Volume 25 Number 4

# ACTA CYBERNETICA



Szeged, 2022

# ACTA CYBERNETICA

**Information for authors.** Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). There are no page charges. An electronic version of the published paper is provided for the authors in PDF format.

**Manuscript Formatting Requirements.** All submissions must include a title page with the following elements: title of the paper; author name(s) and affiliation; name, address and email of the corresponding author; an abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.).

When your paper is accepted for publication, you will be asked to upload the complete electronic version of your manuscript. For technical reasons we can only accept files in LaTeX format. It is advisable to prepare the manuscript following the guidelines described in the author kit available at `https://cyber.bibl.u-szeged.hu/index.php/actcybern/about/submissions` even at an early stage.

**Submission and Review.** Manuscripts must be submitted online using the editorial management system at `https://cyber.bibl.u-szeged.hu/index.php/actcybern/submission/wizard`. Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed.

**Subscription Information.** Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: `acta@inf.u-szeged.hu`

**Web access.** The above information along with the contents of past and current issues are available at the Acta Cybernetica homepage `https://cyber.bibl.u-szeged.hu/` .

**Tibor Gyimóthy**
Department of Software Engineering, University of Szeged, Hungary
gyimothy@inf.u-szeged.hu

**Zoltan Kato**
Department of Image Processing and Computer Graphics, University of Szeged, Hungary
kato@inf.u-szeged.hu

**Dragan Kukolj**
RT-RK Institute of Computer Based Systems, Novi Sad, Serbia
dragan.kukolj@rt-rk.com

**László Lovász**
Department of Computer Science, Eötvös Loránd University, Budapest, Hungary
lovasz@cs.elte.hu

**Kálmán Palágyi**
Department of Image Processing and Computer Graphics, University of Szeged, Hungary
palagyi@inf.u-szeged.hu

**Dana Petcu**
Department of Computer Science, West University of Timisoara, Romania
petcu@info.uvt.ro

**Andreas Rauh**
School II – Department of Computing Science, Group Distributed Control in Interconnected Systems, Carl von Ossietzky Universität Oldenburg, Germany
andreas.rauh@uni-oldenburg.de

**Heiko Vogler**
Department of Computer Science, Dresden University of Technology, Germany
Heiko.Vogler@tu-dresden.de

**Gerhard J. Woeginger**
Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

# Special Issue of the Conference on Software Technology and Cyber Security

*Guest Editors*

**Tamás Kozsik**

Department of Programming Languages and Compilers,
Faculty of Informatics, Eötvös Loránd University,
Budapest, Hungary
kto@elte.hu

**Simon Thompson**

School of Computing, University of Kent,
Canterbury, Kent, United Kingdom
s.j.thompson@kent.ac.uk

Department of Programming Languages and Compilers,
Faculty of Informatics, Eötvös Loránd University,
Budapest, Hungary
thompson@inf.elte.hu

Acta Cybernetica **25** (2022) 753–779.

# Towards a Generic Framework for Trustworthy Program Refactoring*

Dániel Horpácsi[abc], Judit Kőszegi[bd], and Dávid J. Németh[be]

**Abstract**

Refactoring has to preserve the dynamics of the transformed program with respect to a particular definition of semantics and behavioural equivalence. In general, it is rather challenging to relate executable refactoring implementations with the formal semantics of the transformed language. However, in order to make refactoring tools trustworthy, we may need to provide formal guarantees on correctness. In this paper, we propose high-level abstractions for refactoring definition and we outline a generic framework which is capable of verifying and executing refactoring specifications. By separating the various concerns in the transformation process, our approach enables modular and language-parametric implementation.

**Keywords:** refactoring, domain-specific language, refactoring methodology, formal verification

## 1 Introduction

The idea of refactoring is as old as high-level programming. A program refactoring [7] is typically meant to improve non-functional properties, such as the internal structure or the appearance, of a program without changing its observable behaviour. Tool support is necessary for refactoring in large-scale: it has to be ensured that program changes are complete and sound, the behaviour is intact and no bugs are introduced or eliminated by the transformations. Refactoring tools may

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]Project no. ED_18-1-2019-0030 (Application domain specific highly reliable IT solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

[b]ELTE Eötvös Loránd University, Budapest, Hungary, and Faculty of Informatics, 3in Research Group, Martonvásár, Hungary

[c]E-mail: daniel-h@elte.hu, ORCID: 0000-0003-0261-0091

[d]E-mail: koszegijudit@elte.hu, ORCID: 0000-0003-1915-4176

[e]E-mail: ndj@inf.elte.hu, ORCID: 0000-0002-1503-812X

DOI: 10.14232/actacyb.284349

carry out extensive modifications in large projects, which are hard to be performed or comprehended by humans.

Generally speaking, the primary goal of a refactoring framework is to provide automatic tool support for behaviour-preserving (semantics-preserving) program rewriting. Additional general design goals may include support for interactive execution, multiple target languages or extensibility via user-defined transformations. Trustworthiness of refactoring implementations is usually backed by excessive amounts of testing, but research keeps looking for possibilities of formally specifying and verifying the correctness of program transformations. Whether a refactoring tool gets widely adopted highly depends on the extensibility and the trustworthiness of the solution.

Refactoring, from the programmer point of view, is an editor or dedicated tool function that helps modify the program in a well-understood way, increasing code quality. From the tool designer point of view, it is a complex process of creating and analysing a program model, locating elements of interest, and rewriting of the model to an equivalent model. In the typical case, these aspects of the process are mixed up, resulting in a hardly extensible, language-specific tool, which is unreasonably difficult to formally verify.

Our aim with this paper is to outline a generic design for a refactoring framework that has all the above-mentioned features: it uses executable and extensible definitions, supports multiple languages and enables semi-automatic formal verification. In particular, we present abstractions for program representation and refactoring definition, and we describe a language-parametric architecture that can be tailored to programming languages of different paradigms by supplying the formal definition of the language along with some refactoring schemes. The main contributions of this paper are:

- Design of a generic refactoring approach that supports executable and semi-automatically verifiable transformations via language-specific semantic predicates and refactoring schemes;
- Description of a language-parametric framework with language-specific artefacts and language-independent components, with a guideline on how the framework is tailored for a particular language;
- Testimonials of applying the above-mentioned framework to languages of two different programming paradigms.

We structure the rest of the paper as follows. First, we survey related work in Section 2, focusing on language-agnostic approaches and proven-correct refactoring. Then, we present high-level abstractions for defining transformations, which enable language-parametric implementation and formal verification (Section 3); this section is partly based on our previous work [11]. It is followed by the demonstration of the generic refactoring framework and its components in Section 4, and then we outline the main concerns with instantiating the framework for functional and object-oriented languages (Section 5). Finally, we conclude the paper after a brief discussion of our results.

## 2 Related work

There are various approaches to specifying and implementing refactoring, which vary in terms of the model they use for representing programs, in the abstraction level they use for specifying program semantics and program transformations, as well as in the level of guarantees they can provide on correctness. In this section, we overview the most important and influencing related work.

**Preliminaries.** In this paper, we use the terms *object language* or *target language* when referring to the programming language we aim to refactor. By *static semantics*, we mean the context-dependent part of the syntax (e.g. name binding and type information), as well as additional (possibly dynamic) semantic properties that can be statically extracted from, or approximated based on, the source code (such as purity, control flow and data flow). *Dynamic semantics* is the formal definition of program run-time behaviour, presumably given in small-step operational style or in reachability logic [24]. By *refactoring correctness* we mean that the transformation turns any program into a semantically equivalent program. This correctness property could be checked after each application of the refactoring (i.e. whether a particular execution of a transformation was correct), but in our research we aim at verifying the definition itself, i.e. formally reasoning about the transformation being correct applied to any program.

**Compositional definition of refactoring.** Although the abstractions for defining refactoring are varying from approach to approach, almost all solutions incorporate the fundamental work of Opdyke [21] that suggests refactorings be composed of basic steps called micro-refactorings. Simpler transformations are easier to read, write and to verify; on the other hand, decomposition of extensive refactorings to simple steps may require experience and considerable effort. Having said that, the compositional approach enables modularity both in definition, execution and in verification, and is therefore inevitable in designs where generality and formal verification are among the design goals.

**Refactoring framework.** Roberts [22] designed one of the first dedicated frameworks for implementing refactoring transformations. He states that refactoring tools should a) be completely automated; b) be provably correct and c) offer complex refactorings composed from primitive ones. We strongly share these fundamental design goals in our own approach, and in addition, we believe that even the primitive refactorings should be user-definable.

Widely used general-purpose programming languages have all gained their own language processor environments which support analysis and transformations on a model of the program, even functional programming languages, such as Haskell or Erlang. In fact, our approach born as a generalization of a refactoring framework [2] and its API designed for the Erlang programming language. The Erlang-specific solution was summarized along with a case study in [11].

**Language-agnostic approaches.** Language-independent specification of refactorings is an idea that pops up regularly, addressing the problem of semantics-preserving program transformation with generic program representations, analysis and traversal functionality. Lämmel [14] proposes a generic refactoring system based on Strafunski-style generic functional programming. It states that a refactoring can be described by a number of steps of the following kind: a) identification of fragments of a certain type and location; b) destruction, analysis, and construction; c) checking for pre- and postconditions and d) placing, removing or replacing a focus. Just like with work by Roberts, we strongly agree with Lämmel's thoughts about separation of concerns. In particular, we suggest that refactoring should be phrased as a composition of analysis and transformation, where transformation consists of pre-condition checks and actual rephrasing of the program model by using language-agnostic strategies.

Another branch of language-independent transformation specification is based on an XML-based program representation. RefaX [20] brings the premise of a fully language- and model-independent refactoring tool by using XML, while Jrbx [19] generalizes this idea by adding fairly generic static semantic analysis. We share the aim of these approaches, but we add verifiability of transformations at reasonable cost as an additional primary requirement.

**Specific languages for refactoring.** Designing domain specific languages for refactoring programming is also an established idea, there are related results for different object languages with different representations. Some of these define the entire code transformation logic including term-level rewriting, while some only offer a formalism for composing atomic steps in a convenient way.

In the former case, when the refactoring operates on the level of the program model, the actual program representation highly determines the abstraction level of the patterns and the transformation primitives. Gómez et al. [10] introduce a generic model for representing programs and their history, pushing the boundaries of regular program representations in order to support a wide variety of language processing methods.

Rewrite-based transformation languages use different kinds of patterns to match and construct program models. Leitão [16] gives an executable, rewrite-based refactoring language with expressive patterns, Verbaere et al. [28] propose a compact, representation-level formalism for executable definitions. These formalisms are expressive and language-independent, but at this level of generality, correctness checks for refactoring definitions would become practically unfeasible. Rich and high-level program patterns are presented by Gil et al. [9] in their language for defining programming idioms in Java.

Languages for composing already existing transformations exist as well: Kniesel and Koch [12] put the emphasis on ensuring correct composition of transformations, and for Erlang, Li and Thompson [17] define an API for describing prime refactorings and a feature-rich language for interaction-aware composition.

**Proven-correct refactoring.** For the object-oriented paradigm, Schaefer and de Moor introduce a system [26] in which they reason about semi-formal definitions of a set of basic refactorings. This is a very influencing piece of work, but they focus on preserving static semantic properties, not dynamics. Roberts [23] applies a different definition style, with an emphasis on the side-conditions and proper composition of the base refactorings. Neither of them provides formally verified and executable definitions.

There are some results [27] in defining provably correct refactorings for simple languages, and also some mechanised proofs exist even for modern laguages and real-world use cases [6, 25]; on the other hand, these are specific to one particular transformation, and do not allow for defining custom transformations or provide verification for those. [8] presents a preliminary work on defining verifiable and executable refactoring in Maude, with a similar approach to ours as to rewriting-based definitions, but their definitions are very low-level and hardly readable, out of reach for the average programmer to specify their own transformations.

# 3 Refactoring definition

Before describing the refactoring framework itself, we elaborate on the abstractions to be used for defining refactoring program transformations. We start by separating concerns. Analysis, condition checking and transformation are seemingly interdependent phases of the refactoring process, but by careful separation we can make the definition less error-prone [18] and enable a more modular implementation. Then, we introduce the refactoring definition abstractions that allow us to define transformations in a high-level and compositional way and enable semi-automatic formal verification for consistency. We end this section by exploring the consequences of defining the transformation with the proposed abstractions, and analyse how it loosens and simplifies the dependencies among the various components of the refactoring framework. This refined dependency structure lets the language-independent components be parametrised with language-specific artefacts and will lead us to a language-parametric framework.

## 3.1 The refactoring business logic

Refactoring implementations define their input and output as program text in concrete language syntax; however, under the hood, the text is usually turned into an intermediate representation (model) which is further analysed and transformed, and finally, the model is printed back to text. If we properly separate these phases, we can define the refactoring as a composition of *analysis*, *transformation* and *synthesis*.

$$Text \xrightarrow{analysis} Model \xrightarrow{transformation} Model \xrightarrow{synthesis} Text$$

The model we talk about here is a high-level program representation, such as an abstract syntax tree (AST), a higher-order abstract syntax tree or an abstract semantic graph (ASG). In this model, we do not suppose the program logic or high-level architecture to be present (like a UML model) — it is more like a graph that captures the grammatical structure and possibly the static semantic properties of the transformed program. The level of detail in this graph model may vary, we address this question in the next section.

Let us define what we mean by the phases of refactoring:

- *Analysis* is the process of extracting the information from the textual format that is necessary for checking the refactoring side-conditions and locating program elements to be changed by the transformation.

  Analysis can be further divided into two steps: syntactic analysis (parsing) and semantic analysis. Parsing yields a structure tree for the text, then static semantic analysis computes an approximation of fundamental semantic properties of program entities, such as binding relations, data-flow and control-flow.

- The *model transformation is the actual business logic of the refactoring.* It takes place in the middle of the process, given as a (deterministic) relation that maps program models to equivalent program models.

  $$\mathit{Transformation} : \mathit{Model} \to \mathit{Model}$$

  Typically, this function is defined as the semantics of an algorithm written in a programming language or a description in a domain specific language, which does traversals on the model to gather semantic information and carry out rewriting.

- *Synthesis* turns the model back to textual format and obtains the result of the refactoring. We suppose that the model contains the original layout and names of the program and it can be pretty-printed to concrete syntax, but we note that for some higher-level models it may be necessary for the synthesis to incorporate the original text as well.

The strict separation of analysis, model transformation and synthesis simplifies the definition and verification of refactoring transformations, yet the composition of these steps can precisely define the original text transformation. Refactorings may have context-sensitive side-conditions requiring thorough inspection of static semantic properties and therefore complex semantic analysis, but in the rest of the paper, we focus on the model transformation phase.

In particular, from now on by *refactoring* we mean the model transformation and not the text transformation. In the refactoring definition we omit the collection or extraction of static semantic information, and the formal verification of our refactoring definition only proves the model transformation correct, not the text transformation – analysis and synthesis are trusted components in the system.

**Level of model abstraction**

The complexity of the refactoring definition highly depends on the program model, and the abstraction level of the model affects the complexity of the analysis as well. As it will be demonstrated, the boundary between analysis and transformation is movable by adjusting the level of detail in the model. In this sense, analysis is not the process of building the program model but the extraction of static semantics, which may happen alongside the transformation. Apparently, the more detailed the model is, the less analysis-related traversal takes place during transformation.

**Simple model.** If the model does not contain details on the static semantics of the program, the transformation gets more complex as it has to carry out analysis tasks. In a corner case, the analysis only does parsing; thus, the program model is a syntax tree and the transformation has to conduct semantic analysis (syntax tree traversals) for checking the side-conditions of the refactoring (see Figure 1). In this case, transformation definitions are overly complex and they are out of reach when it comes to formal verification of semantics-preservation, or even to check termination properties of analysis and transformation. For instance, in the optimisation steps discussed in [3], traversal strategies carry out behaviour-preserving transformations by linking analysis and term rewriting, and the drawbacks of this approach are discussed in [18].

$$Text \xrightarrow{parsing} AST \xrightarrow{analysis+transform.} \ldots \xrightarrow{analysis+transform.} AST \xrightarrow{deparsing} Text$$

Figure 1: Refactoring with tree rewriting

**Detailed model.** A complex enough static analysis can build a model detailed enough that enables the transformation to check the side conditions by simple queries in the model. This is a trade-off: a more complex static semantic analysis may be harder to reason about, but cuts the complexity from the transformation, making both of them tractable. In the corner case, the model can be so detailed so that predicates in side-conditions are one-to-one mappings to model elements and no analysis-related operations take place during transformation (see Figure 2).

$$Text \xrightarrow{parsing} AST \xrightarrow{analysis} ASG \xrightarrow{transform.} \ldots \xrightarrow{transform.} ASG \xrightarrow{synthesis} Text$$

Figure 2: Refactoring with graph rewriting

Since our main goal with this refactoring framework is to make definitions generic and trustworthy, we aim at using a high-level and detailed program model. This model extends the syntax tree with information on binding, types, data-flow and

control-flow, and even on purity and non-functional properties, so that the refactoring definition side-conditions can be expressed in terms of concepts of the programming language we refactor, and more importantly, analysis and transformation concerns are fully separated. In the case of the Erlang prototype implementation of the proposed framework (see Section 5.2), we rely on the semantic program model introduced in [2].

**Tree rewriting in the graph.** According to Figure 2, the refactoring transformation is a mapping from semantic graphs to semantic graphs, which suggests that it is easiest defined with a graph rewrite system. Nonetheless, the figure also suggests that any AST can be mapped to the corresponding ASG with static semantic analysis. In practice, the ASG is a proper extension of the AST, containing additional edges and nodes that represent static semantic information.

In our approach we divide the model into its syntactic and semantic parts and work with the model like this: we carry out a transformation on the syntax tree whilst using the semantic layer for checking transformation validity. The result of the transformation is semantic graph containing no semantic elements, so it is reanalysed to obtain the semantic graph prior to further transformation. Since both the AST and the ASG are understood as properly formed models, we can refine the previous signature for transformations discussed in the beginning of this section:

$$Transformation : ASG \to AST$$

Note that although this approach alternates analysis and transformation (see Figure 3), it keeps these phases completely separated (unlike in Figure 1), so still realise separation of concerns . Implementing the transformation as a function from graphs to trees provides a good strategy towards defining trustworthy refactoring in terms of semantics-constrained term rewrite rules for which we introduce abstractions in the following section.

$$\ldots \quad AST \xrightarrow{analysis} ASG \xrightarrow{transformation} AST \xrightarrow{analysis} ASG \xrightarrow{transformation} \quad \ldots$$

Figure 3: Refactoring with semantics-constrained tree rewriting

**Summary of assumptions on the model.** In the rest of the paper, we assume that the *refactoring is defined on a high-level model* that captures program syntax and static semantics. The *transformation on this model maps the tree along with the static semantic properties into a transformed tree.* We will also assume that the *model's semantic layer captures all program properties that may be needed to tell side-conditions of refactorings* (e.g. name references, data-flow relations or purity of expressions). These target language-level concepts are defined by a set of so-called semantic predicates, which will be used in the side-conditions of refactorings.

## 3.2 Abstractions for defining transformations

This section surveys how refactorings in the proposed framework are defined such that they provide *trustworthiness* and enable *genericity*. These goals are mainly achieved by keeping the refactoring definition high-level (independent of the representation and the target language), declarative (expressing what to do rather than how to do) and compositional (small definitions combined into bigger ones). We review the refactoring definition abstractions proposed in our previous work [11], and at the same time, we investigate how these abstractions allows for a generic and modular implementation of interpretation and verification.

First of all, higher abstraction level in the definition means less details mentioned explicitly, which reduces the complexity of the definition and the potential for making mistakes. This may come with a performance penalty, but assuming that trustworthiness is more important than efficiency, it is reasonable to opt for higher-level abstractions (as opposed to low-level transformation primitives). For instance, in the context of refactoring, using term rewriting is clearly safer than direct manipulation of the syntax tree as it excludes the risk of constructing structurally invalid subtrees and therefore creating an invalid model.

Careful selection of the transformation abstractions can also make the definitions more amenable to formal verification, further increasing trustworthiness: for instance, refactoring schemes allow us to argue about the correctness of the program transformations in terms of verifying a set of program patterns for semantics equivalence instead of proving imperative term rewrite algorithms correct. This is similar to composing imperative programs with algorithmic skeletons that correctly implement compound control patterns and enable programmers to write complex programs without mentioning the low-level details. Again, this comes with a reasonable penalty: not all program transformations will be expressible with this set of abstractions, but the goal is to be able to define meaningful refactorings in a way that allows for semi-automatic formal verification.

