python language and the DSL program translated to Python (or Python bytecode), which is then loaded into Python runtime. We consider this approach to be compilation, because the code is interpreted by the environment (Python runtime). However, if the DSL program is translated to form that is interpreted by the Python code in the application itself, then we consider it to be interpretation.

The integration options listed in this section mostly differ in the point of time when the DSL program is packaged/compiled and loaded into the system. From the DSL user's point of view the main difference between options is the versioning model of the DSL program. For example, if the DSL program is packaged with the rest of the source code of the system, then the DSL program should be versioned together with the rest of the source code.

There are two principal ways of deploying DSL programs that use interpretation.

- DSL program can be packaged with application source code as a text resource. This is the option that is used e.g. in JEE for various configuration and manifest files. With this approach the life-cycle (deployment schedule, versioning policy) of the DSL program will match the life-cycle of the other application code. Changing the DSL program involves redeploying of the whole application. Therefore, this approach is mainly suitable for technical DSL programs that change with the application code.
- DSL program can be loaded at runtime and stored in file or database. The program can either be in a source form or compiled to some kind of bytecode. The application can also provide environment for editing, testing and debugging the DSL programs. Because changing DSL programs does not involve changing or redeploying the rest of the application code, this approach is suitable for non-technical DSL programs that capture fast-changing business requirements.

For deploying compiled DSLs, there are three options for packaging and loading the DSL programs.

- DSL program can be compiled during the build process of the system. The compiled DSL program is then packaged and deployed together with the rest of the system. Processes for changing and deploying the DSL program are exactly same as for changing and deploying rest of the application code. The DSL program thus becomes integral component of the system that is integrated with other, possibly hand-written components. This approach is suitable for technical DSLs (essentially programming at higher level of abstraction).
- DSL program can be compiled separately and loaded into the system at run time (e.g. as a dynamically loaded plugin). The DSL program can be managed separately from the application code and can be used for describing business logic that changes more often than the application code. This option requires creating a special tool or build system that is able to compile the DSL program and package it for deployment.
- DSL program can be loaded into the running application and compiled by the application code. The compiling can be done in several steps, for example by first generating Java source code and then invoking the Java compiler. The compiled DSL program is stored in file or database and loaded at run time. Using this option requires packaging the DSL compiler with the application software. As with the previous option, the life-cycle of the DSL program is not tied to the life-cycle of the application code.

The main difference between the last two options is the environment where the DSL compiler is run. The latter option can be somewhat more complicated to implement because the runtime environment must also contain all the development tools and libraries needed for compiling the code. Also, invoking the compiler from the application code can be complicated and resource-consuming, especially if the compiler is implemented as an external program. However, embedding the DSL compiler into the application software can offer several benefits. First, it lowers the requirements for working environment of the business engineer[1]. If the application software includes a web-based DSL editing tool, the business engineer can create DSL programs using only a web browser. Second, if the DSL program comes from untrusted sources (e.g., end users customizing their user experience) then it is possible to analyze the DSL program before compiling to ensure it meets the requirements. If the DSL program is compiled before loading into the system, then the analysis becomes more difficult as it becomes necessary to reverse engineer the compiled code.

[1]Business engineer is the person who expresses business problem in DSL program. See [3] for full taxonomy of the roles in DSL-oriented development.

3 Evaluation Criteria

The evaluation criteria described in this section represent functional and non-functional requirements for the DSL tools that are necessary for developing enterprise information systems. Each criterion is given a short code that will be used to refer to it in later text.

First set of criteria concerns functionality of the DSL tool: what parts of the DSL tool chain does the tool provide? Reasonably complete tool support consists of

- [F-AR] abstract representation for the program (usually abstract syntax tree (AST) or metamodel is used),
- [F-PAR] parser generator (takes as input grammar description and produces parser that transforms the DSL program from concrete syntax to abstract representation),
- [F-CG] code generator or a templating engine (takes as input DSL program in abstract form and produces output source code (e.g. Java) or bytecode. The output source code can then be compiled to executable form),
- [F-CHK] program checker (verifies correctness of the program, for example references between program elements, data types, other constraints. Program checker usually works with abstract representation of the program),
- [F-IDE] IDE generator or generic IDE (development environment for the DSL. It can consist of text editor with syntax highlighting, outline view, navigation, automatic completion, refactoring support etc.).

It is possible to do without abstract representation of the program and instead translate directly between two concrete representations of the DSL program. However, this approach is only applicable to quite simple DSLs and does not scale to more complex DSLs with complicated transformations and correctness checks.

Besides functionality, we are interested in non-functional requirements for the DSL tools.

[F-PARA] The DSL tool should support the product line development by offering a way to create a DSL implementation that contains basic functionality and offers extension points where application-specific customisations can be applied. Examples of these customisations are set of objects that can be manipulated by the language and operations that can be applied to the objects. This criterion measures whether the tool allows creating customisable DSLs and composing DSL from several language modules.

[F-GEN-RESTRICT] The DSL tool should not impose artificial restrictions on the code generated from the DSL program. Ideally, it should be possible to generate code in any output language (e.g. Java, C#, XML).

[F-GEN-CHK] The DSL tool should provide mechanisms for ensuring correctness of the generated code. For example, when the code generator is configured with Java syntax, it ensures that only syntactically correct Java programs are generated.

[F-GEN-CPLX] The DSL tool should support creation of complex DSLs where mapping from source code to generated code is not straightforward: elements in source language are not in 1:N relationship with the elements in the target language. Additionally, validating the DSL program can require non-trivial and non-local analysis, such as resolving references between different parts of the input program. The support for complex DSLs can consist of the following features:

- support for compiling the DSL program in several steps where each step transforms program from one abstract representation to another and only the last step transforms the abstract representation to concrete output file. This also implies support for defining metamodels for representing intermediate steps of the program transformation;
- support for processing several input files together (for example, doing a whole program analysis);
- support for resolving references between input files.

If we evaluate whether the DSL tool can support the processes and integration options described in section 2, then we should look at the following features.

- [I-SEP] It must be possible to deploy the generated DSL IDE separately from the DSL tool, so that the roles of language engineer/transformation specialist[2] and business engineer can be separated. With this feature, business engineers can use the IDE for developing DSL programs, but cannot modify the language definition.
- [I-NOIDE] It must be possible to use non-visual features of the DSL tool (parser, code generator, program checker) separately from the DSL IDE. This allows creating processes that process the DSL programs without human intervention (for example, setting up server for doing continuous builds). More specifically,
  - [I-BUILD] it must be possible to integrate the non-visual part of the DSL implementation into build process and
  - [I-SYS] it must be possible to integrate the non-visual part of the DSL implementation into the running system.