Last but not least, the high-level definitions can be given in a language that does not depend on the representation of programs nor on the particularities of the target programming language, enabling a fairly generic implementation parametrised with language-specific components. The resulting modular framework showcases reusable, language-independent components, as well as it provides trustworthiness by reducing the complexity of individual components (see details in Section 4).

### Strategic term rewriting with semantic predicates

The transformation function over models could be defined imperatively, but we aim at defining it as declaratively as possible — as mentioned above, declarative programs contain less details as to how the execution takes place and thus they tend to be more reliable. As one of the building blocks, we employ conditional term rewrite rules to define local tree transformations. This formalism abstracts over the imperative steps of term traversal, pattern matching and replacement construction, serving as a declarative description of simple rewrite steps.

Conditional term rewrite rules consist of two patterns and a condition expression:

$$\frac{\textit{matching pattern}}{\textit{replacement pattern}}\ \textit{conditions}$$

In such a rule, the matching and replacement patterns are first-order terms: they can contain metavariables to extract subterms and use those to construct new terms. In the typical formalisation, the condition is a statement on the rewrite relation itself, potentially referring to the metavariables bound via pattern matching. The set of rules can be interpreted as a normalising term rewrite system by assuming exhaustive application of rules.

*Strategic term rewriting* improves on ordinary systems by introducing explicit control and context for rewrite rule applications, which is more suitable for defining refactoring transformations. We will use a generalised variant of strategic term rewriting to define transformations over the syntactic part of the semantic program model. In order to accommodate the principle of separation of concerns explained in the previous section, we generalise strategic term rewriting in several aspects: we define conditions in terms of a logic formula using language-specific *semantic predicates (metatheory)*, as well as we introduce strategies that use static semantic properties to control the term rewrite rule application. This latter idea of semantics-driven strategies, so-called *refactoring schemes*, provides fully declarative definition for extensive program transformations as it hides the rule application control under generic control schemes.

**Semantic conditions.** As mentioned already, [18] gives in-depth explanation of how difficult it may be to reason about side-conditions expressed in terms of reachability statements. To overcome this issue, unlike traditional term rewriting, we do not refer to the rewrite relation in the condition; instead, the conditions are logic formulae over a predicate set characterising the abstractions of the object language. With this design decision, we fully detach analysis and transformation in the refactoring definition, which will allow for a generic implementation in the framework.

Semantic predicates in our approach have two interpretations: they can be evaluated over a particular program model, or can be mapped to a set of axioms in the dynamic semantics of the target language. This latter is of great importance from the verification point of view. For instance, the predicate *pure* can be evaluated by checking the expression for any side-effects, while the axiomatic specification tells that such an expression can be moved in the control chain while preserving control and data flow (for example usage, see Listing 3).

**Refactoring schemes.** In general, single conditional rewrite rules can only define local changes, so-called *local refactorings*. On the other hand, many refactorings span over entire projects and are inherently *extensive*: they affect many locations in the program, which have to be modified consistently. Strategic term rewriting offers simple operations [29] for combining simple (or local) transformations

with e.g. sequential composition, branching or fixed-point operation, but from the trustworthiness point of view, these combinators are too permissive and tedious to formally tackle. Refactoring is a very special case of program transformation, which gives rise to the idea of strategies specific to program refactoring. We call these *refactoring schemes*.

Schemes are special strategies that combine conditional rewrite rules, and are defined using ordinary control strategies as well as target modifying strategies combined with semantic predicates. They provide a high-level notation for extensive changes, hiding the control primitives and providing a surface language for defining consistent program transformations in a declarative way. Again, schemes pose a restriction on the sorts of expressible transformations, but dividing the definition to multiple levels will allow us to implement the execution modularly and carry out semi-automatic formal verification.

**Consistency.** The key concept behind schemes is dependency: extensive transformations have to follow dependency chains in the program, visit and change those program elements consistently that are interdependent. Schemes can be driven by dependencies induced by data flow or name binding. Correctness proof for a scheme is as hard as proving an imperative strategy correct; however, the method divides the proof in half: verification of the scheme and the verification of the instantiation. In our design, the second half can be carried out semi-automatically as it boils down to machine-checkable expression pattern equivalences.

Dependencies vary from language to language; hence, our method supposes that the set of pre-verified refactoring schemes are defined for each object language the framework is instantiated for, based on the static semantics of the language. Consequently, although the idea of schemes is language-independent, the concrete schemes we define the transformations with are specific to the object language. In the following section, schemes will be identified as artefacts attached to the object language definition.

### Refactoring compositionality

The abstractions for defining local and scheme-based extensive refactorings are supposed to be micro-refactorings: they carry out the least possible amount of transformation steps which form a consistent change in the program. The smaller the steps, the more trustworthy and more easily verifiable they are. Once such a micro-step is proven to be semantics-preserving, it can be combined with other refactoring steps, and the result will be another, compound refactoring.

Our specification language facilitates a sub-language specific to combining refactoring transformations. The steps can be combined by means of basic imperative control: sequencing, branching and (bounded) iteration. These combinators are compositional: if the steps they combine are behaviour-preserving, the resulting transformation will be behaviour-preserving as well. In the end, complex refactorings are defined by decomposition to smaller refactoring steps that are defined as instances of refactoring schemes.

### Examples of refactoring definitions

To facilitate the refactoring definition abstractions introduced in this section, we propose a domain-specific language (DSL) [13] for refactoring. Definitions in the refactoring specification language are both executable (can be mapped to a computable model transformation function) and are semi-automatically verifiable (behaviour preservation can be formally checked by verification of automatically synthesized formulae).

Syntax rewrite rules are written in the inference rule notation, patterns are expressed in the concrete syntax of the object language. In the patterns, metavariables are distinguished from ordinary variables by using a kind of quotation syntax (e.g. `#varname`) or extra predicates (e.g. `is_var(VarName)`). In the Erlang prototype, the normal variable syntax is used for metavariables and target language variables are matched with conditions. Metavariables followed by double-dot match consecutive syntactic elements. Schemes are instantiated with rewrite rules, and refactorings are combined in simple scripts. We showcase some examples borrowed from [11], defined for Erlang [4] as the object language.

**Local refactoring.** Simple, local changes are expressed with conditional term rewrite rules, where conditions are first-order logic formulae constructed with semantic predicates defined by the language metatheory. Listing 1 defines a transformation that encloses an expression into a lambda-abstraction, supposing that the expression does not bind any variables that are referred to by its context (predicate `non_bind`). This definition also demonstrates the usage of matching conditions [29]: the list of free variables (`free_vars`) is bound to a metavariable (`Vars..`) and is being used in the replacement pattern.

```
local refactoring wrap()
    E
    -------------------------------
    (fun(Vars..) -> E end) (Vars..)
when
    Vars.. = free_vars(E) and non_bind(E)
```

Listing 1: Wrap expression refactoring

**Extensive refactoring.** Extensive changes are defined as reductions to refactoring schemes. For instance, a scheme for Erlang is *function refactoring*, which takes two rewrite rules and visits the definition and the references of the function. References may include intra-module and inter-module function applications, both first-order and higher-order. The patterns given in the parameter rewrite rules define the way the dependent program parts are changed. For the scheme instance to be correct, the two parameter rewrite rules have to be consistent.

The function refactoring scheme can be used for implementing various refactoring steps: renaming a function (Listing 2), reordering and grouping its arguments.

```
function refactoring rename_function(NewName)
definition
    Name(Args..) -> Body..
    -------------------------
    NewName(Args..) -> Body..
reference
    Name(Args2..)
    ---------------
    NewName(Args2..)
```

Listing 2: Rename function refactoring

Interestingly enough, the very same scheme can be used to move a binding from the function body to its signature, introducing a new parameter to the function. In this latter case (see Listing 3), the scheme instantiation contains an extra condition expressing that the expression moved from the body to the call site is pure (does not cause any side effects) and closed (does not contain any free variables).

```
function refactoring var_to_param(X)
definition
    Name(Args..) -> X = E, Body..
    ----------------------------
    Name(Args.., X) -> Body..
reference
    Name(Args2..)
    ----------------
    Name(Args2.., E)
when pure(E) and closed(E)
```

Listing 3: Top-level local variable to function parameter refactoring

**Composite refactoring.** Let us consider a simple refactoring that lifts a local variable from the function body to the function scope, as a new parameter. This transformation can be decomposed to 1) iteratively lifting the variable to outer scopes (`outer_variable`) and 2) adding it as a parameter (`var_to_param`) once it is a top-level variable. This composition of refactorings can be expressed by iteration and sequential composition, as scripted on Listing 4 (the pseudovariable `This` refers to the object language variable that was selected as the target of the transformation). Note that in these composite refactorings, transformations are applied to program elements determined by so-called selector functions; in this example, `function` selects the enclosing function of the variable originally chosen as refactoring target.

```
refactoring to_function_parameter()
do
  iterate This.outer_variable()
  function(This).var_to_param(This)
```

Listing 4: Local variable to function parameter refactoring

**Dependencies untangled**

A refactoring framework is said to be language-generic if it is either language-parametric or easily adapted to different programming languages. This aim is highly supported by the abstractions with which we express the implemented refactorings. In this section, we have explained the assumptions we make on the refactoring definition and the abstractions we use for specifying transformations. In particular, term rewriting lets us abstract over tree manipulation, semantic predicates and conditions let us separate analysis from transformation, whilst refactoring schemes serve as another parameter to a generic yet trustworthy implementation.

The refactoring specification formalism, apart from the syntax of the patterns in the rewrite rules, is independent of the object language (we note that although predicates are language-dependent, the condition language over predicate symbols is language-independent). We achieve this language-independence by untangling the dependencies among parsing, semantic analysis, condition checking, transformation and synthesis, and by incorporating the idea of algorithmic skeletons into refactoring. With this, we can decouple the language-independent parts from the language-specific elements, and we can define the latter as plug-in components. In particular, the object language is injected into the framework in terms of definitions for syntax (context-free grammar with metavariable format), static semantics (axiomatic semantic predicates), dynamic semantics (small-step rules) and schemes (semantics-directed strategies).

# 4 Refactoring framework

The previous section described the abstractions we can use for specifying refactoring transformations in a rather generic way. The proposed specification language is independent of the object language[1], and it can be interpreted in a generic framework parametrized by the definition of the object language. In this section, first we overview the artefacts that parametrise the framework for a particular programming language, then we introduce the components that implement execution and verification of refactoring definitions.

## 4.1 The object language definition

The following artefacts define the programming language whose sentences are to be refactored. Essentially, they provide a formal definition of the language in question, as well as they define the refactoring idioms that transform program entities according to their control and data dependencies in the object language.

[1] Although if we use concrete syntax in term rewrite rules, the syntax definition of the object language is needed for parsing first-order terms.

**Context-free syntax**

Parsing and deparsing of both input programs and program patterns in rewrite rules can be driven by a single *context-free grammar* definition, with pattern parsing also guided by the definition of metavariable format [15]. The definition of the object language syntax can be given in the usual BNF-like notation, and bottom-up parsers can be generated from it. In addition, it has to contain the abstract syntax description since the internal representation of both programs and program patterns is based on the AST.

**Static semantics (metatheory)**

The metatheory is defined in terms of a *set of decidable semantic predicates* supplied *with two different interpretations* (or semantics if you like):

- Evaluation of the predicate on a given model, yielding true or false. This part of the definition is used in the execution components of the framework, in particular the condition evaluation directly refers to the predicate evaluation defined in the metatheory.

- Axiomatic specification in terms of semantic rules which can be used when arguing about semantic equivalence. This part of the definition is used by the verification component in the framework: when proving conditional pattern equivalence, the semantic conditions are mapped to a set of hypotheses in terms of small-step semantic rules.

It is apparent that these two interpretations of the predicates need to be consistent: every time a predicate evaluates to true, the hypotheses on the dynamic semantic have to be valid. This can be checked with respect to the dynamic semantics definition of the language, but the details of this problem are out of the scope of this paper.

**Dynamic semantics**

The semantics artefact for the object language contains two definitions: the *specification of program behaviour and the characterisation of semantic equivalence* between programs, program fragments or program configurations.

The framework is designed to facilitate a small-step operational-style definition of semantics. During verification, the semantics rules can be used for symbolic rewriting of program patterns checked for semantic equivalence [5]. Recall that neither the semantics nor the behavioural equivalence is incorporated in the refactoring side-condition specification; thus, the execution part of the framework is independent of the dynamic semantics of the object language.

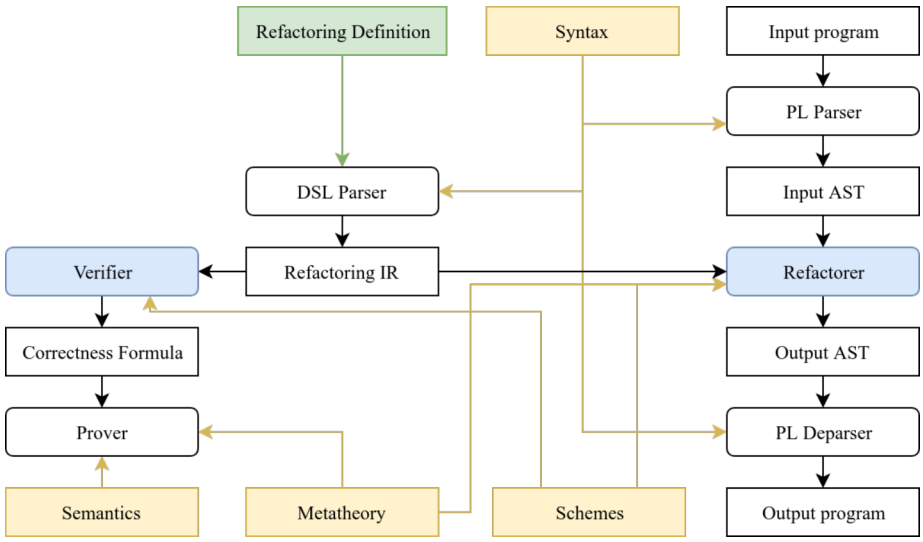



Figure 4: Components and artefacts in the refactoring framework

**Refactoring schemes**

Schemes are language-specific *refactoring idioms (or transformation templates)*, which are parametrised by conditional term rewrite rules. Like for semantic predicates, for schemes we need to provide two interpretations (or semantics) in their definition:

- From the operational point of view, schemes are transformation skeletons, which can be expanded to strategic term rewritings. Thus, it has to be defined how the declaratively specified scheme is expressed in an imperative strategy that applies rewrite rules at appropriate locations in the program model.
- From the verification point of view, schemes need to be mapped to logic formulae that express the correctness property of the extensive refactoring described with them. Our proof-of-concept implementation translates scheme instances to a set of conditional equivalence formulas over the dynamic semantics of the language.

## 4.2 Framework components

As discussed above, the framework (see Figure 4) is parametrised by the definition of the object language given with carefully designed abstractions. Once the framework is tailored to a language, language-specific transformations can be checked for correctness or can be run. In particular, the *refactoring definition* (given as

a specification in the formalism discussed in Section 3.2) determines a semantics-constrained term rewriting relation, which can either be checked for correctness, or it can be executed on a particular input program by the framework implementation.

The framework accommodates a frontend and two backends for the two purposes. The frontend, using a lightweight analysis of the refactoring definition, creates the intermediate representation (IR) of the refactoring specification, whilst the two backends implement the two different sorts of semantics for the refactoring specification. In principle, the two backends can be used independently: one can verify refactoring definitions without execution, and the other way around, execute a definition without verification. Nevertheless, this latter brings the risk of altering the program behaviour during transformation, but it is not prohibited for the sake of situations where the formal verification of the refactoring is practically not feasible in a certain time limit, but the refactoring has to be executed anyway.

### Frontend

The refactoring definition enters the framework via the frontend component. It parses and analyses the definition, and yields an intermediate representation for the transformation specification which can be fed into either the execution or the verification backend. The refactoring specification language, originated from the Erlang prototype, features no static or strong typing; thus, the frontend only does simple sanity checks on the definitions before passing them to one of the backends.

Since the rewrite rules are composed of first-order terms (syntax patterns) written in the concrete syntax of the object language, parsing of the refactoring definition requires parsing of object language syntax patterns. For this, we use the context-free syntax definition of the object language, generalize it to allow metavariables in place of subexpressions and parse the patterns with it into an AST with metavariables. As a result, we obtain a refactoring definition IR in which we embed object language ASTs.

### Execution

One way the refactoring IR can be interpreted is application on a given input program. This is implemented by the compound execution backend, which utilises all object language artefacts except dynamic semantics, and consists of the following components (with the last four grouped into 'Refactorer' in Figure 4):

- **Parser / Deparser** *(uses: Syntax)*
  The input to the refactoring interpreter is program source code, which has to be parsed into a syntax tree before transformation, and needs to be turned back into text following the model transformation. Thus, the context-free syntax definition of the object language is fed into the parser/deparser components which implement the text-to-AST and the AST-to-text conversions, respectively.

- **Scheme expander** *(uses: Schemes)*
  As it was discussed in Section 4.1, schemes have two interpretations, one of which is a translation to lower-level strategies. In this sense, the scheme is a program template for refactoring with holes to be filled with rewrite rules. This component instantiates it with the supplied rewrite rule arguments and yields an imperative term rewrite program.

- **Strategy interpreter**
  This component implements the basic strategies such as composition, left-choice, all-top-down and congruence. Furthermore, we support a number of strategies that rely on the applied program model (abstract semantic graph with references, see [2]), such as applying a rewrite rule on a node, or all of its subtrees, by reference. This component also implements metavariable environments: the metavariables bound with pattern matching are shared with subsequent rewrite rules, thus providing per scheme instance namespaces of metavariables.

- **Condition evaluator** *(uses: Metatheory)*
  The semantic conditions of term rewrite rules are given in terms of object language level predicates combined with simple first-order logic operators. This condition language is interpreted by the condition evaluator component, which relies on the evaluation of predicates over the semantic model of the program. The metavariables used in these formulae are stored in an environment, which is populated by pattern matching executed by the rewrite engine.

- **Term rewrite engine**
  The term rewrite engine carries out the program model transformation (in fact, syntax term transformation) based on the matching and replacement patterns present in the rewrite rules. The current prototypes support expressive syntactic patterns, such as metavariables for matching multiple consecutive nodes in the AST and non-linear patterns, but semantic patterns are not available yet. In addition, in some cases we make use of simple abstract syntax patterns that allow for matching seemingly different concrete syntactic terms.

  The rewrite rules are interpreted in the usual match-and-build semantics: the pattern is matched against the target AST node, creates a substitution (binds the metavariables), builds the replacement subtree, and finally the original node is replaced by the newly created one. Between matching and replacement, the condition evaluator component checks the side-conditions, and the semantics of term rewriting is failure-aware: unsuccessful matching or falsified conditions result in failure of the rewrite rule, which indicates that the rewrite rule was not applicable. Failure is propagated in rule combinators.

**Verification**

Complementing the execution backend, the verifier implements the other semantics to refactoring definitions: statically check their correctness. Ideally, this happens prior-execution, but as discussed before, the framework does not enforce correctness within the execution backend. The verification backend takes the refactoring specification IR, turns it into correctness formulas and verifies their validity. It is implemented in terms of the following two main components:

- **Verifier** *(uses: Schemes)*
  Correctness of refactoring definitions (the property of semantics-preservation) is defined by the validity of a set of logic formulae expressing conditional semantic equivalence of program patterns. The verifier component is responsible for associating the refactoring definition with this set of formulae.

  In our system, refactoring steps are all phrased as a composition of instances of parametrically verified transformation schemes, and these pre-verified schemes determine how the correctness formula is synthesized for the transformation. For a transformation to be correct, its scheme has to be correct as well as the instantiation has to be correct. The formulae that this component synthesises express the correctness of the instantiation. The latter formulae in many cases can be automatically proven with respect to the definition of dynamic semantics and semantic equivalence [11].

- **Prover** *(uses: Semantics, Metatheory)*

  This component checks the validity of the formulae synthesized by the verifier. The prover builds on the metatheory definition by utilizing the axiomatic definition of the semantic predicates, the pattern equivalence is proven upon the definition of dynamic semantics and the definition of semantic equivalence. Once the prover has validated the formulae produced by the verifier component, the transformation is guaranteed to preserve the semantics of any program when applied to by the execution backend.

# 5 Proof of concept

In order to demonstrate the applicability of this generic framework architecture, we investigated instantiating it for two highly different languages: Erlang and Java. This means that we prepared the artefacts detailed in the previous section, i.e. with the appropriate formalism we defined the syntax, static and dynamic semantics for these object languages, as well as we determined some language-specific refactoring schemes with which we can express meaningful refactoring transformations. In this section, we overview the challenges of instantiating the framework for programming languages in general, and for Erlang and Java in particular.

## 5.1 Parametrising the framework

Beside providing a formal definition of the object programming language from syntax to semantics, instantiation needs the identification of recurring transformation patterns and understanding of dependencies induced among program elements. Ideally, when parametrising the framework for yet another language, the formal definition already exists (syntax, operational semantics, static semantic analysis), yet it is challenging to find those reusable and verifiable schemes for transformations. In the following sections we discuss our framework and its prototype implementations from this perspective.

### Defining the metatheory

The semantic predicates provide a high-level interface for embedding static semantic information about the target language into both schemes and scheme instances. Most notably, the metatheory defines what predicates refactoring preconditions should be built from. Constructing the right metatheory for a specific target language is about finding a characterization of its abstractions suitable for both the execution and verification backend.

Naturally, the chosen characterization must be expressive enough to make the preconditions of schemes and scheme instances specifiable. In addition, its elements should also be computable from the underlying program model while executing a concrete refactoring. When composed correctly, the identified functions and predicates must carry enough information to make verification possible.

A starting point towards an appropriate metatheory can be based on the abstractions of the target language, which, of course, are highly influenced by its paradigm(s). Then, the chosen semantic predicates can be iteratively refined in accordance with the requirements above.

### Defining semantic equivalence

The underlying notion of semantic equivalence is probably the most determining aspect of a refactoring. Indeed, it is the basis of both intuitional and formal correctness. An oversimplified definition of equivalence can be as abstract as demanding observed programs to produce the same output for the same input. The problem with this, however, is that it is not concrete enough to be conveniently expressible using a proper metatheory. Therefore we propose to replace the aforementioned definition of equivalence with one of its – more easily specifiable – characterizations, e.g. the conformity of data flow, control flow and binding.

We also have to consider that a refactoring usually transforms only some parts of a program instead of its entirety. Generally, a transformation scope specifying the extent of the modified code can be attached to each refactoring. Our assumption is that rather than using a general notion of equivalence – or one of its characterizations –, it is more intuitive to introduce a stricter, but localized variant for each possible transformation scope. In our framework, transformation scope, and therefore equivalence level, can be matched with refactoring schemes.

**Designing language-specific schemes**

As mentioned earlier, designing refactoring schemes can be the most challenging part of the framework specialization process. There are several key aspects which should be taken into consideration, e.g. generality, usability, verifiability, etc. Schemes must be general enough to be reusable, but not too general, as that would make their instantiation undesirably difficult. On the other hand, we must aim for schemes which optimally split the verification problem, that is checking the general correctness of a scheme wrt. to a contract concerning the rewrite rules it is parameterized by, and checking whether scheme instances satisfy these contracts.

We propose two iterative methods for scheme construction: top-down and bottom-up. The top-down approach starts from a higher level of abstraction, e.g. the level of language elements, and tries to identify schemes based on possible dependencies between the discussed entities. The basis of the bottom-up direction is a number of complex, desirably representative refactorings of the target language, which are then decomposed to microsteps, from which schemes are obtained by appropriate generalization.

Both methods have their advantages and disadvantages. With the top-down method, schemes are inherently general, but not necessarily usable. On the contrary, schemes obtained with the bottom-up method are usable by definition, but their generality is not guaranteed. In both cases, further refinement iterations are required to mitigate these weaknesses. In the former case, more high-level concepts can be added to the dependency analysis; in the latter, more refactorings can be considered during the generalization.

## 5.2 Erlang

The first implementation of our refactoring framework was made for the Erlang programming language, via generalization of an analysis and transformation tool [2] implemented in Erlang. This preliminary work had a high influence on how we model the program, split syntax and semantics, and even on the separation principle of analysis and transformation. The analysis and transformation system the Erlang implementation of the framework is built on uses automatic, incremental static analysis to keep the semantic layer consistent with the AST; in fact, our framework implementation makes heavy use of the underlying original transformation system.

Erlang has a fairly simple syntax. The static semantics is mainly about language abstractions (modules, functions, variables, types) and their static semantic properties (such as scopes or purity). Basic schemes for Erlang were constructed upon primary sources of data and control dependencies in the language: function calls, variable binding, data-flow have been identified as schemes of extensive changes. Some case studies have been formalized already in the refactoring specification language with the Erlang-specific schemes, one of those is available in detail in [11].

### 5.3 Java

The metatheory we constructed for the Java prototype characterizes abstractions from the object-oriented paradigm, most notably inheritance, polymorphism and dynamic binding. Here we used the bottom-up approach to obtain schemes by choosing the *lift segment*[2] refactoring as the base transformation. Its decomposition and generalization resulted in four schemes – local, block, lambda and class – and three equivalence levels – local, block and class. For the verification backend, we used K-Java [1] as operational semantics.

The implementation is built on top of a DSL-engineering framework called Xtext, which comes with Xbase, a reusable, Java-like expression language. By modifying its grammar to resemble Java more closely and to accommodate metavariables, a parser for refactoring definitions could be generated automatically. These definitions are compiled on-the-fly to Java code which uses the refactoring API of the Eclipse IDE. Finally, the translated code is dynamically loaded into the underlying JVM instance and made available from an Eclipse plugin. Our approximation of the metatheory is implemented with the Java Development Tools (JDT).

## 6 Discussion

The ideas discussed in this paper were inspired by a study on high-level, declarative refactoring definitions for Erlang [11]. The concepts of semantic predicates, refactoring idioms and composition operators all seemed to be language-independent, so we adapted the original idea to Java by re-implementing the entire project with JDT and Xtext. After that, we were certain that the two solutions should share a couple of elements that are fairly language-independent.

Apparently, the existing concepts and implementations had to be redesigned in order to expose and extract those language-independent portions, but we managed to obtain a generic design. Although Section 4 presented a fully language-parametric architecture, the proof of concept implementations for Erlang and Java do not share all these language-independent components yet. Nevertheless, realisation of the generic framework for these two substantially different languages justifies that the concept is viable; it is our long-term plan to implement the Erlang and Java tools as proper instances of the language-generic framework.

Before concluding the paper, we briefly evaluate how, and to what extent, the proposed approach allows for generic and trustworthy implementation of refactoring. We also address the idea of language-independent schemes and discuss work in progress on changing the way verification is built into the process.

### 6.1 Genericity

The proposed design is language-generic: the refactoring specification language is independent of the object language, as well as the implementation framework is

[2] Refactoring *lift segment* lifts a code segment into the superclass as a newly introduced method.

language-parametric. Namely, when a new language needs to be supported, the framework is instantiated for the particular programming language by providing a formal definition of the language (syntax, static and dynamic semantics) along with refactoring schemes (transformation idioms). The main components of execution and verification are shared between instances for different languages.

How do we achieve this? We sort of rephrase and restructure the usual way of defining and implementing a refactoring, and this rephrasing allows us to cut out and abstract away some language-specific elements. In particular, the high-level program model lets term rewrite rules incorporate semantic predicate conditions, and it allows strategies to control term traversal based on semantic dependencies. This separation of transformation concerns leads to a clear separation of the so-called refactoring business logic, which, on the other hand, can be defined in a declarative and language-independent way. The language-independence of the refactoring specification directly implies that the interpretation of specifications can rely on components parametrised with language-specific artefacts.

**Language-independence of refactoring schemes.** Language-level refactoring schemes enable high-level description of transformations that respect lower-level dependencies. Parametric verification of schemes involves definition of specific equivalence classes of programs, which in turn imply full semantic equivalence under certain circumstances. Even though schemes seem to be totally language-dependent, we have identified some schemes of schemes: for instance, in many programming languages the abstraction of subroutines exists in some way. Function refactoring in Erlang and method refactoring in Java may be understood as specialisations of a language-independent scheme. Schemes may stem from concepts that are shared among different languages, and in the long term, we plan to investigate the possibility of implementing a set of schemes that are defined in terms of concepts common in various languages.

## 6.2 Trustworthiness

Trustworthiness comes in many forms, ranging from simplicity, modular implementation or open-source code base with excessive testing. In our paper, we focused on enabling semi-automatic formal verification for behaviour-preservation in semantic program model transformations. We managed to split the transformation definition to traversal control and actual term rewriting in a semantics-driven way, which in turn allows these two parts be verified separately, with the latter done semi-automatically.

**How do we prove transformations correct?** Local refactorings are fairly simple to check. Since these are composed of one single rewrite rule, the correctness is expressed as one conditional equivalence statement of two program or expression patterns. If, under the side-conditions, the rewriting preserves the semantics, the transformation is correct.

For changes that span over multiple expressions, subroutines or even modules, a notion of "completeness" and "consistency" is needed. Namely, the rewriting has to visit all interdependent elements in the program and carry out consistent modifications to preserve the semantics of the entire program. These properties are ensured by the schemes, which provide a declarative means to express complex refactorings. For these extensive changes, we synthesise a set of equivalence formulas that are checked for validity by using the dynamic semantics of the language.

With schemes, we reduce the global equivalence problem to multiple local equivalence problems. This is enabled by decoupling traversal control and actual rewriting. Schemes are verified with respect to some conditions on the rewrite rules they are parametrized with. Verification of schemes requires manual proving; however, having the pre-verified schemes, the instantiation conditions may be automatically checked, making the scheme-based refactoring definitions automatically verifiable. The conditions are equivalence statements on term patterns. Verification of pattern equivalence is not decidable, but in a lot of cases, advanced, problem-specific proof tactics can lead to equivalence proofs. If we express the equivalence formula in reachability logic, there is an algorithm [5] that can be used to determine whether the two patterns can be rewritten to the same form by using rules in the the operational semantics of the language.

Although automatic verification of scheme instances would be a convenient feature from the user's perspective, due to the undecidability of pattern equivalences, in most cases the proving requires some human assistance. We started to redesign the framework such that the object language semantics is formalised in a proof assistant and the pattern equivalence proofs are written by hand. This is fundamentally different from the K framework based solution, but gives more control and opportunities to the user of our system.

# 7 Conclusion

Refactoring program transformations are essential in large-scale software development for maintaining code quality. Tools that carry out such transformations need to be trustworthy: there has to be an evidence that the program after the refactoring still behaves the same as before. Correctness of the transformation can be checked for each and every application instance, but the ultimate guarantee on correctness is obtained by the static verification of the refactoring definition.

In our previous work, we have investigated refactoring definition abstractions for different object languages, which allow for semi-automatic verification for correctness. In this paper, we have advanced these previous results by generalising our approach over different object languages and designing a unifying refactoring framework. We have shown that the high abstraction level of the definition enables a fine-grained separation of the various components in a refactoring tool, which in turn allows the recognition and extraction of language-dependent elements, leading to a language-generic implementation. Our proposed solution facilitates execution and static verification of refactoring definitions for different object languages.

## References

[1] Bogdănaş, Denis and Roşu, Grigore. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. DOI: 10.1145/2676726.2676982.

[2] Bozó, István, Horpácsi, Dániel, Horváth, Zoltán, Kitlei, Róbert, Kőszegi, Judit, Tejfel, Máté, and Tóth, Melinda. RefactorErl – Source Code Analysis and Refactoring in Erlang. In *Proceedings of SPLST'11*, pages 138–148, Tallin, Estonia, 2011. URL: https://www.researchgate.net/publication/289641474.

[3] Bravenboer, Martin, van Dam, Arthur, Olmos, Karina, and Visser, Eelco. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inf.*, 69(1-2):123–178, July 2005. ISSN: 0169-2968.

[4] Cesarini, Francesco and Thompson, Simon. *ERLANG Programming.* O'Reilly Media, Inc., 1st edition, 2009. ISBN: 0-596-51818-8.

[5] Ciobaca, Stefan, Lucanu, Dorel, Rusu, Vlad, and Rosu, Grigore. A Language-Independent Proof System for Full Program Equivalence. *Formal Aspects of Computing*, 28(3):469–497, May 2016. DOI: 10.1007/s00165-016-0361-7.

[6] Cohen, Julien. Renaming Global Variables in C Mechanically Proved Correct. In Hamilton, Geoff, Lisitsa, Alexei, and Nemytykh, Andrei P., editors, Proceedings of the Fourth International Workshop on *Verification and Program Transformation,* Eindhoven, The Netherlands, 2nd April 2016, Volume 216 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–64. Open Publishing Association, 2016. DOI: 10.4204/EPTCS.216.3.

[7] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[8] Garrido, A. and Meseguer, J. Formal Specification and Verification of Java Refactorings. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 165–174, Sept 2006. DOI: 10.1109/SCAM.2006.16.

[9] Gil, Yossi, Marcovitch, Ori, and Orrú, Matteo. A nano-pattern language for java. *Journal of Computer Languages*, 54:100905, 2019. DOI: 10.1016/j.cola.2019.100905.

[10] Gómez, Verónica Uquillas, Ducasse, Stéphane, and D'Hondt, Theo. Ring: A unifying meta-model and infrastructure for smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44 – 60, 2012. DOI: 10.1016/j.cl.2011.11.001, SMALLTALKS 2010.

[11] Horpácsi, Dániel, Kőszegi, Judit, and Horváth, Zoltán. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In Lisitsa, Alexei, Nemytykh, Andrei P., and Proietti, Maurizio, editors, Proceedings Fifth International Workshop on *Verification and Program Transformation,* Uppsala, Sweden, 29th April 2017, Volume 253 of *Electronic Proceedings in Theoretical Computer Science*, pages 92–108. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.253.8.

[12] Kniesel, Günter and Koch, Helge. Static Composition of Refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, August 2004. DOI: 10.1016/j.scico.2004.03.002.

[13] Kosar, Tomaž, Bohra, Sudev, and Mernik, Marjan. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77–91, 2016. DOI: 10.1016/j.infsof.2015.11.001.

[14] Lämmel, Ralf. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, pages 15–28, New York, NY, USA, 2002. ACM. DOI: 10.1145/570186.570188.

[15] Lecerf, Jason, Brant, John, Goubier, Thierry, and Ducasse, Stéphane. A Reflexive and Automated Approach to Syntactic Pattern Matching in Code Transformations. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 426–436, 2018. DOI: 10.1109/ICSME.2018.00052.

[16] Leitão, António Menezes. A Formal Pattern Language for Refactoring of Lisp Programs. In *Proceedings of CSMR '02*, pages 186–192, Washington, DC, USA, 2002. IEEE Computer Society. DOI: 10.1109/CSMR.2002.995803.

[17] Li, Huiqing and Thompson, Simon. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Proceedings of FASE'12*, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. DOI: 10.1007/978-3-642-28872-2_34.

[18] Lämmel, Ralf, Thompson, Simon, and Kaiser, Markus. Programming errors in traversal programs over structured data. *Science of Computer Programming*, 78(10):1770 – 1808, 2013. DOI: 10.1016/j.scico.2011.11.006.

[19] Maruyama, Katsuhisa and Yamamoto, Shinichiro. Design and Implementation of an Extensible and Modifiable Refactoring Tool. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 195–204. IEEE, 2005. DOI: 10.1109/WPC.2005.17.

[20] Mendonca, Nabor C., Maia, Paulo Henrique M., Fonseca, Leonardo A., and Andrade, Rossana M. C. RefaX: A Refactoring Framework Based on XML. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 147–156, Washington, DC, USA, 2004. IEEE Computer Society. DOI: 10.1109/ICSM.2004.1357799.

[21] Opdyke, William F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645. URI: http://hdl.handle.net/2142/72072.

[22] Roberts, Don, Brant, John, and Johnson, Ralph. A Refactoring Tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, October 1997. DOI: 10.1002/(SICI)1096-9942(1997)3:4⟨253::AID-TAPO3⟩3.3.CO;2-I.

[23] Roberts, Donald Bradley. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999. URI: http://hdl.handle.net/2142/81948.

[24] Roşu, Grigore, Ştefănescu, Andrei, Ciobâcă, Ştefan, and Moore, Brandon M. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013. DOI: 10.1109/LICS.2013.42.

[25] Rowe, Reuben N. S., Férée, Hugo, Thompson, Simon J., and Owens, Scott. Characterising renaming within OCaml's module system: theory and implementation. In McKinley, Kathryn S. and Fisher, Kathleen, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 950–965. ACM, 2019. DOI: 10.1145/3314221.3314600.

[26] Schaefer, Max and de Moor, Oege. Specifying and Implementing Refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010. DOI: 10.1145/1932682.1869485.

[27] Sultana, Nik and Thompson, Simon. Mechanical Verification of Refactorings. In *Proceedings of PEPM '08*, pages 51–60, New York, NY, USA, 2008. ACM. DOI: 10.1145/1328408.1328417.

[28] Verbaere, Mathieu, Ettinger, Ran, and de Moor, Oege. JunGL: A Scripting Language for Refactoring. In *Proceedings of ICSE '06*, pages 172–181, New York, NY, USA, 2006. ACM. DOI: 10.1145/1134285.1134311.

[29] Visser, Eelco and Benaissa, Zine-El-Abidine. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15:422–441, 1998. DOI: 10.1016/S1571-0661(05)80027-1.

Acta Cybernetica **25** (2022) 781–795.

# Report on the Differential Testing of Static Analyzers*

Gábor Horváth[ab], Réka Kovács[ac], and Péter Szécsi[ad]

**Abstract**

Program faults, best known as bugs, are practically unavoidable in today's ever growing software systems. One increasingly popular way of eliminating them, besides tests, dynamic analysis, and fuzzing, is using static analysis based bug-finding tools. Such tools are capable of finding surprisingly sophisticated bugs automatically by inspecting the source code. Their analysis is usually both unsound and incomplete, but still very useful in practice, as they can find non-trivial problems in a reasonable time (e.g. within hours, for an industrial project) without human intervention.

Because the problems that static analyzers try to solve are hard, usually intractable, they use various approximations that need to be fine-tuned in order to grant a good user experience (i.e. as many interesting bugs with as few distracting false alarms as possible). For each newly introduced heuristic, this normally happens by performing differential testing of the analyzer on a lot of widely used open source software projects that are known to use related language constructs extensively. In practice, this process is ad hoc, error-prone, poorly reproducible and its results are hard to share.

We present a set of tools that aim to support the work of static analyzer developers by making differential testing easier. Our framework includes tools for automatic test suite selection, automated differential experiments, coverage information of increased granularity, statistics collection, metric calculations, and visualizations, all resulting in a convenient, shareable HTML report.

**Keywords:** static analysis, symbolic execution, Clang, testing

*The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: xazax@caesar.elte.hu, ORCID: 0000-0002-0834-0996

[c]E-mail: rekanikolett@caesar.elte.hu, ORCID: 0000-0001-6275-8552

[d]E-mail: ps95@caesar.elte.hu, ORCID: 0000-0001-9156-1337

DOI: 10.14232/actacyb.282831

# 1 Introduction

Any significant change to an open-source static analysis tool (also simply called *an analyzer*) is preceded by a discussion about its possible effects. The minimum typical requirement is the comparison of analysis results and performance on a few software projects before and after applying the changes.

Fulfilling this requirement is hard for various reasons. Developers often have a bias towards a set of projects they are familiar with, which might tempt them to avoid the challenge of finding a set of test projects that most effectively exercise the changed parts of the analyzer. In case of a long-lasting open-source review process [16] (developers often have to wait half a year before their contributions get accepted), changes need to be re-based on top of the latest version of a continuously evolving code base, and the analysis of all test projects needs to be re-run to ensure that the feature still works correctly. Analysis results also have to be processed and summarized to be easily understandable for the reviewers.

Note that, for our explanations throughout the paper, we use the term *author* to refer to the person who implements a change to the open-source application (this comes from being the author of the *patch*, a textual form of the set of actual modifications to the source code). This person has to justify the changes and prove to *reviewers* that there will be no unwanted regressions in the software's behavior. By *reviewers* we mean those fellow developers who audit and approve the changes.

Ideally, reproducing an analysis should be painless, and it should be possible to present results in an easily shareable and digestible format. This format should be simple to archive or embed in documentation, so that major design decisions can be easily re-evaluated later. This is important, since the decisions that make perfect sense today might be less adequate tomorrow.