[2]Language engineer creates the language description and transformation specialist creates the language implementation.

The difference between I-BUILD and I-SYS is that integrating DSL implementation into running system is more difficult. For example, invoking external applications from a multiuser system can be resource consuming. Also, it is very difficult to integrate software that has too many dependecies to external components or that makes too many assumptions about how the DSL implementation is invoked.

- [I-VCS] The DSL tool must support version control of the DSL programs. If the DSL uses human-readable textual syntax, this is very easy because storing and merging text files is supported by all the popular version control systems. However, if the DSL program is stored in some internal format (e.g. serialised XML representation), then the DSL tool must include explicit support for version control.

4 Evaluated Tools

Tool Selection Criteria

This paper evaluates suitability of DSL tools for industrial software development. Therefore, the evaluation is restricted to tools that are reasonably mature, are supported by community or company and are available to the public. These criteria are elaborated below.

- Maturity. The tool should have history of practical use for several years. It should have stable current release that can be used for developing production-level code. There should be cases of using the tool in several production systems (this requirement excludes pure research tools that have been developed and used only once or twice). The tool should be documented reasonably well and work without crashing or malfunctioning.
- Support. The tool should currently be in active development and new releases should appear periodically, in reasonable time intervals. There should be active user community that supports the tool via mailing lists or user forums.
- Availability. The tool should be available to general public under reasonable licence. Open source licences are preferable because they lower the risks caused by e.g. insufficient documentation or community support.

Stratego/Xt

Stratego/XT[3][1] is a language and tool set for program transformation. The toolkit provides facilities for defining concrete syntax of the DSL (parser generator) and applying transformations to the abstract representation of the DSL

[3]Available online at http://strategoxt.org/

program. The transformations are expressed in terms of rewriting rules that can be controlled by programmable strategies. The transformations can be used to accomplish tasks like

- translating DSL program into some other language (compiler or code generator);
- evaluating the DSL program (interpreter);
- analysing the DSL program (type checker, model checker, etc.);
- modifying the DSL program (refactoring tool).

The Stratego/XT toolkit is mainly targeted at the "non-visual" aspects of creating a DSL. Additionally, there is an IDE creation toolkit Spoofax/IMP[4][10] (currently under development) that can generate Eclipse plugins for Stratego-based DSLs. The IDE services are specified in a declarative manner using special DSLs. The Spoofax/IMP is built using the IMP toolkit.

Xtext

XText[5][7] is an Eclipse-based framework for creating textual DSLs. It integrates with the Eclipse Modelling Framework (EMF) and uses Ecore metamodel for describing the abstract representation of the DSL program. Based on the grammar description the Xtext creates parser, metamodel and Eclipse-based IDE. The generated IDE can be customised (e.g. by adding checks or specifying non-default rules for outlining DSL program) by writing Java code.

Xtext includes templating engine Xpand that can be used for simple code generation tasks. For complex program transformation tasks, model-to-model transformation tools, such as QVT or ATL, can be used. Xtext is bundled with Xtend language that can also be used for model-to-model transformations. By default, Xtext supports only one metamodel for language and does not offer direct support for using several transformation steps for generating code. Creating complex transformations is possible, but requires combining Xtext with other tools.

IMP

IMP[6][2] is an Eclipse-based framework, similar to the Xtext. It is based on different parser generator and uses standard Java classes for abstract representation of the DSL program. The IMP requires more programming to create an

[4]Available on-line at http://strategoxt.org/Stratego/Spoofax-IMP
[5]Available online at http://www.eclipse.org/Xtext/
[6]Available online at http://www.eclipse.org/imp/

IDE for a simple DSL than Xtext, but is more customisable (even the parser and abstract representation of the program can be replaced).

IMP is concentrated on creating an IDE for the DSL, it does not include explicit support for creating language implementation (e.g. interpreter or compiler). This must be done manually by writing Java code or by using other tools.

ANTLR and StringTemplate

ANTLR[7][14] is a parser generator. It takes as input description of a DSLs syntax and produces parser that recognises the DSL. By adding actions to the parser, it is possible to build a translator or interpreter. ANTLR offers facilities for parsing the DSL program into its abstract syntax tree and then performing operations on the tree.

StringTemplate[8] is a templating engine that works well with ANTLR and that can be used for generating code from the DSL program. Together with ANTLR it is possible to create a complete implementation (parser and code generator) for a simple DSL. ANTLR itself does not contain mechanisms for implementing complicated languages, but this can be done by combining ANTLR with other tools and/or writing the compiler manually.

Meta Programming System

Meta Programming System (MPS)[9][4] uses different approach to DSL creation than the previous systems. It is based on so-called language workbench [5] paradigm which uses mix of techniques from textual and graphical DSLs. The DSL program is displayed on screen as text, but the user directly edits the abstract representation. This technique is called projectional editing[6] and is usually used for editing graphical DSLs. When creating a DSL in MPS, one usually starts with meta-model and editor. MPS also provides support for defining constraints, type systems and code generators.

MPS version 1.0 was released quite recently and because it does not have long history and several uses in commercial projects, it does not, strictly speaking, fulfil the maturity requirement presented in tool selection criteria. However, it is quite stable and usable and is used to build complex applications[10]. Because MPS also represents an interesting approach to DSL tools, it is included in the evaluation.

[7] Available online at http://www.antlr.org/

[8] Available online at http://www.stringtemplate.org/

[9] Available online at http://www.jetbrains.com/mps/

[10] Jetbrains, the creator of MPS has released issue tracking software YouTrack that is created using MPS.

5 Evaluation Results

In this section, we compare the tools described in the previous section with respect to criteria defined in section 3. Table 1 presents the detailed evaluation results.

From the functionality point of view, the Stratego and MPS offer the most features. Xtext offers also quite complete solution that can be applied to simple DSLs. For simple DSLs, the Xtext also offers very good usability. It is very easy to learn and creating simple languages with parser, IDE and code generator requires very little effort. The downside for Xtext is its lack of support for creating complex DSLs. ANTLR and IMP are somewhat special-purpose tools. IMP is mainly IDE generator and ANTLR is parser generator which can also be used for creating simple code generators.

When looking at the integration options offered by the DSL tools, the picture is quite different. The MPS language runtime (code generator, checker) is tightly integrated with the MPS graphical workbench and it seems that there are no plans to separate the language runtime from the IDE[11]. This complicates using the MPS for large-scale software development, because use of the DSL tool cannot be automated (e.g. by compiling the DSL program as part of the automated build process).

Stratego and IMP currently rely on invoking external executable. Stratego programs (language implementations) are by default compiled to native code, IMP uses external parser generator LPG to compile DSL implementation. Invoking external programs is generally feasible during the build process, but can cause performance issues when used from an on-line application that must serve several concurrent users. Luckily, these tools require executing external programs only when compiling DSL implementation. The DSL implementation itself is native Java application (IMP) or can be run under Java-based runtime (Stratego) so that embedding the DSL implementation into the running system is still possible[12].

Xtext is a native Java-based implementation, but it is heavily dependent on the Eclipse infrastructure. It is possible to compile the Xtext-based language implementation and invoke the parser and generator from the build system, but this requires downloading about 40MB of dependencies. This is acceptable for build system, but can be too large overhead to add to the application.

ANTLR stands out from the crowd because it introduces the least amount of run-time and compile-time dependencies. Embedding ANTLR-based DSL implementation into the build process or into the application is straightforward.

[11] See http://www.jetbrains.net/devnet/thread/283315

[12] However, the Java-based Stratego runtime is considerably slower than the native implementation. This can cause problems for processing large DSL programs.

Feature	Stratego	Xtext	IMP	ANTLR	MPS
F-AR	AST	ECore	Java	AST	Custom
F-PAR	Yes	Yes	Yes	Yes	N/A [1]
F-CG	Yes	Yes	No	Yes	Yes
F-CHK	Yes	Partial [2]	No	No	Yes
F-IDE	Yes [3]	Yes	Yes	No	Yes
F-PARA	Yes	Yes	No	Partial [4]	Yes
F-GEN-RESTRICT	Yes	Yes	Yes	Yes	Yes
F-GEN-CHK	Yes	No	No	No	Yes
F-GEN-CPLX [6]	Yes	No	No	No	Yes
I-SEP	Yes	Yes	Yes	N/A	Partial [7]
I-NOIDE	Yes	Partial [8]	Yes [9]	N/A	No
I-BUILD	Yes	Partial [8]	Yes [9]	Yes	No
I-SYS	Yes	No	Yes [9]	Yes	No
I-VCS	N/A [10]	N/A [10]	N/A [10]	N/A [10]	Yes [11]

1. Because MPS uses projectional editing, it does not need parser for converting from plain text to AST – the program is always stored and edited in the AST form.
2. Xtext automatically performs reference validation based on grammar definition.
3. Stratego does have IDE generator Spoofax/IMP. However, it is currently not very mature.
4. ANTLR does have mechanism for grammar inheritance, but it only supports some use cases for reuse and parametrisation.
5. Stratego, ANTLR, Xtext and IMP use generators that can generate any plaintext languages.
6. In principle, all the tools can be used for implementing complex DSLs (any Turing-complete programming language would suffice). What is compared here is the explicit support for creating several metamodels, starting from the model corresponding to AST and ending with model corresponding to generated code. Compiling the DSL program thus becomes series of model to model transformations, followed by final model to text transformation for generating the actual output.
7. For MPS, the language IDE still requires full language workbench. However, it is possible to export the language as read-only package to prevent its accidental modification by the DSL user.
8. Xtend can be used outside IDE, but it requires installing large number of Eclipse libraries.
9. Main focus of IMP is the IDE generator. The DSL runtime support consists of LPG parser generator.
10. Because tools store DSL programs as text, there is no need for explicit VCS support.
11. MPS stores all files in structured XML format and therefore requires custom VCS support for merging and diffing. MPS supports all the VCS systems supported by IntelliJ IDEA.

Table 1: Tool comparison matrix

6 Conclusions

In this paper we have analysed the requirements and needs that arise when using DSL-oriented approach for developing enterprise information systems. Based on these requirements, we evaluated five DSL tools for suitability in the enterprise development. All the tools offer good functionality and ease of use. However, there are several issues when integrating the tool with the overall development environment. In general, the IDE-based tools assume that all the work is performed using the IDE. This makes it difficult to integrate the tool into bigger systems. Our recommendation for the tool vendors is to consider modularising the tool so that it can be invoked from different contexts. It seems that currently there exists no perfect tool that has all the features and also offers all the reasonable integration options.

Our work on this topic continues. The main fields of interest are better understanding of the requirements posed by the EIS development and evaluating usability of the DSL tools for creating complex DSLs.

References

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.

[2] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton, Jr. Imp: a meta-tooling platform for creating language-specific ides in eclipse. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 485–488, New York, NY, USA, 2007. ACM.

[3] Johan den Haan. Roles in Model Driven Engineering. http://www.theenterprisearchitect.eu/archive/2009/02/04/roles-in-model-driven-engineering, 4 February 2009.

[4] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/, November 2004.

[5] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, May 2005.

[6] Martin Fowler. Projectional Editing. http://www.martinfowler.com/bliki/ProjectionalEditing.html, January 2008.

[7] Peter Friese, Sven Efftinge, and Jan Köhnlein. Build your own textual DSL with Tools from the Eclipse Modeling Project. http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html, 18 April 2008.

[8] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functctional Programming*, 8(4):437–444, 1998.

[10] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Domain-specific languages for composable editor plugins. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, April 2009.

[11] Benot Langlois, Consuela elena Jitia, and Eric Jouenne. DSL Classification. In *The 7th OOPSLA Workshop on Domain-Specific Modeling*, October 2008.

[12] Linda M. Northrop and Paul C. Clements. A Framework for Software Product Line Practice, Version 5.0. Technical report, Software Engineering Institute, July 2007.

[13] Turhan Özgür. Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the context of Model-Driven Development. Master's thesis, Blekinge Institute of Technology, 22 January 2007.

[14] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[15] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *The 8th OOPSLA Workshop on Domain-Specific Modeling*, 19 October 2008.

[16] T. Sloane, M. Mernik, and J. Heering. When and how to develop domain-specific languages. Technical Report SEN-E0309, CWI, 2003.

[17] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

Formally Specified Type Checkers for Domain Specific Languages

M.G.J. van den Brand, A.P. van der Meer, A. Serebrenik, and A.T. Hofkamp

Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{m.g.j.v.d.brand,a.p.v.d.meer,a.serebrenik}@tue.nl
a.t.hofkamp@tue.nl

Abstract

An important part of the usability of a programming or specification language lies in the presence of supporting tools that are provided with the language, e.g., compilers, debuggers and simulators. Development of such tools for domain-specific languages imposes a number of specific requirements pertaining to evolvability of the tools and suitability of these tools for domain experts with little or no programming experience. We developed an MSOS-based approach to automatic generation of formally specified type checkers for these languages. Our approach has been applied to Chi, a high level specification language for describing concurrent systems. The resulting type checker has been successfully integrated in the tool chain of the Chi language.