In this paper, we present our toolchain that we call the *Clang Static Analyzer Testbench* (or *CSA Testbench*)[1], designed for the Clang Static Analyzer [1] and Clang-Tidy [2], two open-source static analysis tools built on top of the Clang [11] compiler for C family languages. The Clang Static Analyzer uses symbolic execution [10] to find bugs, while Clang-Tidy is a collection of syntactic checks. We are long-term contributors to these tools, and would like to share the principles of the differential testing infrastructure we have built with a wider community.

Our framework aims to enhance the open-source review process by supporting *reviewers* and *authors* (as defined earlier) in the following ways:

- help authors select a set of relevant projects for testing,
- help authors run static analysis on the selected projects,
- aggregate statistics about the analysis (e.g.: how often a cut heuristic is triggered while building the symbolic execution graph),
- aggregate the results of the analysis, i.e. the reported warnings,
- help authors and reviewers evaluate and share the results,

[1] The code is open-source, licensed under the MIT License, and can be downloaded from https://github.com/Xazax-hun/csa-testbench.

- help reviewers reproduce the results and suggest changes to the test setup,
- help authors maintain the tests.

The input of the toolset is a single and easy-to-interpret configuration file in JSON format. Since the format is textual, reviewers can comment on the test setup using conventional review tools and it can also be embedded in documentation. Moreover, it is convenient to store such files in version control systems. The output is a customizable HTML report with useful information, various plots, and a record of the input configuration, including the version numbers, to ensure reproducibility. The goal is to store all the information required to repeat the experiment.

Our principles can be reused by developers of other static analyzers, and we also describe some alternative use cases for our framework.

The paper is structured as follows. Section 1 gives an overview of the difficulties faced during an open-source review process that requires differential testing. Section 2 introduces the principles behind the framework we built to tackle these problems.

# 2 The CSA Testbench Toolchain

## 2.1 Semi-automatic test suite generation

**Problem** After implementing a missing feature or tweaking an existing part of a static analyzer, testing the robustness of the change and checking whether a regression occured is a natural requirement towards the author. One conventional approach is to run the analyzer tool on a number of real-world software projects and artificial regression tests.

Finding a sufficient number of relevant real-world projects can be challenging. Ideal projects should be open-source and easy to set up, so that reviewers have a better chance of reproducing the results. Additionally, projects should exercise the right parts of the analyzer. For example, if the change is related to dynamic type information modeling, only projects using dynamic type information should be included.

One option is to use a trial-and-error approach and check a random sample of open-source projects, hoping to find enough that display the required traits. A slightly better approach is to use code searching and indexing services and look for projects with interesting code snippets. These services, however, are optimized to present the individual snippets and suboptimal to retrieve the most relevant projects according to some criteria.

**Solution** We present a script that harvests the results of an existing code search service, and recommends projects to be included in the test suite based on the results. This script can spare a significant amount of development time and help authors find relevant projects on which their changes can be verified.

Our script uses the SeachCode [5] service for its backend. For example, in order to test a new static analysis check written to detect `pthread_mutex_t` abuse, we might be interested in projects that use `pthread` extensively. Using the syntax on Listing 1, we can specify the keywords to search for, the languages we are interested in, the desired number of projects and optionally a filename for the output:

```
$ ./gen_project_list.py 'pthread_mutex_t' 'C C++' 3 -o pthread.json
```

Listing 1: A sample invocation of the project list generator tool.

This call creates a configuration file with the suggested projects in the following format:

```
{
  "projects": [
    {
      "url":"github.com/itkovian/torque.git",
      "name": "torque"
    },
    {
      "url":"github.com/snktagarwal/openafs.git",
      "name": "openafs"
    },
    {
      "url":"github.com/cfenoy/slurm.git",
      "name": "slurm"
    }
  ]
}
```

Listing 2: A fraction of a configuration file generated by the project list generator tool invocation showed on Listing 1.

This configuration file can be directly used as input to the main driver script of the testing infrastructure as detailed in Section 2.2.

Sometimes we only want to do a *stress test* to ensure that the analysis engine behaves gracefully for all projects and does not crash. We created an alternative tool to create a configuration file based on a Debian FTP mirror for package sources. The resulting file will contain more than 20 000 projects.

```
$ ./project_list_from_debian.py \
  --url ftp://ftp.se.debian.org/debian/ --output debian.json
```

Listing 3: Sample call of an alternative project list generator tool that lists all packages available at the specified Debian mirror.

## 2.2 Easy analysis reproduction and sharing

**Problem** A regular pattern is that the developer sharing text files that contain static analysis results on a set of projects. This makes evaluation considerably difficult for reviewers. First of all, they might not be familiar with the test projects at all. Text dumps of static analysis results are hard to interpret and the measurements are hard to reproduce. Further questions that might arise: How did the author compile the project? Which version of the analyzed project was used? How did the author invoke the analyzer? Which configuration options were used? Which revision (commit) of the analyzer was used?

**Solution** Our tools use a concise configuration format that contains all the relevant information about the analyzed projects: repository, tag/commit, configuration options for the analysis, etc. Obtaining this configuration file enables reviewers to reproduce the exact same measurements, with the help of our driver script. They can also easily suggest modifications to the conducted experiment.

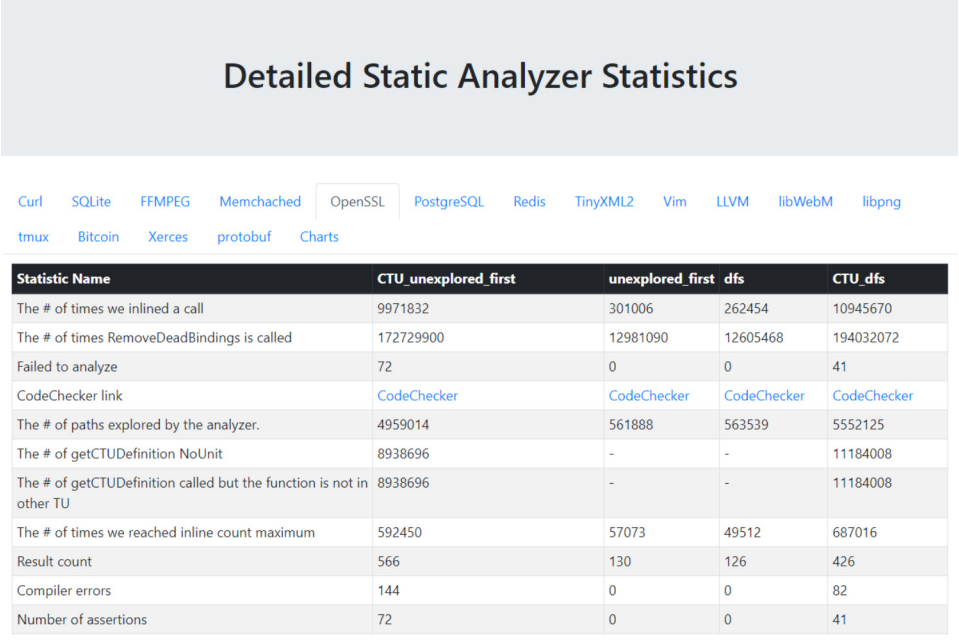
The scripts aggregate useful information about the analysis into an easy-to-share HTML format (as seen in Figure 1). Analysis results are not mere text dumps anymore, but are presented on a convenient web user interface that also displays the path associated with the report (showed in Figure 2). Other information such as the number of code lines in the project, version of the analyzer, analysis time, analysis coverage (Figures 3 and 4), and statistics from the analysis engine is recorded and charts are generated automatically (Figure 5). The web user interface also has permanent links to each individual error report in order to make it easier to refer to them in discussions. These pages are hosted by the person sharing the results, code reviewers do not need to install anything to browse the results.

The configuration file showed in Section 2.1 is almost enough to run the analysis on its own. The only extra information needed to be specified is the URL of the *CodeChecker* server where analysis results are intended to be stored for later inspection (Listing 4).

```
1  {
2    "projects" : ...
3    "CodeChecker": {
4      "url" : "localhost:15010/Default",
5    }
6  }
```

Listing 4: A segment of the configuration file specifying the address of the CodeChecker server.

*CodeChecker* [3] is a tool we designed to integrate the Clang Static Analyzer and Clang-Tidy into C/C++ build systems. It also acts as a mature bug management system that supports commenting on static analysis reports and suppressing false positives. It has a convenient user interface to visualize path-sensitive bug reports (see Figure 2) and to support differential analysis. We can compare two analysis



Detailed Static Analyzer Statistics

Curl SQLite FFMPEG Memchached OpenSSL PostgreSQL Redis TinyXML2 Vim LLVM libWebM libpng tmux Bitcoin Xerces protobuf Charts

| Statistic Name | CTU_unexplored_first | unexplored_first | dfs | CTU_dfs |
|---|---|---|---|---|
| The # of times we inlined a call | 9971832 | 301006 | 262454 | 10945670 |
| The # of times RemoveDeadBindings is called | 172729900 | 12981090 | 12605468 | 194032072 |
| Failed to analyze | 72 | 0 | 0 | 41 |
| CodeChecker link | CodeChecker | CodeChecker | CodeChecker | CodeChecker |
| The # of paths explored by the analyzer. | 4959014 | 561888 | 563539 | 5552125 |
| The # of getCTUDefinition NoUnit | 8938696 | - | - | 11184008 |
| The # of getCTUDefinition called but the function is not in other TU | 8938696 | - | - | 11184008 |
| The # of times we reached inline count maximum | 592450 | 57073 | 49512 | 687016 |
| Result count | 566 | 130 | 126 | 426 |
| Compiler errors | 144 | 0 | 0 | 82 |
| Number of assertions | 72 | 0 | 0 | 41 |

Figure 1: A section of the automatically produced HTML report containing information about analysis runs with different analyzer configurations. The table contains links to the corresponding analysis runs in a web user interface (see Figure 2), and links to detailed line-based coverage reports (Figures 3 and 4). A similar table for each analyzed project can be found under appropriately labeled tabs in the header of the report. The *Charts* tab hides a number of interactive charts generated from the results (for an example, see Figure 5).

runs using CodeChecker to differentiate between common reports and those present only in a specific analysis run. CodeChecker's web GUI allows sharing the results with the rest of the world without needing to repeat the experiment. It can be used to share not only bug reports, but also classifications and comments explaining why some findings are considered false positives or true positives.

After adding this detail to the configuration file, we are ready to run the analysis on the previously selected set of projects (Listing 5).

```
1 $ ./run_experiments.py --config pthread.json
```

Listing 5: Sample invocation of the main driver script of the experiment.

The script checks out each project, attempts to infer their build system, builds them, runs the analysis, and finally collects the results. At the time of writing this paper `autotools`, `CMake`, and `make` are supported out of the box.

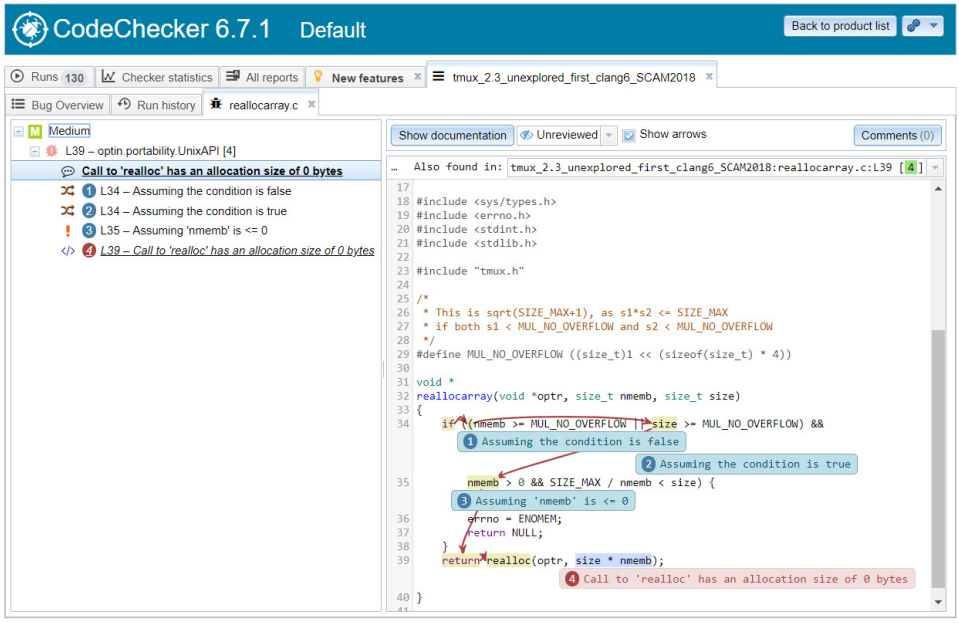



Figure 2: The CodeChecker web user interface. Path-sensitive reports guide the user along the execution path leading to the bug. On the web user interface, different runs can be compared against each other, and bug reports can be filtered by many criteria, e.g. by severity, by review status, by detection status, by detection date, by checker name, by checker message, etc. Bug reports can also be marked false positive, with the possibility of leaving an explaining note for the record.

However, the script will not download and install all the dependencies required to compile the projects. It is the user's responsibility to ensure that the host machine is able to compile the projects, which turned out to be a big burden for the authors. For this reason, we introduced support for the two emerging C++ package managers, Conan [4] and Vcpkg [6]. Relying on these package managers instead of repository URLs ensures that the analysis will not fail due to a missing dependency. In Listing 6, we can see how easy it is to test on a project which is available in one of the package managers.

```
{
  "projects": [
    {
      "name": "zlibconan",
      "package": "zlib/1.2.11@conan/stable",
      "package_type": "conan",
    },
    {
      "name": "zlibvcpkg",
      "package": "zlib",
```

```
      "package_type": "vcpkg",
    }
  ]
}
```

Listing 6: A sample configuration file that will instruct the framework to download projects using the *Conan* and *Vcpkg* package managers.

In case a special build command is required, or the build system is not yet supported, the user can specify the build command and the configuration command. Building a specific version of the project determined by a tag, a commit hash, or a URL to a source tarball instead of top of tree is also possible and highly encouraged, in order to get consistent results in subsequent experiments.

Finally, differential analysis can currently be conducted by running the same projects multiple times with different options passed to the analyzer or using different versions of the analyzer (Listing 7).

```
{
  "projects": [
    {
      "url": "github.com/itkovian/torque.git",
      "name": "torque",
      "tag": "tag name",
      "build_command": "special build command"
    }, ...
  ]
  "configurations": [
    {
      "name": "original",
      "clang_sa_args": "",
    },
    {
      "name": "variant A",
      "clang_sa_args": "argument to enable feature A",
      "clang_path": "path to clang variant"
    }
  ], ...
}
```

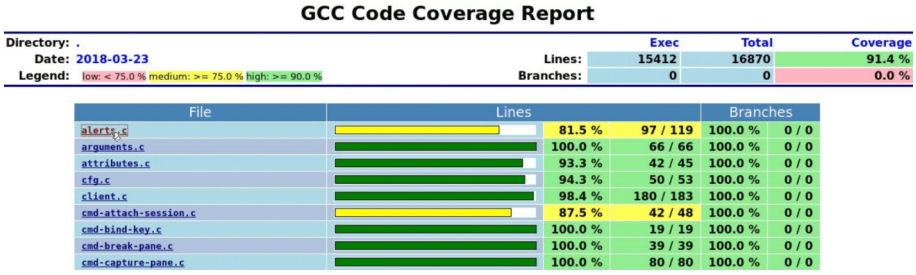
Listing 7: Differential testing can be achieved by running many analysises on the same projects with different options passed to the analyzer.

## 2.3 A more precise differential analysis

**Problem** Currently, coverage measurements provided by the Clang Static Analyzer are limited. The engine can only record the percentage of basic blocks reached during the analysis of a translation unit, which is not sufficiently precise for multiple reasons. First, the analysis can stop in the middle of a basic block due to running out of the analysis budget for that specific execution path. Secondly, there

is no precise way of merging information from different translation units. Finally, inline functions or templates in header files might appear in multiple translation units and their contribution will be counted multiple times upon attempting to aggregate information over translation units.

**Solution** We implemented line-based coverage measurement based on the `gcov` [7] format. We do not calculate coverage as an overall percentage value, but record it separately for each line. This makes it possible to precisely aggregate coverage information over translation units, and to do differential analysis on the coverage itself. Our toolset includes scripts to aid that kind of analysis.



**GCC Code Coverage Report**

| | | Exec | Total | Coverage |
|---|---|---|---|---|
| Directory: . | | | | |
| Date: 2018-03-23 | Lines: | 15412 | 16870 | 91.4 % |
| Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 % | Branches: | 0 | 0 | 0.0 % |

| File | Lines | | Branches | |
|---|---|---|---|---|
| alerts.c | 81.5 % | 97 / 119 | 100.0 % | 0 / 0 |
| arguments.c | 100.0 % | 66 / 66 | 100.0 % | 0 / 0 |
| attributes.c | 93.3 % | 42 / 45 | 100.0 % | 0 / 0 |
| cfg.c | 94.3 % | 50 / 53 | 100.0 % | 0 / 0 |
| client.c | 98.4 % | 180 / 183 | 100.0 % | 0 / 0 |
| cmd-attach-session.c | 87.5 % | 42 / 48 | 100.0 % | 0 / 0 |
| cmd-bind-key.c | 100.0 % | 19 / 19 | 100.0 % | 0 / 0 |
| cmd-break-pane.c | 100.0 % | 39 / 39 | 100.0 % | 0 / 0 |
| cmd-capture-pane.c | 100.0 % | 80 / 80 | 100.0 % | 0 / 0 |

Figure 3: A sample report summarizing coverage percentages over the analyzed files. Line-based coverage information can be browsed by clicking on filenames.

In some cases, we are interested in the reason behind a specific bug report disappearing when running the analysis with different parameters. Performing differential analysis on the coverage, we are able to determine whether the analyzer actually examined the code in question during both runs.

The Clang Static Analyzer can output different kinds of statistics such as the number of paths examined, the number of times a specific cut heuristic was used etc. Instead of having a fixed set of statistics to collect, we used some text mining to process the output of the analyzer, in which we are able to automatically detect newly added custom statistics without any additional configuration, and aggregate them over translation units.

As mentioned in Section 2.2, the final report of our toolset includes figures like charts and histograms. The list of figures can be set in the configuration file. After adding a new statistic to the analyzer engine, the author only needs to add a single entry in the configuration file to make the toolset generate a figure based on that statistic. One sample use-case is producing a histogram of analysis times per translation unit. This can help us track down performance regressions in outliers.

We cannot emphasize the importance of automatically generated figures enough. The statistics about a run of the symbolic execution engine is not easy to interpret. For example, an increase in the number of generated symbolic states can be both a good and bad news depending on how the rest of the statistics are changed. Not requiring the author to create the figures from the numbers manually is a great productivity boost.


Figure 4: Our amended version of the Clang Static Analyzer can provide coverage information for each executed line of each analyzed file. In Figure 3, lines covered by the analysis are shown in a green color, while lines not covered are shown in red. White lines contain no executable code.

C-Reduce [15] is a tool that takes a large C, C++, or OpenCL file that has a property of interest (such as triggering a compiler bug) and automatically produces a much smaller C/C++ file that has the same property. We also use C-Reduce to get minimal examples that showcase differences between two versions of the static analysis engine. First, we need a file on which analysis engine versions produce different results. This can be a different set of warnings or other statistics emitted by the engine. These minimal examples can greatly aid our understanding of the effects of a change. The main shortcoming of C-Reduce is the lack of support for reducing multiple translation units at once. We do plan to add this feature in the future.