1 Introduction

An important part of the usability of a programming or specification language lies in the presence of supporting tools that are provided with the language, e.g., compilers, debuggers and simulators. In the case of domain specific languages (DSLs) development of such tools requires addressing a number of specific challenges: tools should be suited for understanding by domain experts, support evolution and facilitate decision making by language developers.

- Correctness of the tools supporting DSLs should be clear to the *domain experts* usually having little to no programming experience. As opposed to traditional testing we would like the domain experts to establish correctness of the tools prior to their implementation.
- Whenever the DSL changes as the part of language *evolution*, the tools have to be adapted to stay compatible to the DSL, and with each other.

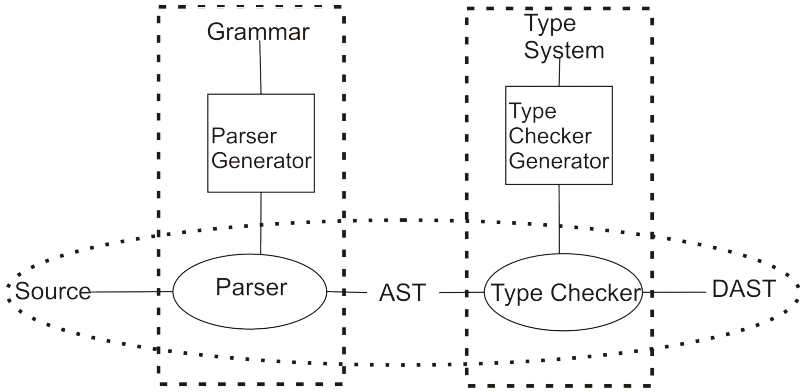


Figure 1: Basic structure

- Development of a DSL is a creative process and consequences of a syntactic or a semantic change might not be obvious to the language developers. *Decision-making by the language developers* should therefore be supported by easily modifiable prototype tool-set.
- Moreover, one should be able to easily integrate the tool with other tools already available for the DSL.

In this paper we focus on type checkers, being a part of the compiler tool chain. The basic structure of the methodology proposed is shown in Figure 1. The bottom ellipse shows part of the basic structure of a compiler. The starting point is the source code of the program to be compiled, which is then parsed. The resulting Abstract Syntax Tree (AST) is then processed by the type checker, which produces a Decorated AST (DAST). Here, the word "decorated" refers to additional type information that is added to various nodes in the tree.

A type checker extracts information from the AST according to a *type system*, description of the desired behaviour of the type checker. A type system can be seen as a set of rules mapping the AST fragments to a predefined set of types. Traditional approaches to type checker construction fail to meet the requirements above: e.g., manually implemented type checkers are hard to modify, that is, they are ill-suited for evolution or decision-making support.

Therefore, we propose to *generate a type checker from a formal specification*. Our work has been inspired by the fact that the early parsers [21] have been hand-written, while the modern ones [22] are usually generated by a parser generator, based on a grammar. This way, the parser is easier to create and maintain, though it also creates a dependency on the parser generator and the resulting parser might be less customizable.

As Figure 1 shows, we advocate a similar approach for type checkers. Similarly to the grammar describing syntactical well-formedness rules of a language,

a type system is a set of well-formedness rules describing which terms are correct with respect to types. Once the type system has been formally defined, using a type checker generator one can generate a type checker in the same way parsers are generated from grammars by parser generators.

The remainder of the paper is organized as follows. In Section 2 we review different approaches to specifying a type system, i.e., to describing the intended type checker behaviour, and their applicability to domain specific languages. Section 3 discusses ways of producing a type checker implementation based on the type system. Similarly to generated parsers we expect generated type checkers to be easier to create and maintain but more difficult to fine-tune. Next we apply our methodology to develop a type checker for Chi, a high level specification language for describing concurrent systems (Section 4). In Section 4.4 we review the case study and show how the challenges listed above have been addressed. Finally, Section 5 discusses related approaches, concludes and sketches directions for future work.

2 Type System: Specification of a Type Checker

2.1 Existing Approaches

Numerous ways of specifying a type system can be found in the literature. For example, type systems can be described using *denotational semantics* [13, 31]. Denotational semantics is based on functions that give the meaning or denotation of a program by linking inputs to the appropriate outputs. In this case, each node of an abstract syntax tree is linked to the appropriate type. On the other hand, [29] uses Structural Operational Semantics (SOS) to represent type systems. Unlike the denotational semantics, *operational semantics* gives a meaning to terms by defining an abstract machine consisting of states connected by a transition relation. In SOS, the transition relation is defined by sets of rules allowing a step from one state to another if the rule's preconditions are met. When used in type systems, the initial state is the AST, and the resulting state should be the desired type, or a DAST with type information in some or all nodes. Another form of operational semantics are evolving algebras. An *evolving algebra* consists of two parts, a partial, many-sorted algebra that describes the given program as the initial state, and a set of transition rules that describe transitions that allow state changes. Fundamentally similar in approach, evolving algebras mainly differ from SOS-based formalisms in that an imperative-programming style rather then the more set-theoretical style used by SOS. Industrial implementations are usually based on *attribute grammars* [1, 32, 16]. Attribute grammars consist of grammars where attributes have been defined for some or all nodes, usually in terms of other attributes. During type checking, attributes are evaluated as needed until the values for the desired attributes, like the type of an expression, is known. Finally, one can develop a special *domain specific language* for type systems [19].

We considered all the specification methods above as a basis for our type

checker generator. *Denotational semantics* are clean and implementation-independent, but involve heavy mathematical machinery often surmounted by a cryptic syntax [28]. Minor changes in the language might require a complete rewriting of the specification [25]. Since we target domain-specific languages we need a formalism that would be comprehensible for domain experts without background in formal modelling. As rapid evolution is not uncommon for domain-specific languages, the rigidity of the denotational semantics becomes a major problem. *Attribute grammars* are easier to understand, but focus on how to calculate type values, instead of on what the type should be. Moreover, attribute grammars usually require a mental switch from the domain experts: even auxiliary data structures need to be implemented through production rules [17]. *SOS* is closer to natural reasoning, but can be very verbose and specifications tend to have high levels of coupling between rules. Finally, while developing a separate *domain specific language* is a valid option for type checker specification, the language developed cannot be reused for other forms of static semantic analysis. Moreover, application of such language requires a special learning effort from the domain experts. *Evolving algebras* share many of the problems above, in addition to being, in our opinion, less clear in general.