## 2.4 Recommended workflow, how to use the toolchain

Using our toolset the recommended workflow is shown in Figure 6. The author of the patch uses some of our scripts to select the project to test the changes on. After running the experiments she makes sure all the data support the hypothesis. Then she uploads tha patch for review and provides reviewers with a link to the test results which includes the configuration. Reviewers can choose to either merely look at the results or repeat the whole experiment based on the configuration, depending on the verification effort required for the change. They can also suggest

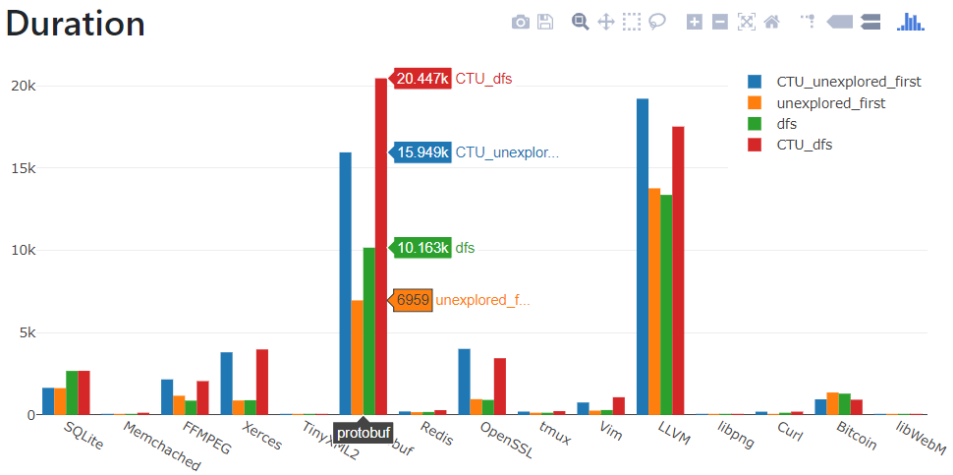



Figure 5: One of the many interactive charts generated based on statistics collected during analysis, this figure shows duration times for different analyzer configurations for different open-source projects. Precise numbers are shown when hovering over the block of columns corresponding to a project.

changes to the configuration to gather more insight about the changes. After such suggestions it is as easy to re-run the whole experiment as pushing a button.

## 2.5 Alternative applications

The tools we introduced in the previous section can be generalized beyond supporting only static analysis engines. First, obtaining a set of projects with certain properties (e.g. projects using runtime type information) can be valuable for the testing of any language tool. Secondly, the ability to check out and analyze any number of past tags of a project and perform differential analysis on them enables the collection of historical data about the evolution of the project's coding conventions. We can also track the number of findings over time for a certain project.

We also found that these scripts are great to build CI loops. Running the analysis on a set of projects for each commit is a great way to find regressions. We introduced a flag to break the CI loop each time the analysis of a project fails for some reason. The reported HTML will contain useful information about the analysis failures as well as assertion messages.

Finally, one of the most interesting applications of our scripts is automatic parameter tuning. Some static analysis engines have a great number of adjustable parameters. Our tools are not only suitable for running the analysis, but also for setting its parameters and measuring time, coverage, engine statistics, and the number of reported bugs. Using this information, a machine learning algorithm can attempt to optimize the parameters in order to improve the quality of the analysis.

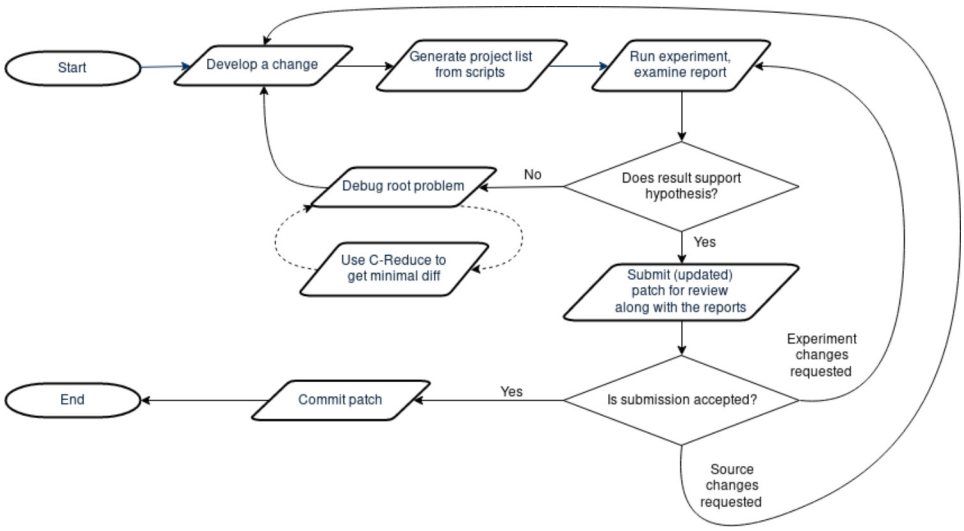



Figure 6: A flowchart describing the recommended workflow when using the CSA Testbench to do differential testing of an analyzer change.

### 2.6 Future work

Unfortunately, using textual queries to get a set of interesting projects is not sufficient. There are certain language constructs that are hard to query this way, such as implicit casts or structured bindings. Likewise, using code search services is also an imperfect solution, a semantic indexer would probably be more suitable.

We intend to introduce more (optional) measurements into the scripts such as memory profiling during analysis. We also plan to perform a more detailed analysis of how the proposed process can improve the quality of the static analysis engine.

These set of tools are the result of optimizing the productivity of our team while working on some static analysis tools. While each added feature helped to improve our work-flow, it is hard to quantify the improvements. We plan to conduct some surveys in the future to verify the usefulness of our framework among a wider community of researchers and developers.

## 3 Related Work

The difficulty of performing static analysis varies among programming languages, due to differences in the number and maturity of tools written for them. Two languages on the worse end of the spectrum are C and C++, as no widely used build system or package repository exists in their fragmented ecosystem. Having tools to deal with software repositories directly can be a step towards overcoming this problem and helping researchers perform more rigorous evaluations for their tools targeting these languages. Since C++ is a language of enormous size, most

projects use a relatively small subset of it. For this reason, finding a good set of test projects is even more critical.

This problem is less likely to surface during the analysis of other languages. Some of them, like Java, armed with Maven repositories, are in a convenient position for experimentation. Software packages can be easily downloaded, built and analyzed. Fortunately, the C++ community realized the value of having package managers, and now two of them named Conan [4] and vcpkg [6] started to gain popularity, but have not reached wide adoption yet.

In the following paragraphs, we describe tools that play a similar role for other programming languages than our framework for C++.

VISUFLOW [9] is a tool to help debug static analysis software. While it is great for debugging problems on small reproducers, it is not suitable to debug problems that only manifest on large projects, such as cut heuristics and exploration strategy related issues in symbolic execution. The same author conducted a survey with 115 analysis writers [8]. They concluded that the state-of-the-art tools were not sufficient to fulfil the needs of static analysis software authors. The participants of the survey identified graph visuals, access to the intermediate representation and intermediate result count as very important features, and our framework excels at visualizing intermediate counters (statistics) over a large corpus of test projects.

Using static analysis together with mining is not a new idea. Macedo et. al. [12] used the mining of malware and static analysis together to extract behavioral patterns aiming to identify malware. The difference from our work is that we are mining repositories in order to improve the quality of a static analysis tool.

Covrig [13] is a tool to run dynamic and static analysis on several projects and aggregate the results. It is supporting a different use-case than our tool. Its emphasis is on collecting metrics about the analyzed projects and not on collecting metrics about the analyzers.

Ray et. al. [14] used entropy as a measure for comparing static analysis findings in order to correct code. They found that search-based bug-fixing methods may benefit from using entropy both for fault-localization and for the searching for fixes. Our presented toolset might help conduct similar studies in the future for C family languages, as it supports comparing a patched and unpatched (or differently configured) version of the static analysis engine.

# 4 Conclusions

We find the traditional practice of static analysis tool testing cumbersome and insufficient. One of the greatest problems is that a fixed set of test projects might not stress the newly introduced code paths of the analysis engine. The other concern is reproducibility, which is not only essential for reviewers, but for any subsequent re-evaluation of the changes. As the analyzer evolves, some of its distinct parts interact with each other. Consequently, some of the changes that seemed sensible in the past might become irrational in the future. Having a record of experiments from the past facilitates the re-evaluation of those decisions in the light of new circum-

stances. Finally, the current practice of presenting the measurement results does not aid the interpretation of the raw data. Using an easier-to-digest representation of measurements would reduce the effort needed to evaluate the changes.

In order to mitigate these issues, we suggested a particular analysis workflow and developed a toolchain supporting the Clang Static Analyzer and Clang-Tidy. These tools not only help collect relevant candidate projects for testing, but also perform differential analysis on the test projects, and generate easy-to-interpret figures for reviewers. We also added a new line-based coverage measurement mechanism to the Clang Static Analyzer that improved the precision of differential testing.

## References

[1] Clang Static Analyzer, a source code analysis tool for C, C++, and Objective-C programs. URL: https://clang-analyzer.llvm.org/ (Retrieved: 23/03/2019).

[2] Clang-Tidy, a static analysis and code refactoring tool. URL: http://clang.llvm.org/extra/clang-tidy/ (Retrieved: 23/03/2019).

[3] CodeChecker, a defect database and viewer extension for Clang-Tidy and the Clang Static Analyzer. URL: https://github.com/Ericsson/codechecker (Retrieved: 23/03/2019).

[4] Conan, an open-source C/C++ package manager. URL: https://conan.io/ (Retrieved: 23/03/2019).

[5] SearchCode, a free source code search engine. URL: https://searchcode.com/ (Retrieved: 23/03/2019).

[6] Vcpkg, a C/C++ library manager for Windows, Linux, and MacOS. URL: https://docs.microsoft.com/en-us/cpp/vcpkg (Retrieved: 23/03/2019).

[7] Bhushan, Ram Chandra and Yadav, Dharmendra Kumar. Number of test cases required in achieving statement, branch and path coverage using 'gcov': An analysis. In *2017 the 7th International Workshop on Computer Science and Engineering*, pages 176–180, 2017. DOI: 10.18178/wcse.2017.06.031.

[8] Do, Lisa Nguyen Quang, Krüger, Stefan, Hill, Patrick, Ali, Karim, and Bodden, Eric. Debugging static analysis. *CoRR*, abs/1801.04894, 2018.

[9] Do, Lisa Nguyen Quang, Krüger, Stefan, Hill, Patrick, Ali, Karim, and Bodden, Eric. Visuflow: a debugging environment for static analyses. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 89–92. ACM, 2018. DOI: 10.1145/3183440.3183470.

[10] Hampapuram, Hari, Yang, Yue, and Das, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, September 2005. DOI: 10.1145/1108768.1108808.

[11] Lattner, Chris. LLVM and Clang: Next generation compiler technology. Lecture at BSD Conference 2008, 2008.

[12] Macedo, Hugo Daniel and Touili, Tayssir. Mining malware specifications through static reachability analysis. In Crampton, Jason, Jajodia, Sushil, and Mayes, Keith, editors, *Computer Security – ESORICS 2013*, pages 517–535, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-40203-6_29.

[13] Marinescu, Paul, Hosek, Petr, and Cadar, Cristian. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 93–104, New York, NY, USA, 2014. ACM. DOI: 10.1145/2610384.2610419.

[14] Ray, Baishakhi, Hellendoorn, Vincent, Godhane, Saheel, Tu, Zhaopeng, Bacchelli, Alberto, and Devanbu, Premkumar. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM. DOI: 10.1145/2884781.2884848.

[15] Regehr, John, Chen, Yang, Cuoq, Pascal, Eide, Eric, Ellison, Chucky, and Yang, Xuejun. Test-case reduction for c compiler bugs. In *ACM SIGPLAN Notices*, Volume 47, pages 335–346. ACM, 2012. DOI: 10.1145/2345156.2254104.

[16] Rigby, Peter C and Storey, Margaret-Anne. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550. IEEE, 2011. DOI: 10.1145/1985793.1985867.

Acta Cybernetica **25** (2022) 797–815.

# Type Inference of Simple Recursive Functions in Scala

Gergely Nagy[ab], Gábor Oláh[ac], and Zoltán Porkoláb[ad]

**Abstract**

Scala is a well-established multi-paradigm programming language known for its terseness that includes advanced type inference features. Unfortunately this type inferring algorithm does not support typing of recursive functions. This is both against the original design philosophies of Scala and puts an unnecessary burden on the programmer. In this paper we propose a method to compute the return types for simple recursive functions in Scala. We make a heuristic assumption on the return type based on the non-recursive execution branches and provide a proof of the correctness of this method. We implemented our method as an extension prototype in the Scala compiler and used it to successfully test our method on various examples. The algorithm does not have a significant effect on the compilation speed. The compiler extension prototype is available for further tests.

**Keywords:** Scala, type inference, recursion

## 1 Introduction

Scala is a well-established programming language providing both object-oriented and functional programming language elements. As a consequence, the language syntax needs to reflect both paradigms that results in a high level of expressiveness. Most of the new language properties are targeting the extensibility, safety and flexibility of the language. Examples for such features include advanced pattern matching, lambda expressions, by-name parameter passing and case classes. Improving the readability of the source code was a primary goal of language design; including the ability to avoid unnecessary boilerplate, repetitive code elements.

Terseness is important not only to make software code cleaner, but also to accentuate key parts of the solution expressed by the existing elements. Furthermore,

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: njeasus@caesar.elte.hu, ORCID: 0000-0002-0736-8903
[c]E-mail: olikas@caesar.elte.hu, ORCID: 0000-0001-5804-6448
[d]E-mail: gsd@caesar.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.282762

the more automatically computed information the compiler can provide, the less possibly erroneous code snippets the developer writes.

The Scala programming language is well-known for its terseness that includes advanced type inference features. These range from automatic deduction of variable types based on their initializations, to infer type parameters of generic functions based on call-side information.

Function return types are inferred for most trivial cases as well: the type of the lastly evaluated expression provides the return type of the function. In cases when multiple return statements are present the least upper bound ($LUB$) type of these will be used.

Since the subtyping relation ($<:$) is reflexive ($A <: A$), transitive ($A <: B \wedge B <: C \Rightarrow A <: C$) and antisymmetric ($A <: B \wedge B <: A \Rightarrow A = B$), it defines a *partial ordering* on the set of the types and thus this set has the *least upper bound* property. Scala supports all three kinds of variance, so it is up to the programmer to define how the type arguments affect the subtyping in case of generics.

However, if any of the lastly evaluated expressions in a function include a reference to the containing function, this algorithm cannot provide a meaningful result. This causes recursive functions without explicitly provided return types (such as on Figure 1) to fail the compilation process.

```
1  def factorial(n: Int) = n match {
2      case 0 => 1
3      case _ => n * fact(n-1)
4  }
```

Figure 1: A recursive function with no defined return type.

For any developer it is obvious that the `factorial` function will return an `Int`. Such similar simple recursive functions (see formal definition on Def. 3.1) occur frequently in most functional codebases. Not inferring their return type is both against the original design philosophies of Scala and puts an unnecessary burden on the programmer even in these simple cases.
Consider the example on Figure 2 that does not compile under current Scala type inference rules. In such situations the programmer may choose an unnecessarily wide type, such as the top type `Any`, corrupting the highly praised type system of Scala.

In this paper we propose a method to compute the return types for simple recursive functions. Similarly to the intentions of the developer, our heuristic assumption on the return type is based on the non-recursive execution branches. Assuming that the recursive functions will always end up in a non-recursive execution branch, we argue that the $LUB$ of these branches provides a sufficient return type for the function. If this assumption is not met, our method reports the same error as the current Scala compiler.

To create a prototype implementation we have extended the `typer` not to im-

```
1  abstract class Base
2  class Derived1 extends Base
3  class Derived2 extends Base
4  class Derived3 extends Base
5
6  def lousyType(d: Base, m: Double) = d match {
7      case _: Derived1 if m > 3 => new Derived3
8      case x: Derived2 if m < 2 => lousyType(x, m)
9  }
```

Figure 2: A recursive function with return types of a hierarchy.

mediately fail on recursive functions but use the proposed method to calculate the missing type. The extension does not have a significant effect on the compilation speed. We implemented our method as an extension prototype for the Scala compiler version 2.12.4 and used it to successfully test our method on various examples. The compiler extension prototype is publicly available for further tests.

This paper is structured as follows. In Section 2 we further evaluate the problem space with more examples and real-world issues. We provide the theoretical foundations of our mechanism in Section 3. We overview our results in Section 4 while also describing implementation details. Related works in Section 5 discusses similar problems in C++. Our paper concludes in Section 6.

## 2 Motivation

One of the main focuses of software development methodologies and practices nowadays is trying to lift off work and complexity of programmers as much as possible. This is achieved by various methods ranging from tooling and programming methods to language design. This trend is sensible with the spread of multiparadigm and especially functional languages which are usually tightly bound by their type systems. These type systems are mathematically proven to be correct and well-known algorithms were developed that can be used to prove if programs meet these bounds. The algorithms in question are built into compilers, so any programmer can easily check the correctness of their program. Being peer-reviewed and fully proven, one can trust that if the compiler finishes work on a piece of code it meets certain criteria. This should lead to fewer bugs and runtime errors in production software.

Scala is one of the more recent multiparadigm languages, that tries to solve a lot of complex problems before a developer meets them. The presence of this idea can be found in many of Scala's design goals, for example having as clean of a syntax and being as terse as possible [13]. This is achieved by introducing a fairly complex type inference system in the compiler, so programmers do not have to take time and effort to annotate their programs with types that can be deducted by an

algorithm.

The type inference algorithm of Scala is far from complete though. It does not support any recursion, let it either be a simple recursive function, a recursive chain or a recursive type declaration. This can be surprising and frustrating to anyone writing Scala code, furthermore it can lead to non-trivial issues in one's code and later, a software product.

In the following we show a few examples where the lack of type inference on recursive functions may lead to possible runtime application errors.

```
1  def map[C, A <: C, B <: C](y: Seq[A], f: A => C) /*: Seq[C]*/ = {
2      y match {
3          case Nil => Seq[B]()
4          case x :: xs => map(xs, f) :+ f(x)
5      }
6  }
```

Figure 3: A recursive map implementation with explicit type annotation.

In our first example we start off with a higher-order function, `map` with our own interpretation. In the version that can be seen on Fig. 3, we leverage generic types as well as functions as first-class entities in Scala. We would always like to get the minimum type of the collection from this function, hence we are providing information on the relationships of the types. Unfortunately the Scala compiler is not much of a help here: it will throw an error when we try to call `map` with the remainder list. This becomes even more annoying when we provide an incorrect return type: the compiler will be able to recognize the error at the place of concatenating the computed element to the list. One can spend minutes on trying to find the type that makes the type system satisfied by recompiling several times, but it would be much more convenient if the compiler were able to find it for us at the very beginning.

```
1  class Level1
2  case object Class1Level1 extends Level1
3  case object Class2Level1 extends Level1
4  class Level2 extends Level1
5  case object Class1Level2 extends Level2
6  case object Class2Level2 extends Level2
```

Figure 4: A multi-level class hierarchy.

Let us consider another recursive method that computes its result that has one of the types of a multi-level class hierarchy, as seen on Fig. 4. An example method using these types can be found on Fig. 5. The Scala compiler will fail to infer the correct return type `Level1`, and will require the developer to define it for the

```
1  def deepRec(n: Int): Level1 = {
2      if (n == 0) {
3          Class1Level2
4      }
5      else if (n == 1 || n == 2 || n == 3) {
6          n match {
7              case 3 => Class2Level2
8              case _ => {
9                  if (n != 1) {
10                     deepRec(n − 1)
11                 } else {
12                     Class2Level1
13                 }
14             }
15         }
16     } else if (n == 4) {
17         Class1Level2
18     } else {
19         Class1Level1
20     }
21 }
```

Figure 5: A recursive function using type hierarchy on Fig. 4. with explicit type annotations.

function explicitly. Determining `Level1` as the return type is trivial in this case as we have listed all the types involved near the function definition in one place, but recognizing it when for example, class definitions are scattered in a fairly large and complicated framework can be challenging and time consuming even for seasoned developers.

Unfortunately, there is a pretty easy shortcut to make compile errors disappear in this case: define the return type as `Any` (the base type of all classes in Scala), making the type system and the compiler temporarily happy. As we all know, marking objects by the widest type is simply neglecting the type system, thus we are not using one of the main services offered by Scala.

# 3 Theoretical foundations

In this section we present the details of the theoretical background for our solution. We focus only on typing recursive functions, thus we do not detail typing e.g. objects.

Scala is a statically typed language, thus type checks happen at compile time. Type declarations can be omitted in the source in many places, and the compiler

runs static type analysis to infer the types of variables, functions and other language elements. Scala compilation is designed as multi-staged process. In the first step an AST of the program is constructed. Our main focus in this paper, typing, is executed as the third phase. Upon successful type inference, the abstract syntax tree is enriched with type information in this stage. Scala's type system contains such features that are not compatible with the Hindley-Milner type inference algorithm so it relies on one-directional, context-unaware local type inference.

### 3.1 Related works on type systems

Most statically typed languages such as C++ or Java require explicit type declarations (for later advances in C++, see Section 5). Other statically typed languages like Haskell or ML use static type inference to calculate types for functions. Some unification-based type inference [11] can be used to calculate types for functions in these languages. Another unification-based type inference is Hindley-Milner method [5], supposing that the return type of a function has a well defined type. Scala on the other hand is less restrictive on return types, branching expressions like the `match` construct allow that the return values on different branches have different types [4]. (E.g. a function can return either an integer or a string. In this case the return type of the function will be the $LUB - type$ of integer and string which is the top type, `Any`.)

Dynamically typed languages like Erlang [2] also allow to return values of different types. These kinds of *polymorphic* return types are extensively used in Erlang. Since types are not first-class citizens, external tools were developed to check for discrepancies in software [6], incorporating a type system called *success typing* [7]. The major difference between success typing and other type inference algorithms is that the aim of success typing is not to prove the type correctness of the program, but rather to discover cases where there would most certainly be a type error at runtime. It uses least upper bound types to enable the constraint solving algorithm to reach a fixed point.

Success typing uses union types to express coupling between types that are not in subtype relation. It is very useful for languages like Erlang where types are not an integral part of the language. It has a major drawback though when both the input parameter and the return type of a function are union types. The connection between the input and output is not expressed by the inferred type, and that decreases the number of discoverable errors. A possible improvement can be the use of conditional types similarly to the work of Aiken et al.[1]. This soft-type system (also using union types) includes conditional types where the constraints between type variables are built into the type. This type is more accurate in the above sense of finding discrepancies, but the size and complexity of inferred types makes it comprehensible for humans. If the human-readable criterion is ignored then the precision of inferred types can be increased without the need to calculate a fixed point [10]. These types are also very complex but can be used for specific tasks, e.g. automatic test data generation.

Success typing inspired us to type recursive functions. Since Scala is statically

$$
\begin{array}{rcl}
e & ::= & x \mid c(e_1, \ldots, e_n) \mid e_1\, e_2 \mid f \mid \\
 & & \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 \mid \\
 & & \texttt{letrec}\ \ x = f\ \ \texttt{in}\ \ e \mid \\
 & & \texttt{case}\ \ e\ \ \texttt{of} \\
 & & \quad (p_1\ \ \texttt{if}\ \ g_1 \Rightarrow b_1); \\
 & & \quad \ldots; \\
 & & \quad (p_n\ \ \texttt{if}\ \ g_n \Rightarrow b_n) \\
 & & \texttt{end} \\
f & ::= & \lambda(x) \Rightarrow e \\
p & ::= & x \mid c(p_1, \ldots, p_n) \\
g & ::= & g_1\ \ \texttt{and}\ \ g_2 \mid g_1\ \ \texttt{or}\ \ g_2 \mid x_1 = x_2 \mid \texttt{true} \mid e\, x
\end{array}
$$

Figure 6: The $\lambda_s$ language.

typed and types are inserted into the AST, using union types is not suitable for our needs. Unions would introduce new types to our program, that we do not intend to do since it would be hidden to the programmer and might cause unforeseen errors. Instead, we use the type hierarchy already present in the language. Scala already has a solid type system for nested classes, abstract types, path dependent types, etc [4, 9]. Since type inferring is solidly working in Scala, we do not want to replace or improve these theories.

## 3.2 The $\lambda_s$ Language

We propose a small language and the corresponding calculus to demonstrate the theoretical soundness of our approach to type simple recursive functions. Let us call the language $\lambda_S$ and be defined in Fig. 6. We would like to emphasize that this small language is not intended to be either generic-purpose or a full representation of Scala, rather to be the minimal language that can help us describe our proposed method formally.

The language contains variables ($x$) that are immutable. Data constructors ($c$) can be used to construct any kind of data, including constants, objects, etc. $\lambda_S$ contains only single-argument function application. We assume that all Scala functions with at least one parameter can be curried, that is, they can be transformed to a function with multiple parameter lists containing only one parameter. It contains the standard polymorphic *let* expression. The recursive *let* expression has only one function component ($x = f$) since in this paper we deal with only self-recursive function. We define a branching expression (`case`). In the head of the case expression, $e$ is matched against the patterns ($p$) sequentially. The first matched pattern will invoke the evaluation of the body ($b$) for the pattern. If no patterns match,

then an exception is raised. Each branch has a pattern and a guard. A pattern can be a variable or a construct of patterns. Guards, that are always present in the syntax, can be type check or other value checks. Using `true` as a guard, we can express the the case when we actually do not need any guards.

Our main focus will be on combining recursive *let* and *case* expressions. Recursive functions can be typed if they have a branch that is not recursive. Having this branch fulfills the termination criteria. If a function does not contain any terminating branches, then the function is divergent, and it cannot be typed.

$$
\begin{aligned}
E \quad ::= \quad & \texttt{letrec}\ x = \lambda(a) \Rightarrow \\
& \quad \texttt{case}\ a\ \texttt{of} \\
& \quad\quad (p_1\ \texttt{if}\ g_1 \Rightarrow b_1); \\
& \quad\quad \ldots; \\
& \quad\quad (p_n\ \texttt{if}\ g_n \Rightarrow b_n) \\
& \quad \texttt{end in}\ e
\end{aligned}
$$

Figure 7: The syntax of simple recursive functions

**Definition 3.1** (Simple recursive function)**.** The expression $E$ in Fig. 7 is considered a *simple recursive function*, iff there exists $i \in [1..n]$ that $b_i$ symbolically contains $x$ and there exists $j \in [1..n]$ that $b_j$ does not contain $x$.

Simple recursive functions do exist and provide an abstract pattern over recursive functions that are not in recursive call chain.

## 3.3 Derivation rules

We provide type derivation rules for the syntactic constructs of $\lambda_S$. We assume that the type of objects, member functions and other language constructs not covered in this paper can be computed.

We present type derivation rules (Fig. 8) in the following form of statements: $\Gamma \vdash e : \tau$, read as "supposing $\Gamma$ the type of the expression $e$ is $\tau$". $\Gamma$ is the *context* of mappings from variables to types. The $\cup$ operator is used to denote that a particular mapping is present in the context.

The derivation rules describe a standard way to type our language. A variable can be typed (Rule (Var)), if its type is present in the variable context. The type of a data constructor (Rule (Cons)) is composed of the types of the components. A type is considered a subtype of another type (Rule(Sub) if an expression of the subtype can also be typed to the wider type. A function application can be typed if the argument expression ($e_2$) is a subtype of the parameter type ($\tau_1$) of arrow type. In our case the arrow type is calculated via the existing type inferring algorithm of Scala. The type of a function (Rule (Fun)) and the let (Rule (Let)) expression

$$\frac{}{\Gamma \cup x : \tau \vdash x : \tau} \quad \text{(Var)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \qquad (\forall i \in [1..n])}{\Gamma \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)} \quad \text{(Cons)}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \quad \text{(Sub)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau' \quad \tau' <: \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \quad \text{(Appl)}$$

$$\frac{\Gamma \cup x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x) \Rightarrow e : \tau_1 \to \tau_2} \quad \text{(Fun)}$$

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1' \to \tau_2 <: \tau_1 \to \tau_2'} \quad \text{(S-Fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2} \quad \text{(Let)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{x : \tau_x | x \in Var(p_i)\} \vdash \\ \quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ (\forall i \in [1..n]) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash\ \ \texttt{case}\ \ : \bigsqcup_{i=1}^{n} \tau_{b_i}} \quad \text{(Case)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{x : \tau_x | x \in Var(p_i)\} \vdash \\ \quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ (\forall i \in [1..n]) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash\ \ \texttt{case}\ \ : \bigsqcup_{i=1}^{n} FIX\tau_{b_i}} \quad \text{(Letrec)}$$

Figure 8: Derivation rules for $\lambda_S$.

follow standard definitions. With the derivation rule of subtyping of functions (Rule (S-Fun)) we would like to express that it is safe to allow a function of one type $\tau_1' \to \tau_2$ to be used in a context where another type $\tau_1 \to \tau_2'$ is expected as long as none of the arguments that may be passed to the function in this context will surprise it ($\tau_1 <: \tau_1'$) and none of the results that it returns will surprise the context ($\tau_2 <: \tau_2'$).

To type a `case` expression, first we type the head expression ($e$, which has the form defined in the `case` branch on Figure 6). For each branch we extend the context with types for free variables of patterns ($Var$) and calculate types for the patterns and the body of the function. The guards must evaluate to the boolean type. We denote the least upper bound type by the operator $\tau_1 \sqcup \tau_2$. The head of the expression has to be the subtype of the $LUB$ of the types of the patterns ($\tau_{p_i}$). The return type is the $LUB$ of the types of the bodies of the branches.

Letrec(Rule (Letrec-C)) is similar to `case`, but it uses the fixed point of the return types of the branches ($FIXf = f(FIXf)$).

The recursive let expression can be typed in our scope only if it consists of a simple recursion function. We provide the constructive algorithm in the following theorem:

**Theorem 3.2** (Constructive derivation rule of `letrec`). *Suppose the notation of Fig. 7. Let us denote* $\mathcal{J} := \{i \mid b_i$ *does not contain* $x\}$. *Then the following constructive derivation rule holds:*

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{y : \tau_y | y \in Var(p_i)\} \vdash \\ b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ (\forall i \in \mathcal{J}) \\ \Gamma \cup \{y : \tau_y | y \in Var(p_k)\} \cup \{x : \bigsqcup_{i=1}^{n} \tau_{p_i} \to \bigsqcup_{i=1}^{n} \tau_{b_i}\} \vdash \\ b_k : \tau_{b_k}, p_k : \tau_{p_k}, g_k : \tau^{bool}\ (k \in [1..n] \setminus \mathcal{J}) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash E : \bigsqcup_{i=1}^{n} \tau_{b_i}} \quad \text{(Letrec-C)}$$

*where* $E$ *is the* `letrec` *expression defined on Figure 7.*

*Proof.* Let us first divide the case expression into two parts: the ones that do not contain recursive calls and the others that do. For the non-recursive branches we use the regular case typing derivation rule, hence $\Gamma \cup \{y : \tau_y | y \in Var(p_i)\} \vdash b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ \ (\forall i \in \mathcal{J})$ holds. This expression has the type of $\bigsqcup_{\forall i \in \mathcal{J}} \tau_{b_i}$ as per the case derivation rule. Let us later refer to this as the non-recursive type.

For the recursive branches, we have two cases:

1. Tail-recursion, as in $b_k \equiv x\, b_k'$. The expression in this case holds the type of the previously mentioned non-recursive type. We extend the type context

with $x : \tau_{p_k} \to \tau_{b_k}$ where $\tau_{p_k} <: \bigsqcup_{i=1}^{n} \tau_{p_i}$ and $\tau_{b_k} \equiv \bigsqcup_{\forall i \in \mathcal{J}} \tau_{b_i}$, i.e. not changing the non-recursive type.

2. Non-tail recursion. We type the body by applying the intermediate type that has been calculated so far to the recursive expression, then calculate $x : \tau_{p_k} \to \tau_{b_k}$. This will be then added to the type context thus the intermediate type of the expression will be extended by $\tau_{b_k}$ to $\bigsqcup_{i \in \mathcal{J}} \tau_{b_i} \sqcup \tau_{b_k}$.

With the above considerations we can type all branches of `letrec`, turning it into a regular `case` expression that has the type of $\bigsqcup_{i=1}^{n} \tau_{b_i}$, resulting in the following type: `letrec` $: \bigsqcup_{i=1}^{n} \tau_{b_i}$.

□

# 4 Results

In this section we will discuss how the previously described algorithm works in practice. First we apply the algorithm to the two examples shown in section 2 then we will show how this is implemented as an extension in the Scala compiler.

## 4.1 Typing simple recursive functions

Our first example is an unusual version of the map function on Fig. 9. Its purpose is to return the mapped results in a list of the smallest type possible. As one can see in the listing, the main body of the function is a `match` with two possible `case` branches. This instruction flow is very similar to the extended $\lambda_S$ language. Each branch has a return type, namely the first one returns a `Seq[B]` and the second one a `Seq[C]`. The type of the second case is defined by the result of the concatenation(`:+`) method. This part can be rewritten in a form of `map(xs, f).:+(f(x))`. If we consider this function call, the appended element is of type `C`, as we have declared `f` to be a function of `A => C`. Since `A` is a subtype of `C`, the result of the map function is a subtype of `C`, making the result of the append call be a type of `Seq[C]`. This leads us to two calculated types for the branches: `Seq[B]` and `Seq[C]`. The typing mechanism we propose would now calculate the least upper bound for these types that would be `Seq[C]`, and typing our special `map` function with `Seq[C]`.

The next example is more involved. First we start off with declaring a multi-level type hierarchy as seen on Fig. 4. This hierarchy declares 3 levels with leaf nodes being `case classes`. The recursive function using these classes is defined on Fig. 10. The control flow graph created by the predicates in the function has several branches, unlike the straight tree of the `match` and `cases` in the first example. This will cause no problems to our typing algorithm, as it can be applied to all leaf branches, and then going upwards in the tree using the previously calculated types. This calculation starts with the `if-else` on line 9. The first branch contains a recursive call, so we cannot type this branch. We need to start with the `else`

```
1  def map[C, A <: C, B <: C](y: Seq[A], f: A => C) = {
2      y match {
3          case Nil => Seq[B]()
4          case x :: xs => map(xs, f) :+ f(x)
5      }
6  }
```

Figure 9: A recursive map implementation with type inference.

branch first, making the calculated type `Class2Level1`. The `case` branch on line 8 then would be typed the same. We have a trivially typed `case` on line 7, with `Class2Level2`. The least upper bound for these types is `Level2`, so the `else` branch will be typed `Level2`. We have arrived at the top-most level of predicates, that has types of `Class1Level2`, `Level2`, `Class1Level2` and `Class1Level1`, in respective order. The least upper bound defined by the hierarchy is therefore `Level1`. This will be the final type of the `deepRec` function.

```
1  def deepRec(n: Int) = {
2      if (n == 0) {
3          Class1Level2
4      } else if (n == 1 || n == 2 || n == 3) {
5          n match {
6              case 3 => Class2Level2
7              case _ => {
8                  if (n != 1) {
9                      deepRec(n − 1)
10                 } else {
11                     Class2Level1
12                 }
13             }
14         }
15     } else if (n == 4) {
16         Class1Level2
17     } else {
18         Class1Level1
19     }
20 }
```

Figure 10: A recursive function using type hierarchy on Fig. 4. with type inference.

The $\lambda$ language was only defined for single-parameter functions. The reason we can still use it to calculate types for these functions is that by currying multi-parameter functions, we can always transform them to a chain of function appli-

cations having only one parameter. Since Scala supports currying by default – by denoting a function call with the _ (underscore symbol) –, we assume that all functions in question can be transformed to this kind.

## 4.2 Typing recursive functions with multiple branches

In the previous examples we discussed functions with a single recursive branch. In the following we show that similar solution exists for simple functions with multiple recursive branches.

```
1  class A {
2      def toC: C = ...
3      def toD: D = ...
4  }
5
6  class B extends A
7  class C extends A
8  class D extends A
9
10 def multi(n: Int) = n match {
11     case 0 => new B
12     case n > 0 => foo(n − 1).toC
13     case n < 0 => foo(n + 1).toD
14 }
```

Figure 11: A function with multiple recursive branches.

In the example shown on Fig. 11 there are two recursive branches resulting in two different but related types. Our algorithm finds the non-recursive branch on line 11 and calculates type `B`. Typing the second branch will use this information as the return type of the `foo` call. Using `B` as a placeholder type, the default type inference algorithm of Scala will calculate type `C` as the return type of the branch on line 12. Similarly, type `D` will be calculated for the branch on line 13. Finally, the *LUB* of types `B`, `C` and `D` will be determined as `A`. Therefore, the return type of function `multi` will be `A`. This result complies with the expected result type of our algorithm and meets the intention of the developer.

## 4.3 Extending the Scala compiler to handle simple recursive functions

The fundamental design and structure of the Scala compiler makes it an excellent candidate to be extended. The features of the compiler that makes this possible are high separation of compiler phases, the support for macros and fully independent compiler plugins and the compiler being an open source project[14]. The phases of

the compiler start by parsing the source files and generating an AST; later phases transform this AST. The current version of compiler is written in Scala, using Scala objects to describe the nodes of the AST. The compiler also acts as a library to analyze and compile Scala source code, providing a programmatic API that can be accessed from applications.

As the first step, we had to find a way to circumvent the type error generated for recursive functions. In the default version of the compiler, when the typer starts calculating the type of a (recursive) function and it meets an entity that has been defined previously, but the type of it is yet to be determined, it will throw a `CyclicReference` exception that is collected and handled by the generic error handling infrastructure of the compiler. This does not stop the typing phase from continuing with typing other entities. We leverage this property, as it collects all the erroneous recursive calls, but types all other to-us trivial cases.

Our main approach of extending the compiler has focused on creating a separate codebase, as modifying the main branch directly was found to be too time consuming and difficult. Fortunately, we were helped in this effort by the various API calls provided by the package `scala.tools.nsc._` (`nsc` stands for New Scala Compiler).

The extended `scala.tools.nsc.typechecker.Analyzer` contains methods overridden that handle control flow ASTs –namely `if`s and `case`s– and method definitions, so we can annotate these methods with our calculated type. We of course use the original typer to first type these entities and only interrupt cases that are of interest to us. Finding the least upper bound of types is another key point in our algorithm, but we were very fortunate in this regard: we use the function `lub` on `scala.tools.nsc.TypeChecker.Typer`.

The last remaining piece to have a working compiler was inserting the newly created typer into the chain of compile phases and invoking it from the first step, the parser. We have achieved this with our own entry point to an application that simply passes a path to the compiler and invokes it. We only use regular console reporting by `scala.tools.nsc.reporters.ConsoleReporter` and global settings by `scala.tools.nsc.{Global, Settings}`.

We have measured how our extension affects compile speeds by calculating the total time spent in the `main` method of our application. The results are shown in Table 1. For simplicity, we have listed measured microseconds with the default version of the compiler –i.e. invoking it without setting the extended typer– and with the extension in place. As it can be seen in change percentage, the extension has no significant impact on performance. Example1 and Example2 refer to the examples seen in the previous subsection, while Multiple methods contains several other test cases.

The prototype can be downloaded and can be used for further tests from [15].

## 4.4 Restrictions

Scala supports defining recursive types using explicit type annotations. Soundly calculating all recursive types would require extending our inference method with a

Table 1: Compile times ($\mu$s) with and without using our extension

| Code snippets | Default | Extended | % change |
|---|---|---|---|
| Example1 | 2440166 | 2479896 | 101.62 |
| Example2 | 3390745 | 3422435 | 100.93 |
| Multiple methods | 5691124 | 5712335 | 100.37 |

fixed point calculation. Henceforth, our proposed method in its current form does not support recursive types.

We implemented and tested our compiler extension only on Scala compiler version 2.12.4. It is not guaranteed to work with any other version than 2.12.4.

### 4.5 Future work

When we created the Scala compiler extension to infer types of simple recursive functions, our main intentions were focused on prototyping the theoretical background we have described in this paper, not developing an industry-standard, complete implementation. This leaves great space for future improvements. Firstly, we can provide a better integration to the compiler by disabling `CyclicReference` exceptions for our cases, then properly type AST nodes "in-place". Then we can merge our changes back to the main line of the compiler.

Besides creating a more robust implementation, we also plan to work on extending the theory by finding methods to analyze recursive chains then proving the soundness of these methods. This would require extending our $\lambda$-language and also the way it handles types. As another step, we are looking into providing a clean, correct and complete theoretical background and implementation for inferring types of recursive type definitions.

## 5 Related works: type inference in C++

The C++ programming language is one of the current mainstream general purpose languages [12]. Its popularity is originated to its suitability in almost all application areas from high performance computing and telecommunication to embedded systems. C++ provides language tools for the programmer to implement complex systems from gradually specified and implemented building blocks without compromising run-time efficiency. C++ is often described as a multiparadigm programming language [3], as it has imperative, object-oriented, generic and functional programming features.

C++ is a strongly typed programming language in the sense, that the type of every (sub)expression is determined in compilation time. However, templates use duck-typing,i.e. no constrained generics exist in current C++. There are plans to improve the template mechanism with constraints.