Hence, we choose to use an improved form of SOS, Modular Structural Operational Semantics (MSOS) [26, 27]. In our previous work [11] we have shown that MSOS is well-suited for specifying type systems of evolving languages.

2.2 MSOS

MSOS was developed by Peter Mosses [27] in an attempt to simplify the notation of SOS and to reduce the coupling between rules. In particular, as an SOS specification grows, more and more rules require the knowledge of specific context information, such as lists of declared variables and constants. Such context information is represented in SOS by the *environment* that usually consists of a number of components describing, e.g., values of variables, types of variables, labels declared and the state of the heap. SOS requires the component structure of the environment to be indicated on each and every rule. Hence, as the number of environment components grows, *all* rules become more complex, although some of the rules do not need the knowledge of the environment at all. Additionally, if the environment contains components that are not mentioned by a rule, that rule cannot be applied, which makes it harder to combine or modify SOS specifications. MSOS removes this problem by storing environment information in a more flexible and implicit manner using transition labels.

A *specification* in modular structural operational semantics (MSOS) is a structure of the following form. $\mathcal{M} = \langle \Omega, Lc, Tr \rangle$ where Ω is the signature, Lc is a label category declaration and Tr is the set of transition rules. In the *signature*, the symbols and constructs of the language are defined. The *transition relation* defines rules consisting of a conclusion and a set of preconditions. A precondition can be a transition itself, a predicate or a label expression. If there are no preconditions for a given rule, it is called *simple*; otherwise the rule is called *complex*.

A set of MSOS rules defines an arrow-labelled transition system, where the states are represented by the nodes of a category, and the labels and transitions by the the arrows of that category. Note that this implies that transitions can be composed in an associative way and that there are identity transitions for each state. Details about the formal definition of MSOS can be found in [8].

Example 2.1 *Consider the following fragment of the Chi type system, consisting of typing rules for integers, reals, variable references, and two operators: a unary operator demonstrating the hybrid nature of Chi and a binary operator demonstrating polymorphism.*

$$1\frac{\text{integer}(I)}{I \Longrightarrow \textbf{int}} \qquad 2\frac{\text{real}(R)}{R \Longrightarrow \textbf{real}} \qquad 3\frac{(ID, T) \in statenv}{ID = \{statenv, \ldots\} \Rightarrow T}$$

$$4\frac{(ID, \textbf{real}) \in statenv \qquad (ID, \textbf{cont}) \in dynenv}{\text{dot}(ID) = \{statenv, dynenv, \ldots\} \Rightarrow \textbf{real}}$$

$$5\frac{E_1 = \{X\} \Rightarrow T_1 \qquad E_2 = \{X\} \Rightarrow T_2 \qquad \max(T_1, T_2) = T_3 \qquad \max(T_3, \textbf{real}) = \textbf{real}}{\text{less}(E_1, E_2) = \{X\} \Rightarrow \textbf{bool}}$$

Rules 1 and 2 define the types for integer and real constants. Independent of the current environment these constants have types **int** *and* **real**, *respectively. Independence of the environment is represented by the label "...", which can be interpreted as a* "don't care".

Rule 3 implements variable references. This rule depends on the environment component that contains information on the type of variables, statenv. *We regard* statenv *as a set of identifier-type pairs. If the given identifier occurs in one of the pairs of* statenv, *then the type of the identifier is provided by the second part of the pair.*

Rule 4 implements the `dot` *operator, which is used in Chi to access the current value of the derivative of a given variable. In Chi the* `dot` *operator is applicable only to variables of type* **real** *explicitly marked in the specification as continuous, i.e., evolving in time according to a (differential) equation. To record the information about the behaviour of variables in time we use the* dynenv *environment component. The* dynenv *component is similar to* statenv, *except that it links variables to their so-called* dynamic type, *such as* **cont**, *for continuous variables, and* **disc**, *for discrete variables. The actual check is implemented in the same way is in the previous rule.*

Finally, Rule 5 defines the result type of the `less` *operator. As one would expect, this operator takes two numbers and returns the smaller of the two. In order for this operator to be applicable, both operands must have valid types and these types should be numerical. The two transition preconditions,* $E_1 = \{X\} \Rightarrow T_1$ *and* $E_2 = \{X\} \Rightarrow T_2$, *indicate that we can determine a type,* T_1 *and* T_2, *for each one of the operands* E_1 *and* E_2 *respectively. Preconditions* $\max(T_1, T_2) = T_3$ *and* $\max(T_3, \textbf{real}) = \textbf{real}$ *ensure that the arguments are of numerical types. Based on a partial order on types, an auxiliary function* `max`

returns the larger of two types, if they are comparable. In the example we assume **int** $\prec$ **real**. *If the* `max` *function returns a type* T_3 *and that type is not larger than* **real**, *we conclude that* T_1 *and* T_2 *are numerical types, and hence,* E_1 *and* E_2 *can be compared with the* `less` *operator. If this is the case, the type of the comparison expression is* **bool**.

2.3 MSDF

MSOS Definition Format (MSDF) is a textual representation of MSOS, that is easier to transform and manipulate. In addition to MSOS rules, MSDF also supports declarations, formulas and modules. *Declarations* describe algebraic data types and variables of those types. In our case, algebraic data types are used to represent the structure of the input tree, the type values and (possibly) the output tree. *Formulas* can then be used to give initial values to the variables or link them to each other. In a type system definition, initial values can be used to initialize the environment, e.g., by introducing default functions. Finally, *modules* increase readability of the semantic rules by allowing the related rules to be stored together. Modules can be imported as a whole into other modules.

```
1   see Bool.
2   see Id.
3   see Int.
4   see Real.
5
6   E:Exp  ::= less(Exp,Exp) | dot(Id) | Id | Int | Real.
7   T:Type ::= bool | int | real.
8
9   I:Exp ===> int.
10
11  R:Exp ===> real.
12
13  lookup(ID,STATENV) = T
14  ----------------------------
15  ID:Exp ={statenv=STATENV,...}=> T .
16
17  lookup(ID,STATENV) = real, lookup(ID,DYNENV) = cont
18  ----------------------------
19  dot(ID):Exp ={statenv=STATENV, dynenv=DYNENV,...}=> real .
20
21  E1 ={...}=> T1, E2 ={...}=> T2, max(T1,T2) = T3, max(T3,real) = real
22  ---------------------------
23  less(E1,E2) : Exp ={...}=> bool .
```

Figure 2: MSDF specification of a type checker for a fragment of Chi.