Earlier C++ codebase was known about notoriously long type notations. To unburden programmers' task and make source more readable, the C++11 standard introduced the `auto` keyword as a placeholder for types [16]. Its primary usage is to avoid needlessly verbose type declarations, like those are used with connection in STL algorithms:

```
1  // C++03
2  typename std::vector<T>::iterator i = v.begin();
```

This can be replaced by usage of keyword `auto`. The type of the `i` variable will be inferred from the initialization expression: `v.begin()`

```
1  // C++11
2  auto i = v.begin()
```

The keyword `auto` to replace the return type for functions but only with a new trailing type syntax introduced in C++11:

```
1  template <typename T, typename S>
2  auto max( T a, S b) -> decltype(a+b) // C++11
3  {
4      if ( a > b )
5      return a;
6      else
7      return b;
8  }
```

Notice, that this usage of `auto` syntax does not imply type inference, the return type is explicitly expressed in the trailing syntax. The role of `auto` here is only a placeholder: since the language elements used in the trailing syntax (`a` and `b` parameters in the `decltype` expression are not in scope *before* the function name).

In the C++14 standard, however, there is automatic type inference available for function return types in the most simple cases [8].

```
1  auto f(); // return type is unknown
2  auto f() // return type is int
3  {
4      return 42;
5  }
6  auto f(); // redeclaration
7  int f(); // error, declares a different function
```

A function with auto return type can have multiply return statements. However, there is a strict restriction here, that each return statement should return the same single type, otherwise the compiler reports error. That is different to Scala, where in case of multiple return statements the return type is inferred as the least upper bound of the return types.

Recursion is allowed by the proposed C++14 inference rules in a very restricted way. The recursive return branch should be preceded by at least one non-recursive

return, from which the return type of the function is inferred. Subsequent return statements are checked against this type. Therefore the following code will be accepted by the proposal:

```
1  auto fib(int n)
2  {
3      if ( 0 == n ) return 1;
4      else return n*fib(n-1);
5  }
```

While the variation, where we have changed the recursive and non-recursive branches will be rejected:

```
1  auto fib(int n)
2  {
3      if ( n > 0 ) return n*fib(n-1);
4      else return 1;
5  }
```

The authors have the opinion that these rules are unnecessary restrictive and can be relaxed without compromising compile-time efficiency.

# 6 Conclusions

In this paper we have analyzed Scala inference rules for function return types. We stated that with certain types of simple recursive functions, automatic calculation of the return type can be done with some effort. Such an additional feature is in parallel with the original design philosophies of Scala that try to lift unnecessary burden off the programmer.

We have proposed a new method to compute the return types for simple recursive functions. We have defined a small language to demonstrate the theoretical soundness of our approach. Our heuristic assumption on the return type is based on the non-recursive execution branches and we have also provided a proof of its correctness. Furthermore, we have assumed that the recursive functions will always end up in a non-recursive execution branch. The least upper bound type of these branches provide sufficient information allows the default Scala type inference algorithm to infer the return type of recursive branches. Finally, by taking the least upper bound type of all branches we can define the return type of the function.

A prototype implementation has been created by extending the Typer not to immediately fail on recursive functions, but use the proposed method to calculate the return type. We have implemented our method as an extension prototype for the Scala compiler v2.12.4 and have used it to successfully test our method on various examples. In case type discrepancies already exist in the program, our compiler extension will report the same error as the current Scala compiler.

The extension is proved to be effective in the sense that it does not significantly affect compilation speed. The compiler extension prototype is publicly available for further tests.

## References

[1] Aiken, Alexander, Wimmers, Edward L, and Lakshman, TK. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1994. DOI: 10.1145/174675.177847.

[2] Armstrong, Joe, Virding, Robert, Wikström, Claes, and Williams, Mike. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[3] Coplien, James O. *Multi-paradigm design for C++*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[4] Cremet, Vincent, Garillot, François, Lenglet, Sergueï, and Odersky, Martin. A core calculus for Scala type checking. In *International Symposium on Mathematical Foundations of Computer Science*, pages 1–23. Springer, 2006. DOI: 10.1007/11821069_1.

[5] Hindley, Roger. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969. DOI: 10.1090/S0002-9947-1969-0253905-6.

[6] Lindahl, Tobias and Sagonas, Konstantinos. Typer: A type annotator of Erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25, 2005. DOI: 10.1145/1088361.1088366.

[7] Lindahl, Tobias and Sagonas, Konstantinos. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178, 2006. DOI: 10.1145/1140335.1140356.

[8] Merill, J. Return type deduction for normal functions, 2013. Revision 5. N3638, 2013.04.17. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/~2013/n3638.html.

[9] Odersky, Martin, Cremet, Vincent, Röckl, Christine, and Zenger, Matthias. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming*, pages 201–224. Springer, 2003. DOI: 10.1007/978-3-540-45070-2_10.

[10] Oláh, G., Horpácsi, D., Kozsik, T., and Tóth, M. Type interface for core Erlang to support test data generation. *Studia Universitatis Babes-Bolyai Informatica*, LIX(2014/1):201–215, 2014.

[11] Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, and Washburn, Geoffrey. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices*, 41(9):50–61, 2006. DOI: 10.1145/1159803.1159811.

[12] Stroustrup, Bjarne. The C++ programming language (special 3rd edition), 2000.

[13] Venners, Bill and Sommers, Frank. The goals of Scala's design. A conversation with Martin Odersky, Part II. In Artima developer's web site, 2009. http://www.artima.com/scalazine/articles/goals_of_scala.html.

[14] The GitHub home of the Scala compiler. https://github.com/scala/scala.

[15] The GitHub home of the Scala compiler extension for simple recursive functions. https://github.com/njeasus/ScalaRecTyper.

[16] The ISO C++11 standard, ISO/IEC 14882:2011(E) – Information technology – Programming languages – C++, 2011. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372.

Acta Cybernetica **25** (2022) 817–846.

# Adaptation of a Refactoring DSL for the Object-Oriented Paradigm*

Dávid J. Németh[ab], Dániel Horpácsi[acd], and Máté Tejfel[ae]

**Abstract**

Many development environments offer refactorings to improve specific properties of software, but we have no guarantees that these transformations indeed preserve the functionality of the source code they are applied on. An existing domain-specific language, currently specialized for Erlang, makes it possible to formalize automatically verifiable refactorings via instantiating predefined transformation schemes with conditional term rewrite rules.

We present a proposal for adapting this language from the functional to the object-oriented programming paradigm, using Java in place of Erlang as a representative. The behavior-preserving property of discussed refactorings is characterized with a multilayered definition of equivalence for Java programs, including the conformity relation of class hierarchies. Based on the decomposition of a complex refactoring rule, we show how new transformation schemes can be identified, along with modifications and extensions of the description language required to accommodate them. Finally, we formally define the chosen base refactoring as a composition of scheme instances.

**Keywords:** verifiable refactoring, scheme-based refactoring, microrefactoring, program equivalence

## 1 Introduction

Software development in practice is usually an iterative process. That is, the end product is not the result of a single step, instead it is constructed by iteratively

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]ELTE Eötvös Loránd University, Budapest, Hungary, and Faculty of Informatics, 3in Research Group, Martonvásár, Hungary

[b]E-mail: ndj@inf.elte.hu, ORCID: 0000-0002-1503-812X

[c]E-mail: daniel-h@elte.hu, ORCID: 0000-0003-0261-0091

[d]Project no. ED18-1-2019-0030 (Application Domain Specific Highly Reliable IT Solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

[e]E-mail: matej@inf.elte.hu, ORCID: 0000-0001-8982-1398

DOI: 10.14232/actacyb.284280

refining an initial prototype. The longest phase in the development lifecycle, maintenance [17], also requires the modification and extension of existing code. Changes made between two iterations can be seen as source-level transformations.

If such a modification preserves the functional behavior of software, it is called a refactoring. These behavior-preserving transformations are mainly used to improve non-functional properties (e.g. maintainability) without altering the meaning of the program [6]. Many development environments offer refactorings, but we have no guarantees that these transformations indeed preserve the functionality of the source code they are applied on. Moreover, in case of safety-critical systems, formal verification of refactorings can also be deemed desirable.

An existing domain-specific language, explained in Section 3, makes it possible to formalize automatically verifiable refactorings via instantiating predefined transformation schemes with conditional term rewrite rules [9, 8]. This language was constructed for refactorings mainly on Erlang and the functional programming paradigm. The goal of our research was to investigate whether this method could be applied to other paradigms and languages. Since one of today's most popular paradigm is OOP, we have chosen it as our target, and selected Java as its representative due to its high-level nature and widespread support.

The main contributions of this paper are the following:

- A multilayered definition of equivalence for Java programs, used to characterize the behavior-preserving property of discussed refactorings.
- A case study which shows how new transformation schemes can be identified from a complex refactoring rule, along with modifications and extensions of the description language required to accommodate them.
- Semantic functions and predicates related to abstractions of the target paradigm, which we use to describe transformation preconditions.

The rest of the paper is structured as follows. As an introduction, Section 2 presents the foundations of our work and also summarizes results related to it. Section 3 gives a brief overview of the adapted refactoring language. In Section 4 we discuss general decision concerns of the adaptation process, including the choice of target language; refinement of equivalence; and methods to synthesize new refactoring schemes. Section 5 presents a case study where we identify new schemes based on the decomposition of a complex refactoring rule; modify and extend the description language to accommodate the new schemes; formally define the chosen base refactoring as a composition of scheme instances; and provide a step-by-step example of their application to a concrete program. Finally, Section 6 lists further research directions and Section 7 concludes.

## 2 Foundations and Related Work

In this section, we give an overview on our method's foundations by citing publications that described them, and also present related work on approaches aimed

at specifying and/or verifying refactorings. Please note that the direct base of this paper, the scheme-based methodology, is discussed separately in Section 3.

## 2.1 Foundations

**Strategic term rewriting.** In his dissertation [10], Kalleberg discusses language agnostic methods for source code analysis and transformation using data and function abstraction techniques. From the latter he emphasizes *strategic term rewriting*, which serves as a basis for program transformations defined with the combination of traversal strategies and rewrite rules. He mentions *System S* [20] as an underlying formal model, and also a possible implementation in the form of the *Stratego* [3] language.

**Microrefactorings.** The concept of *microrefactorings*, grounded by Opdyke in his fundamental work [13], is widely incorporated to discussions addressing verified program transformations. The main idea of this method is to decompose complex refactorings in order to obtain small, atomic transformations which are easier to write, understand and verify. Then, the original refactoring can be reconstructed using these microsteps as its building blocks, potentially resulting in a pre-verified transformation. Note that in this case, the composite correctness is a consequence of the microsteps being refactorings themselves.

**Object-oriented refactorings.** In his previously cited publication [13], Opdyke also presents *refactorings characteristic of object-oriented systems*. Based on the method of microsteps, he gives decompositions for three complex transformations, discussing both breakdown strategy and target language metatheory in detail. He also provides informal correctness proofs along with the described refactorings.

**Characterization of program equivalence.** Schäfer et al. enhance the concept of microrefactorings in two novel aspects [15]. In order to eliminate the need for complex and fragile preconditions, they chose to dynamically check whether the application of a transformation results in an equivalent program. In the cited paper *equivalence is characterized* with the preservation of data flow, control flow and binding. Additionally, intermediate steps in a composite refactoring are allowed to consume and produce code over an extension of the target language, potentially increasing the expressiveness of transformations.

## 2.2 Related Work

Verbaere et al. present a scripting language for refactoring in [19]. It is a hybrid-paradigm DSL with a functional part for defining transformations and sublanguages based on logic and path queries to describe complex relations between program elements. Compared to our approach, transformations are expressed not declaratively with syntactic patterns, but with imperative commands to modify code at the level

of its internal representation. In addition, neither scheme-like language elements, nor verifiability is addressed in the work.

In [11], Leitão proposes a pattern language for refactoring Lisp programs. Like ours, it is a high-level DSL utilizing code patterns with metavariables, therefore transformations are specifiable in it without the knowledge of internal representations. The DSL itself is expressed within Lisp, resulting in an embedding that makes its language elements more easily executable, but at the cost of them containing more syntactic noise. Again, possible transformations are not outlined with generalized strategies, and verifiability is not discussed.

Li and Thompson describe the refactoring DSL of Wrangler, a tool for the interactive and extensible analysis and transformation of Erlang programs in [12]. Like us, they also distinguish primitive and composite refactorings, providing two high-level sublanguages with templates (code patterns) and combinators. The predefined strategies they offer for composite transformations make their descriptions more concise, but they do not apply this methodology to primitive refactorings. Moreover, as the proposed DSL follows the syntax of Erlang closely, definitions contain much syntactic noise.

In [16], Schäfer and de Moor present a high-level yet precise specification language for refactoring definitions. As already mentioned while discussing [15] in the previous section, they still aim for dynamic correctness guarantees instead of relying on overly complex preconditions. The language itself is built upon the abstractions of the target language, but its definitions are still imperative and lack syntactic patterns as well as general strategies.

Garrido and Meseguer present a mathematically rigorous framework for the formal specification and implementation of Java refactorings in [7], which they demonstrate by several verified refactorings. Although the proposed language contains reusable constructs akin to transformation schemes, in general it is defined on a lower level of abstraction than ours: refactorings are specified imperatively in the realm of the underlying formal semantics.

That is, neither of the above-mentioned related works offer a standalone refactoring language that enables users to describe executable transformations declaratively by high-level code patterns, while aiding usability and potential verifiability with general refactoring schemes for pragmatically composable microrefactorings.

## 3 Scheme-Based Refactoring

The basis of our work is an existing domain-specific language which makes it possible to define executable and verifiable refactorings using syntactic code patterns over the to-be-refactored language [9]. This allows to specify transformations without knowing any internal representations. The main idea of this existing method is to provide pre-verified refactoring skeletons, called schemes, which can be instantiated with conditional term rewrite rules, resulting in composable microrefactorings that serve as building blocks for complex transformations. In the following we give a brief overview of its description language and the verification technique it uses.

## 3.1 Description Language

The core of the description language is conditional term rewriting – a powerful tool for specifying program transformations based on syntactic patterns. As an illustration of this existing description language, we present a rewrite rule embedded in it. Note how the example resembles Erlang, the original target language of the method we aim to adapt to Java, and OOP in general.

```
    [#Head | #Tail]
    ------------------------
    #X = #Head, [#X | #Tail]
when
    fresh(#X)
```

In this example, the part before the **when** keyword defines the actual transformation in the form of a matching (above the line) and a replacement (below the line) pattern. During pattern matching, corresponding code segments are assigned to matching metavariables (indicated by a hashmark-prefix in the example). The second part (after the **when** keyword) specifies the precondition of the transformation, that is rewriting takes place only if the precondition holds.

The problem with term rewriting, however, is that it is a low-level approach which makes definitions of complex refactorings complicated and error prone, especially in the case of extensive transformations with many compensational modifications (e.g. a refactoring renaming a function has to modify the original call sites as well). To make refactoring definitions safer and even verifiable, the discussed method restricts the set of possible transformations by introducing high-level, reusable refactoring schemes. The provided schemes already contain the necessary control logic, and only have to be parameterized by term rewrite rules to yield concrete microrefactorings.

```
function signature refactoring swapFirstTwoParameters()
    #F(#A, #B, #Ps..)
    -----------------
    #F(#B, #A, #Ps..)
```

The example above, also taken from the original, to-be-adapted language, presents an instance of one of its schemes, namely *function signature refactoring*. The resulting refactoring swaps the first two parameters of the selected function not only in its definition but in its applications as well.

Microrefactorings defined as scheme instances can then be composed to obtain more complex transformations. In the following example, we show how two refactorings can be applied sequentially.

```
composite refactoring f()
do
    g()
    function().h()
```

Note that in addition to (here not explicitly present) combinators controlling the order of application, selectors are also provided to dynamically change the target to be modified. For example, here `h` is executed on the enclosing function of the original target.

### 3.2 Verification

In this section, we briefly specify the ideas behind, and requirements of, the to-be-adapted method's verification process, which we need to take into account during the adaptation. For a more detailed description, please refer to the original publication [9].

By restricting the set of specifiable transformations, automatic verification becomes feasible. Naturally, the verified property in this case is behavior-preservation with regards to an appropriately constructed definition of equivalence. The chosen formal model is the operational semantics of the target language, which makes it possible to mathematically reason about the execution of programs, e.g. by symbolically computing the possible outputs and side effects of a given function.

The verification process is two-fold. At first, the provided refactoring schemes are manually verified based on assertions concerning their rewrite rule parameters, collectively called the contract of the scheme. Then, scheme instances are examined whether the concrete rewrite rules used in them satisfy the contract of the instantiated scheme. Given schemes are appropriately identified, conformity to contracts becomes automatically verifiable.

Generally, in order to achieve this, contracts should demand no more than equivalences of specific rewrite rule patterns. The reason behind is that in this case, the formal method presented by Ciobaca et al. [4] can be applied to carry out the verification automatically with the correct tooling. The cited work reduces equivalence to the correctness property of a uniquely constructed, aggregated program which becomes verifiable with the formal proof system discussed by Stefanescu et al. [18]. The basis of this method is the operational semantics of the target language, which is embedded into reachability logic [14]. The proposed proof system is sound, but not necessarily complete, however, as neither the to-be-adapted refactoring language, nor our adaptation aims for completeness, the soundness of the verification backend can be considered adequate in both cases.

## 4 Adapting the Framework

Even though aiming for language independence, the refactoring framework briefly introduced in Section 3 [8] was developed having Erlang as its target language. The main motivation behind this paper is to recreate the existing framework for a significantly different target language, ultimately achieving another step towards making it more language-agnostic. In the following sections we discuss general aspects of the adaptation process.

### 4.1 Choosing the Target Language

Erlang is a functional and dynamically typed programming language. While selecting the alternative target language, our main concern was to choose a candidate belonging to another paradigm. In this way, we can possibly reason about how the framework should be adapted not only to a different language, but also to a different paradigm. Considering this, it becomes important that the selected language must be as high-level as possible, that is, it should be an appropriate representative of the chosen paradigm without much syntactic or semantic noise.

Due to its popularity, we chose the object-oriented paradigm. As for the representative, we considered classroom-variants of Java, namely COOL [1] and Bantam Java [5] but in the end we decided on Java. The reasoning behind this decision is based on the fact that unlike alternatives listed above, tool support required for executability and formal semantics needed for verification are only available for Java. However, as the main goal of our work is not complete language support, we had to restrict the target language substantially. Moreover, the formal semantics we plan to use defines Java 1.4 [2], therefore we cannot support e.g. generics or lambda functions.

More precisely, our currently supported target language is Java 1.4 – as described by its formal syntax and semantics in [2] – but with the *exclusion* of the following features:

- non-structured control statements, but with the exception of `return` (e.g. no `continue`, `break`, etc.),
- exception handling (e.g. no `throw`, `catch`, etc.),
- modifiers apart from visibility keywords (e.g. no `static`, `final`, etc.),
- field initializer expressions (e.g. no `class A { int x = 0; }`, etc.),
- class initializer blocks (e.g. no `class A { { /* ... */ } }`, etc.),
- local class definitions (e.g. no `class A { class B {} }`, etc.),
- packages, but with the exception of the default one (e.g. no `package a.b;`, etc.),
- reflection (e.g. no `A.class`, etc.),
- concurrency (e.g. no `Thread.start()`, etc.),
- JVM manipulation (e.g. no `ClassLoader.loadClass("A")`, etc.)
- and, naturally, language elements that were introduced in later versions of Java, e.g. generics, lambda functions, annotations, etc. (e.g. no `List<T>`, `(c) -> c + 1`, `@Resource`, etc.).

A number of these restrictions could be bypassed, for example by using anonymous classes instead of lambda functions. Some of them, however, like the loss of generics, indeed limit the current usability of our framework, but we hope to incrementally extend the list of supported language elements in the future.

## 4.2 Refining Program Equivalence

What transformations we consider refactorings is mainly determined by the underlying notion of semantic equivalence. Indeed, both intuitional and formal correctness is based on its chosen definition, which is also an important parameter of the verification backend discussed in Section 3.2. An oversimplified version of the classic characterization of program equivalence demands observed programs to produce the same output for the same input. The problem with this notion, however, is that it is not close enough to abstractions of the target language for a refactoring programmer to being reasoned about on the level of source code. To overcome this issue, we propose to replace the aforementioned definition of equivalence with one of its – more easily specifiable – characterizations, e.g. the preservation of data flow, control flow and binding, as suggested by Schäfer et al. [15].