Example 2.2 *Consider the MSDF specification with rules for the type system presented in Figure 2. This module consists of three main parts,* imports*, in Lines 1–4,* declarations*, in Lines 6–7, and* transition rules *in Lines 9–23. The imports are part of the modular structure of MSDF, and allow transition rules and declarations from other modules to be used in this module. In our case, we import generic definitions for identifiers* `Id` *and basic data types* `Bool`*,* `Int` *and* `Real`*. These modules also define variables for these types: variables with names consisting of* `ID` *(*`B`*,* `I`*,* `R`*) followed by zero or more digits are considered to be of type* `Id` *(*`Bool`*,* `Int`*,* `Real`*, respectively). Note that the data type definition method in MSDF is not suited for efficient definitions of the integers and the reals, so* `Int` *and* `Real` *are actually defined outside MSDF.*

Lines 6–7 contain declarations for the types `Exp` *and* `Type` *and variables of those types. The first defines possible expression constructs, in the form of a list of options separated by the "|" symbols. An expression in the listing above is, therefore,* `less` *applied to two expressions,* `dot` *applied to an identifier, an identifier, an integer or a real number. "*`E:Exp`*" means that variables with names consisting of* `E` *followed by zero or more digits are considered to be of type* `Exp`*. In the next line, the type* `Type` *defines three constants representing the basic types:* `bool` *for boolean values,* `int` *for integer values, and* `real` *for real values. As above, variables with names starting with* `T` *and followed by one or more digits are of the type* `Type`*. We stress that while* `Bool`*,* `Int` *and* `Real` *are sets of values,* `bool`*,* `int` *and* `real` *are constants representing the types of these values.*

Lines 9–23 contain the transition rules. The first two (Lines 9 and 11) are simple rules, without explicit preconditions corresponding to Rules 1 and 2 of Example 2.2. Implicitly, the types of the variables used restrict their values to integer and real constants, respectively. The next three rules (Lines 13–15, 17–19 and 21–23) are the counterparts of Rules 3–5. In the case of the rules at Lines 13–15 and 17–19, the preconditions are based on the built-in `lookup` *function. Given a list of key-value pairs and a key that occurs in the one of the pairs, this function returns the corresponding value. If the key does not occur in the list of pairs, a special value "none" is returned. The* `lookup` *function is used to inspect the environment components, such as* `statenv` *and* `dynenv`*. The rule in Lines 13–15 reads, thus, "If* `ID` *is known to have type* `T` *in the static type component of the environment, any reference to* `ID` *also has type* `T`*"; and the rule in Lines 17–19 "If* `ID` *is known to be a continuous real variable, then application of the* `dot` *operator results in a value of type* `real`*". The last transition rule (Lines 21–23) is the counterpart of Rule 5 in Example 2.2.*

3 Implementation of the type checker

Once the type system has been formally specified, the next step in the approach is the creation of a component to apply the type system to AST. There are three general ways to obtain this component: the specification can be implemented by hand, executed by an interpreter or translated (compiled) to a different language already supported by an execution mechanism. Note that for the

implementation of parsers all three options are used. Indirect implementation, i.e., interpretation and generation, however, seem to be more promising for rapidly evolving languages. Based on this, we want to explore the two options pertaining to an indirect implementation of a type checker. That still leaves a choice to be made: do we focus on interpreted or generated type checkers?

Intrinsically, there is little reason to prefer an interpreted type checker over a generated implementation or vice versa. Based on concerns over the efficiency of the resulting type checker, and our existing experience with the ASF+SDF Meta-Environment, we chose to try a generative approach first. In order to make our transformation, we first needed to identify our source and target language, which should both be textual. As our source language we chose MSDF, as discussed in Section 2.3. This leaves the target language and the transformation technique to be determined.

3.1 Choice of the target language

In order to be used as a part of a tool set, the resulting type checker must communicate with other tools, so effective communication facilities are a necessity. In particular, the type checker and the tool that provides it with input, usually the parser, must agree on the structure of the AST. Additionally, the output of the type checker must be communicated with other tools, for example when errors have to be reported to the user through an existing user interface. This is straightforward when the type checker and the other tools share a common implementation language, or the implementation languages have some dedicated communication facilities such as native interfaces. Using languages that are already used by other components also reduces the need for additional libraries and simplifies installation. If direct communication is not possible, indirect communication, for example using the file system, is an option.

3.2 Choice of the transformation technique: ASF+SDF

ASF+SDF [10] is a term-rewriting language based on the Syntax Definition Formalism (SDF). ASF+SDF has numerous applications [9]. Of specific interest for our approach are the applications of ASF+SDF to program transformation and code generation, including the migration of legacy databases to relational databases together with the adaptation of the corresponding program code [14] and restructuring of Cobol programs [12, 33]. Hence, we have chosen for ASF+SDF due to its proven success record in application of program transformation in industrial cases. Examples of other languages that could have been used are Stratego[34] and Tom[4].

4 Case Study

Based on the methodology proposed have implemented a type checker for Chi 2.0 [6, 20], a high level specification language for concurrent systems.

4.1 Chi 2.0

Chi 2.0 is a high level specification language for describing concurrent systems. The language has been applied to solve real world industrial-sized problems successfully, e.g. [30, 7]. Originally created to design control of manufacturing systems, nowadays it is used to design control systems for complex high tech machines [3] as well, using model-based engineering and supervisor synthesis.

The language allows specification of pure discrete-event behaviour (where changes only take place during events), pure continuous time behaviour (where values of variables are decided by equations), or hybrid behaviour (a combination of discrete-event and continuous time behaviour). An example of the latter is for example a bottle filling line, where a discrete supervisor is used to control a liquid flow. An actual Chi specification consists of one or more parallel processes, that can communicate via synchronous actions, channels or shared variables, and differential equations that describe continuous behaviour.

With each new generation of the language, new concepts are added and existing ones change. Also, the application domain of the language is slowly shifting, causing more changes. In earlier versions a basic type system was sufficient. As the scale of the systems grows, more features are needed to express the processes in the system and to keep them maintainable. These include additional primitives, like matrices and vectors, alternate algebras, such as max-plus algebra, and stronger type derivation, such as implicit type widening, overloading, and templates. All these changes increase the effort to implement a type checker. The runtime semantics of Chi is formally defined in process algebra, but in previous versions the type system was unformalized. To keep the costs limited, an alternative approach is needed.

4.2 Target language: Pyke

As discussed in Section 3 we have opted for a generational approach. Therefore, one has to choose a target language, language of the generated type checker implementation. Since the type checker has to be integrated in the entire suite of the Chi tools, as target language the implementation language of the Chi tools was chosen, i.e., Python. In addition to a smooth integration with other Chi tools, we can benefit from a direct mapping of the MSOS labels to the concept of Maps native to Python. However, Python does not natively support the backtracking that is needed for cases where there are multiple transition rules that can be applied. Rather than add this ourselves, we used Pyke [18].

Pyke is a Prolog-inspired inference engine operating on backward-chaining rules that can interact directly with Python. Backwards-chaining rules are uniquely *identified* by means of labels and consist of the *conclusion*, indicated by `use`, and a (possible empty) set of *premises*, indicated by `when`. Identifiers can be used, e.g., to refer to rules during error reporting. Both the conclusion and each premise is a term. During the derivation, the Pyke inference engine maintains a set of *goals*. For each goal the engine looks for a rule with the conclusion matching it. If such a rule is found, the goal is replaced by the

set of premises corresponding to the conclusion. According to the result of the matching, variables appearing in the goal might be replaced by the actual values.

4.3 Transformation implementation

A crucial step in the process of generating the type checker is the ASF transformation that constructs the actual type checker code based on the MSDF specification. The result of the transformation is a set of Pyke rules, that, combined with the type-system-independent supporting Python code that provides support functions, like `max`, implement the type system as described in MSDF.

4.3.1 Imports

Recall that the main differences between MSDF and MSOS pertain to the introduction of modules and corresponding imports, and declarations. As a preprocessing step preceding the transformation we eliminate the imports by collecting all MSDF modules into a single specification. This step reduces the number of constructs that have to be handled and allows duplicate rules to be removed or combined before the transformation is carried out.

4.3.2 Formulas

Once the imports have been removed, only three constructs remain to be handled: formulas, declarations, and transition rules. Formulas are translated to Python and added to the knowledge base directly, to allow re-use of values. Declarations and transition rules are implemented in Pyke using backwards-chaining rules, discussed in Section 4.2, and form the main part of the transformation.

4.3.3 Declarations

Recall that declarations describe algebraic data types, and the main operation involving them is testing whether a given term matches a certain data type. When transforming MSDF declarations to Pyke we create a separate backwards-chaining rule for each case in a declaration. Hence, for the declaration `E:Exp ::= less(Exp,Exp) | dot(Id) | Id | Int | Real` (Line 6 in Figure 2) we create five backwards-chaining rules. The rules created are similar and we illustrate only one of them, corresponding to `dot(Id)`:

```
1  labela
2      use Exp((dot,$la))
3      when Id($la)
```

Here, `labela` in Line 1 is the rule identifier. In Line 2 the keyword `use` introduces the conclusion of the rule, while in Line 3 the keyword `when` is followed by the premises of the rule. In this rule there is only one premise. The rule

reads "`(dot,$la)` is an expression if `$la` is an identifier". The underlying assumptions are that the AST is represented by means of tuples (in this case the operator `dot` and the variable `$la`) and that `$` prefixes a Pyke variable.

4.3.4 Transition rules

Finally, we have to create backward-chaining rules that implement the transition rules of MSDF. Each transition rule is translated to exactly one backward-chaining rule, where the conclusion of the MSDF transition rule corresponds to by the conclusion of the backward-chaining rule, and the preconditions of the transition rule correspond to the premises of the backward-chaining rule. Unlike the preconditions of a transition rule, the premises of a backward-chaining rule are processed in a fixed order. Hence, in our transformation we take care that in the backward-chaining rule the input types are checked first, then the preconditions and finally the label is updated and the result is constructed. To illustrate the generation consider the following fragment of Figure 2:

```
21  lookup(ID,STATENV) = real, lookup(ID,DYNENV) = cont
22  ----------------------------
23  dot(ID):Exp ==={statenv=STATENV, dynenv=DYNENV,...}=> real .
```

The corresponding backward-chaining rule is

```
1   labelb
2       use trans((dot,$ID), $LABEL, 'real')
3       when
4           first
5               Id($ID)
6           $STATENV = $LABEL['statenv']
7           'real' = $STATENV.get($ID)
8           $DYNENV = $LABEL['dynenv']
9           'cont' = $DYNENV.get($ID)
```

Here, `labelb` in Line 1 is a generated rule identifier. As above, in Line 2 the conclusion is introduced, in this case a transition from the parse tree `(dot,$ID)` to the type `real`. Starting from the following line the premises are listed. First, we test the implicit precondition that the argument that is given for the operator must be valid, i.e., that it is an identifier. This check is delegated to declaration rules like the one generated earlier. We use the Pyke keyword `first` here because there can be multiple proofs that `$ID` is indeed an identifier, but we need only one. The keyword `first` is akin to `once/1` as defined in the Prolog standard[15].

The remaining lines consist of two pairs, each implementing one of the explicit preconditions of the rule. In Lines 6 and 8 we select the relevant environment component. In Lines 7 and 9 the actual lookup is done, and the result is compared to the desired value.

4.4 Implementation validation

During the introduction, we stated four challenges in tool development for DSLs, and we proposed an approach for development of type checkers for DSLs. In this subsection we review the challenges and discuss how the challenges were addressed by the approach.

The first challenge required correctness of the tools supporting DSLs to be clear to the domain experts. We addressed this challenge by (1) separating the specification of the type checker (type system) from its implementation and (2) specifying the type checker in a formal language close to the popular SOS. For the case study discussed in Section 4 we closely cooperated with the Chi domain experts having previous experience with specifying formal semantics in SOS.

The second challenge demands the approach to be suited for evolution, and the third one requires support of decision making by language designers. While these two challenges are quite different, our approach allows to address both of them in a uniform way. Recall that the main components of our approach are type system specification in MSOS and automatic generation of the type checker automatically. In our previous paper [11] we have shown that MSOS specifications are well-suited both for specification of type checking evolving languages and for the decision making support. Automatic type checker generation allows to regenerate a type checker when the type system has evolved. Implementation of the transformation required merely 235 ASF+SDF transformation rules, amounting to 1174 lines of code. The MSDF grammar used has 115 production rules, and the Pyke grammar 280, 234 of which are part of an imported Python grammar. Hence there are only 46 Pyke-specific production rules.

Our final challenge required the generated tool to fit the existing tools available for the DSL. We have discussed the ways to satisfy our requirements while considering the choice of the target language in Section 3.1. Specifically, this is the reason why in our case study Pyke was chosen as the target language in Section 4.2.

We have received a very positive feedback from the designers of Chi. The MSOS+generation methodology advocated in this paper has been chosen by them to support type checking of CIF[5], a new generation hybrid system specification language established by the members of the European network of excellence HYCON.

5 Related work and conclusion

Conclusion We have presented a novel approach to the development of formally specified type checkers for domain-specific languages. The approach requires to specify the type system using MSOS and MSDF and to generate a type checker in the desired target language using ASF+SDF. MSOS is formal, well-suited for evolution and close enough to the popular SOS to be understood by the domain experts with limited programming experience. ASF+SDF allows us to generate a type checker and to reuse the generation process when the DSL

evolves. We have illustrated feasibility of the approach by developing a type checker for Chi 2.0, a high level specification language for concurrent systems.

Other approaches The idea of using a generative approach to construct type checkers can be found, e.g., in [24, 2]. Application of this idea to domain-specific languages, however, necessitates addressing the challenges stated in Section 1. These challenges render such approaches as [24, 2] not suited for domain-specific languages as these approaches were not conceived with the issues of language evolution and tool integration in mind. In particular, in [24], languages are divided into features. If two features are not related in a straightforward manner, extra effort is required to combine them. If a new feature is to be added to a languages that already has many features, this is obviously not desirable. TYPOL programs as described in [2] appear to suffer from the same problem, though the original paper does not state this explicitely.

Future work While we have focused on developing type checkers, we believe that our approach is applicable to other forms of static semantic analysis as well. As long as the desired static semantics can be specified using MSOS and MSDF, the implementation can be derived by means of a ASF+SDF generated transformation in a way similar to Section 4.3. For instance, using our approach one should be able to detect potential unsafe operations, e.g., division by zero. Hence, one possible direction for future work consists in applying our approach to different kinds of static semantic analysis.

Another possible direction for future work consists in investigating alternative means of specification the type system. While MSOS has numerous advantages for specification of type systems while compared to such techniques as SOS, denotational semantics and attribute grammars (cf. Section 2.1), it still might be too complex for domain experts. One might consider developing visual specification languages combining advantages of MSOS with intuitiveness of visual representation. Appropriateness of such a language should be investigated by means of comprehension studies.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman, 2006.

[2] I. Attali. Compiling TYPOL with attribute grammars. In P. Deransart, B. Lorho, and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming*, volume 348 of *LNCS*, pages 252–272. Springer, 1988.

[3] J.C.M. Baeten, D.A. van Beek, P.J.L. Cuijpers, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, and R.J.M. Theunissen. Model-based engineering of embedded systems using the hybrid process algebra Chi. 209:21–53, 2008.

[4] E. Balland, P. Brauner, R. Kopetz, P. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[5] D.A. van Beek, P. Collins, D.E. Nadales, J.E. Rooda, and R.R.H. Schiffelers. New concepts in the abstract format of the compositional interchange format. In A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, editors, *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255, Zaragoza, Spain, 2009.

[6] D.A. van Beek, A.T. Hofkamp, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of Chi 2.0. SE Report 2008-01, Eindhoven University of Technology, 2008.

[7] D.A. van Beek, A. van den Ham, and J.E. Rooda. Modelling and control of process industry batch production systems. In *15th Triennial World Congress of the International Federation of Automatic Control*, 2002.

[8] C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`.

[9] M.G.J. van den Brand. Applications of the ASF+SDF Meta-Environment. In Lämmel et al. [23], pages 278–296.

[10] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[11] M.G.J. van den Brand, A.P. van der Meer, and A. Serebrenik. Type Checking Evolving Languages with MSOS. In J. Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 207–226. Springer, 2009.

[12] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Sci. Comput. Program.*, 36(2-3):209–266, 2000.

[13] K.B. Bruce, J. Crabtree, T.P. Murtagh, R. van Gent, A. Dimock, and R. Muller. Safe and decidable type checking in an object-oriented language. *ACM SIGPLAN Notices*, 28(10):29–46, 1993.

[14] A. Cleve, J. Henrard, and J. Hainaut. Co-transformations in Information System Reengineering. *ENTCS*, 137(3):5–15, 2005.

[15] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: the standard: reference manual*. Springer-Verlag, London, UK, 1996.

[16] A. Dijkstra and S.D. Swierstra. Typing Haskell with an Attribute Grammar. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 1–72. Springer, 2004.

[17] T. Ekman and G. Hedin. Modular Name Analysis for Java Using JastAdd. In Lämmel et al. [23], pages 422–436.

[18] B. Frederiksen. Pyke, 2009. http://pyke.sourceforge.net.

[19] R. Grimm, L. Harris, and A. Le. Typical: taking the tedium out of typing. Technical Report TR2007-904, New York University, November 2007.

[20] A.T. Hofkamp and J.E. Rooda. Chi 2.0 language reference manual. SE Report 2008-02, Eindhoven University of Technology, Mechanical Engineering, Eindhoven, The Netherlands, 2008.

[21] E.T. Irons. A syntax directed compiler for ALGOL 60. *Commun. ACM*, 4(1):51–55, 1961.

[22] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[23] R. Lämmel, J. Saraiva, and J. Visser, editors. *Generative and Transformational Techniques in Software Engineering, International Summer School, Revised Papers*, volume 4143 of *LNCS*. Springer, 2006.

[24] M.Y. Levin and B.C. Pierce. TinkerType: A Language for Playing with Formal Systems. Technical report, 2000.

[25] P.D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[26] P.D. Mosses. Foundations of Modular SOS. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, volume 1672 of *LNCS*, pages 70–80. Springer, 1999.

[27] P.D. Mosses. Modular structural operational semantics. *JLAP*, 60–61:195–228, 2004.

[28] M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *LNCS*. Springer, 1999.

[29] B.C. Pierce. *Types and programming languages*. MIT Press, 2002.

[30] H.J.A. Rulkens, E.J.J. van Campen, J. van Herk, and J.E. Rooda. Batch size optimization of a furnace and pre-clean area by using dynamic simulations. In *Advanced Semiconductor Manufacturing Conference and Workshop*, pages 439–444, Boston, MA, 1998. IEEE/SEMI.

[31] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[32] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M Dausmann, G Winterstein, and W. Kirchgässner. *An Attribute Grammar for the Semantic Analysis of Ada*, volume 139 of *LNCS*. Springer, 1982.

[33] N.P. Veerman. Revitalizing Modifiability of Legacy Assets. In *CSMR*, pages 19–29. IEEE Computer Society, 2003.

[34] E. Visser. Program Transformation with Stratego/XT. Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. Technical Report UU-CS-2004-011, Utrecht University, 2004.