In addition, individual refactorings are mainly designed not to modify a whole program, but rather specific parts of it – we call the actual extent of a transformation its *scope*. We claim that as a result, it is more natural to think and reason about the correctness of a refactoring concerning only its scope. To support this assumption, instead of using a global definition of program equivalence, we introduce several, generally stricter variants of it, specialized for the possible types of transformation scopes. Ideally, it must be separately proven that each local equivalence implies the chosen global one. In the discussed framework, transformation scope, and therefore equivalence level, can be matched with refactoring schemes.

The following example shows – possibly in its simplest form – how ambiguous it could be to reason about program equivalence in case of partial code fragments, typically seen in rewrite rules.

```
int x = 0;
```

```
int x = 1;
```

Deciding whether the specific code transformation of rewriting the first variable declaration to the second one should be considered a refactoring is not straightforward. In fact, the answer depends on the wider context: if the modified statement is located in an unused private method, the behavior of the enclosing program is guaranteed to remain the same. However, a situation where the behavior truly changes can easily be imagined. This ambiguity is why we reason about a scope-dependent equivalence definition: we do not want the developer of the refactoring to think about conditions which reach out of the current transformation scope.

Another interesting observation arises from the examination of the following pair of class definitions:

```
class A {
    public int f() { return 6; }
}
```

```
class A {
    public int f() { return 3 * g(); }
    public int g() { return 2; }
}
```

In this case, the first question is that how do we characterize the meaning of a class definition? Based on intuition, our proposal is to reduce their equivalence to the equivalence of their public interfaces. On the other hand, adding a new method to a public API while preserving the semantics of its existing methods surely does not alter the previously accessible functionality of the examined library. Therefore, this example shows that the mathematical relation we are looking for is not necessarily an equivalence ($\equiv$, i.e. a relation which is reflexive, transitive and symmetric): exchanging the property of symmetry for antisymmetry, the resulting partial ordering ($\preccurlyeq$) can model the asymmetric nature of program transformations better than an equivalence.

In conclusion, we summarize three types of equivalence levels:

- **Local.** In the lowest level of abstraction, i.e. in case of a refactoring defined over expressions and statements, we cannot leverage any information about the environment of the target. Therefore, here we expect syntactic equivalence.

- **Block.** One abstraction level higher, in case of refactorings concerning code blocks, we can assume that the overall behavior does not depend on block-local variables as long as the blocks themselves are equivalent. We can extend this level to methods if we consider their bodies blocks and their formal parameters block-local variables.

- **Class.** As mentioned above, in case of refactorings modifying classes we only want to ensure that the transformed class hierarchy provides at least the public interface of the original library, but also in a semantically block-equivalent way.

## 4.3 Synthesizing Schemes

When trying to tackle the task of constructing new refactoring schemes, we have to take three main design goals into account: generality, usability and verifiability. The first two of them are interconnected, as schemes possessing a high level of generality tend to be more difficult to instantiate; and conversely, schemes usable with minimal effort usually show a lack of generality. The additional requirement of verifiability demands schemes to appropriately split the two-fold correctness-checking problem between proving their parametric validity wrt. their contract, and checking whether concrete rewrite rules in scheme instances satisfy these assumptions.

With the aim of invoking an intuitional understanding in the reader, we present the main ideas behind two possible iterative techniques for scheme construction.

- **Top-down.** The top-down approach starts from a higher level of abstraction and tries to identify new schemes, or concretize (break down) existing ones based on general categorization possibilities. Two recommended initial categorization dimensions could be the elements of the target language and aspects of program equivalence. For example, if our target language offers

only fields and methods, and we characterize program equivalence with data flow and binding, the top-down method would yield 4 initial schemes: data flow refactoring of fields, data flow refactoring of methods, binding refactoring of fields, binding refactoring of methods.

- **Bottom-up.** The basis of the bottom-up direction is a number of complex, desirably representative refactorings of the target language. Firstly, these complex refactorings have to be decomposed in order to obtain microsteps from them. Then, the resulting refactorings are generalized until they become schemes. Finally, the results can be validated by reconstructing the original base refactorings from scheme instances. Instead of providing a concrete example here, we refer the reader to Section 5, where we discuss this approach in detail.

Both methods have their advantages and disadvantages. The top-down technique yields general schemes by definition, but usually the results are too abstract to be practically usable. On the contrary, schemes constructed with the bottom-up method are inherently usable, but not necessarily general. We can overcome these weaknesses by refining the obtained schemes iteratively. In the former case this means the consideration of additional categorization possibilities, while in the latter more concrete refactorings can be added as a base of the generalization process.

# 5 Case Study

In this section, we present a case study where we identify new schemes based on the bottom-up method. That is, we select and decompose a complex refactoring; modify and extend the description language to accommodate the newly generalized schemes; formally define the chosen base refactoring as a composition of scheme instances; and finally illustrate the usage of the described transformations with their step-by-step application to a concrete example.

Please note that we do not consider the chosen refactoring and the resulting scheme instances as main contributions of this paper. They rather provide only a base for the presented adaptation process of the language discussed in [9] and [8] to OOP. Constructing a refined and widely usable scheme library for Java is still a future work of ours – see Section 6 for details.

## 5.1 The Base Refactoring

As the result of the bottom-up scheme synthesis process highly depends on the chosen base refactoring, it is crucial to select one which can be considered a suitable representative of program transformations defined over the target language. Related work offer numerous candidates. For example, we could pick *generalize function* from [8] or *extract method* from [15]. Although on these we could illustrate the concept of decomposition, none of them depend heavily enough on

object-oriented abstractions. On the other hand, refactorings from Opdyke [13] are too general and complicated for our purposes.

To overcome this problem, we specifically construct a refactoring capable of demonstrating both decomposition and object-oriented concepts. For the former, we reuse *extract method* from Schäfer, which we extend with *lift method* found in Opdyke's dissertation to address the latter. We call this construction *lift segment* and informally specify its semantics as follows. This refactoring, when applied to a consequent region of statements (code segment), lifts its target to the superclass in a separate method. The arguments of the transformation are the visibility and name of the method to be introduced. See Figure 1 for a concrete example.

```
1  class A {}
2  class B extends A {
3      int a, b;
4      void f() {
5          int x = 1;
6          a = x;
7          g();
8          int y;
9          a = y;
10     }
11     void g() { a = b = 0; }
12 }
```

```
1  class A {
2      int a, b;
3      void g() { a = b = 0; }
4      void h(int x) { a = x; g(); }
5  }
6  class B extends A {
7      void f() {
8          int y;
9          int x = 1;
10         h(x);
11         a = y;
12     }
13 }
```

Figure 1: Program code before and after *lift segment*. The refactoring was applied to the segment marked in blue on the left, with the *package* visibility modifier and `h` as function name. Parts of the code changed by the transformation are also highlighted in blue on the right.

## 5.2 Decomposition of the Base Refactoring

To advance towards new schemes, we present the decomposition of the base refactoring step-by-step. We start by dividing the base refactoring itself, and then we continue breaking down the resulting subtransformations recursively, until we get refactorings which are sufficiently simple. For each decomposition step, we provide both an informal reasoning and also a concise list of the derived subtasks. Please note that these descriptions are only meant to invoke a high-level insight in the reader. The presented microrefactorings will be explained in detail in Section 5.7.

As mentioned, first we need to decompose the base refactoring. The joining point between the two main components of our custom construction seems natural to choose for its division.

Lift segment:

1. extract segment,
2. lift method.

The decomposition of *extract segment* has already been presented by Schäfer et al. [15]. Their process consists of three iterative steps, where each of them refines the result of the previous one. This decomposition is specifically defined in a way that separates transformations which together would potentially modify more than one of the three equivalence aspects, namely control flow, data flow and binding.

Extract segment:

1. move segment to block,
2. extract block to lambda,
3. refine and extract lambda.

To preserve name binding, statements of the selected segment is moved in reverse order, one by one to a new block. In this way, variable declarations can be handled separately – this is indeed required, as extracting a referenced declaration might change bindings, and therefore behavior.

Move segment to block:

1. insert new block,
2. move selected statements in the block one by one, in reverse order, handling variable declarations separately.

After the initial segment has been extracted to a new block, it can be transformed into a lambda without the fear of modifying binding during the process. At this point, however, control flow becomes fragile because of the potential jump statements located inside the segment. If value returns are present, the return type of the new method can also be calculated here.

Extract block to lambda:

1. inspect jump statements,
2. inspect external assignments (due to limitations of Java).

In the next step, transformations modifying the data flow are used to make data dependencies related to the lambda (and thus to the initial segment) explicit in the form of parameters and a possible return value. Finally, the now binding- and data/control flow independent lambda can be extracted.

Refine and extract lambda:

1. make data dependencies explicit,
2. extract lambda to method.

As the last step, based on the decomposition of Opdyke [13], we define how a method should be lifted to its superclass.

Lift method:

1. lift referenced local fields,
2. lift referenced local methods[1],
3. lift independent method.

In the following subsections we show how microrefactorings listed above can be defined with the adapted framework.

## 5.3 Extending the Description Language

In this section we discuss modifications and extensions of the description language required to accommodate the new refactoring schemes. As part of the process, not only do we introduce new scheme clauses related to the object-oriented paradigm, but we also mention new language elements meant to make even existing refactoring definitions more concise.

In the base language, pseudovariable **`this`** represents the target of the current refactoring. Because this identifier has a different meaning in the object-oriented paradigm, we replace ours with **`target`** to avoid confusion. Additionally, we make syntactic patterns more expressive by the flexible handling of the `;` delimiter in them: instead of a concrete syntactic element, we interpret it as an abstract sequencing symbol with multiple possible manifestations. For example, pattern `#S;#S'` matches both `{}{}` and `{};{}`.

Finally, we introduce the following scheme clauses:

- **`target`**: A clause dedicated to match the target of the refactoring. On the one hand, this can eliminate pattern duplications in the original matching pattern. Moreover, referring to the context of the target in the matching pattern also becomes possible. The following examples respectively present the above-mentioned interpretations of the target clause:

```
1    target
2    ----------
3    { target }
4  target
5    #S ; #S'
```

```
1    target ; #S'
2    ------------
3    #S' ; target
4  target
5    #S
```

Concrete usage analogous with the previous examples can be found, respectively, in scheme instances `moveIntoNextBlock` and `moveToTop`, see Section 5.6. The first one eliminates duplication from the description of a move transformation by using the target expression in its rewrite-pattern-pair. The second one uses the target expression to make the selected code's environment accessible for a precondition.

[1] Note the recursion.

- **`shadowed references`**: Clause for specifying a compensational transformation for changes in binding induced by moving code. For example, this is where we can restore binding to a field by using the **`this`** qualifier after shadowing it with a local declaration – as seen in the description of the block refactoring scheme in Section 5.5.

- **`top level definition`**: Clause to define or modify a file-level program entity (e.g. class or interface). For example, the second variant of the lambda scheme uses this clause to modify an interface, as seen in Section 5.5.

- **`definition in class`**: Clause for defining a new member (e.g. field or method) inside the enclosing class of the refactoring target. A concrete example can be found in scheme instance `extract` in Section 5.6, where the clause is used to introduce a new method.

- **`definition in super`**: Clause for defining a new member inside the superclass of the refactoring target's enclosing class. This is, for example, how we lift a method in the second variant of the class refactoring scheme by removing it from the base class and reintroducing it in the superclass by the *definition in super* clause, see in Section 5.5.

## 5.4 Constructing the Metatheory

By metatheory we denote the semantic functions and predicates that capture, and provide a high-level interface for, various static semantic information about the target language. In particular, the metatheory defines what predicates the preconditions of schemes and scheme instances can be built from. To make the metatheory intuitively usable, we define these functions and predicates over a high-level model that closely resembles the abstractions of the target language. The basis of this model is the AST metamodel, therefore we will implicitly use operations commonly defined on it.

In the following we group elements of our metatheory by the semantic property they provide information about, separately listing the ones which are closely related to the object-oriented paradigm.

**Data flow.** Information about data flow can be used to check variables and fields before and during a transformation, or even while verifying a scheme or an instance. Here we declare functions for obtaining variables and fields through a given entity: `variableReads`, `variableWrites` and `accessedFields`.

**Control flow.** One of the main notions of control flow analysis is the concept of the path of execution, describing a possible ordering of statements for a given language entity. Here we reuse the definition of `controlSuccessor` and `exitNode` from Schäfer et al. [15], referring to possible control flow successors and the symbolic exit point of a method, respectively. We also introduce the `callGraph` of a method

definition, which is the maximal, directed, vertex-labeled graph of method definitions containing all *possible* (see dynamic binding) call relations starting from the selected method definition.

**Binding.** One of our most important semantic functions is `definitionScope`.

**Definition 1.** *The scope of method definition $d$ is the set which contains exactly the classes whose instances resolve method calls with $d$'s signature to $d$.*

As a demonstration, consider the following example:

```
1  class A {
2      public void f() { /* ... */ }
3  }
4  class B extends A {
5      public void f() { /* ... */ }
6  }
7  class C extends B {
8      public void f() { /* ... */ }
9  }
10 class D extends B {}
```

Here the scope of `A::f()` consists of `A`, the scope of `B::f()` consist of `B` and `D`, and the scope of `C::f()` consist of `C`.

**Paradigm.** Statically reasoning about the behavior of object-oriented software is made difficult by dynamic aspects of the paradigm, namely polymorphism and dynamic binding. Apart from trivial query functions (e.g. `isSubType`, `superHierarchy`, `subHierarchy`), our main concern here is to find a proper approximation of behavioral properties which may influence our class-conformity relation (see Section 4.2).

Considering the microrefactorings we identified in Section 5.2, a number of them requires a new method definition to be added into a class. In the following, we will call such to-be-added definitions *predefinitions*. Our goal regarding the metatheory is to provide a safe approximation for deciding whether a predefinition could potentially change how a method reachable from public API behaves. For the sake of simplicity, we do not statically check whether two method definitions are equivalent – we simply assume that they are not. In conclusion, we propose the following definitions about the so-called intra- and interhierarchy-reachability of a predefinition.

Please note that the following definitions are meant as readable alternatives to the underlying logic formulae.

**Definition 2.** *We say that predefinition $p$ is* reachable *if it*

- *overrides a method,*
- *and is* inter- *or* intrahierarchy-reachable.

**Definition 3.** *We say that predefinition $p$ is* interhierarchy-reachable *if there exists a definition which*

- *is located outside the class hierarchy of $p$*
- *and refers to a signature of $p$ that*
  - *is qualified with either one of the superclasses of $p$'s enclosing class,*
  - *or with a class belonging to the definition scope of $p$,*
- *and which is* non-constrained intrahierarchy-reachable.

**Definition 4.** *We say that predefinition $p$ is* intrahierarchy-reachable *if*

- *it overrides a public method,*
- *or there exists a definition which*
  - *refers to the unqualified signature of $p$*
  - *and is $D_p$-*constrained intrahierarchy-reachable *where*
    - *$D_p$ is the definition scope of $p$.*

**Definition 5.** *Definition $d$ is $D$-*constrained intrahierarchy-reachable *if*

- *$D_i$ is not empty, and*
  - *either the visibility of $d$ is public,*
  - *or there exists a definition $d'$ that*
    - *refers to the unqualified signature of $d$,*
    - *and is $D_i$-*constrained intrahierarchy-reachable
  - *where $D_i$ is the intersection of $D$ and $D_d$ where*
    - *$D_d$ is the definition scope of $d$.*

**Definition 6.** *Definition $d$ is* non-constrained intrahierarchy-reachable *if it is*

- *$D_d$-*constrained intrahierarchy-reachable *where*
  - *$D_d$ is the definition scope of $d$.*

In short, interhierarchy-reachability denotes whether a predefinition could be called from an external public API, while intrahierarchy-reachability indicates if a predefinition could be resolved from a public API inside its class hierarchy. The reason why the latter is slightly more complex is the fact that in that case, there is a possibility for specific call-chains, starting from a public method and almost reaching a predefinition, to be broken due to disjoint definition scopes.

We illustrate these reachability-definitions with the following three examples. In the first one, the essence of interhierarchy-reachability is shown.

```
1 class A {
2     protected void f() { /* ... */ }
3 }
4 class B extends A {
5     /* protected void f() { /* ... */ } */
6 }
```

```
 7 class C extends B {}
 8 class D extends C {
 9     protected void f() { /* ... */ }
10 }
11 class X {
12     public void g(A a, C c, D d) {
13         a.f(); c.f(); d.f();
14     }
15 }
```

Here, the now commented-out `B::f()` denotes the to-be-added definition. Its enclosing hierarchy consists of classes `A`, `B`, `C` and `D`, where `B` and `C` form its definition scope. Class `X` lies outside of this hierarchy. In definition `X::g(A, C, D)`, which is obviously reachable because of its public visibility, signatures `A::f()`, `C::f()` and `D::f()` are called. In this method, the `d.f()` call is safe, as the dynamic type of `d` can only be `D`, and `D` is not in the scope of predefinition `B::f()` – therefore, the newly added method could not possibly be called. However, the other two calls are unsafe, because for both of them there exists a compatible class from the scope of the predefinition. For example, in both cases the dynamic type of the called object can be `C`, which would result in signatures `A::f()` and `C::f()` being resolved to predefinition `B::f()` – therefore, it is interhierarchy-reachable.

The second example demonstrates an intrahierarchy-reachable predefinition.

```
1 class A {
2     protected void f() { /* ... */ }
3     public void g() { f(); }
4 }
5 class B extends A {
6     /* protected void f() { /* ... */ } */
7 }
```

Once again, the commented-out `B::f()` denotes the predefinition. The publicly defined, and therefore intrahierarchy-reachable `A::g()` in its superclass calls `A::f()` without qualifiers. Generally, this would not necessarily be problematic – see the next example. This call, however, is still unsafe, because the definition scopes of `A::g()` – i.e. {`A`, `B`} – and `B::f()` – i.e. {`B`} – are not disjoint. Indeed, as a result, on instances of `B`, the publicly accessible signature `B::g()` is resolved to definition `A::g()`, which then calls predefinition `B::f()` in place of signature `A::f()`.

The last example shows how disjoint definition scopes can prevent a predefinition from being intrahierarchy-reachable.

```
1 class A {
2     protected void f() { /* ... */ }
3     public void g() { f(); }
4 }
5 class B extends A {
6     /* protected void f() { /* ... */ } */
7     public void g() {}
8 }
```

Compared to the previous example, definitions `A::f()` and `A::g()`, as well as predefinition `B::f()` remain the same. The only difference is the introduction of definition `B::g()`, which reduces the definition scope of `A::g()` to just {`A`}. As a result, the scopes of `A::g()` and `B::g()` become disjoint, thus there are no classes where `A::g()` could call `B::f()`. Because there are no other references to signature `f()` inside this hierarchy, the predefinition here is not intrahierarchy-reachable.

## 5.5 Identifying Refactoring Schemes

During the discussion in previous sections, we introduced all concepts and tools necessary for constructing our own schemes, based on a generalization of microstreps that were presented in the form of the chosen base refactoring's decomposition. To define a scheme, we have to provide its name, potential clauses, rewrite control logic, preconditions and contracts. We also assign a level of equivalence to each scheme in accordance with its transformation scope. In the following we briefly show the four scheme(families) we specified: local, block, lambda and class.

**Local refactoring scheme.** The local scheme can be used to define simple, block-local refactorings on the level of single expressions and statements. It has no special preconditions or control logic.

```
1 local refactoring <name>
2     <matching pattern>
3     ---------------------
4     <replacement pattern>
5 target
6     <optional target pattern>
7 when
8     isInsideBlock(target)
9     and <optional preconditions>
```

In fact, the local scheme can be considered as a way to purely embed conditional term rewrite rules into the language. The sole precondition requires its target to be inside a block (line 8). Naturally, the assigned equivalence level is *local* and the contract of the scheme demands the matching and replacement patterns to be locally equivalent when preconditions hold.

**Block refactoring scheme.** The block scheme can be seen as an extension to the local one. It can be used to refactor entire code blocks and even contains some simple control logic.

```
1 block refactoring <name>
2     <matching block-pattern>
3     --------------------------
4     <replacement block-pattern>
5 target
6     <optional target pattern>
```

```
7  shadowed references
8      #reference
9      --------------
10     #qualifiedName
11 when
12     #qualifiedName = #reference.qualifiedName()
13     and <optional preconditions>
```

Inside the matching and replacement patterns we allow a special type of pattern matching. If keyword **target** is explicitly referenced there in a way that its context is unboundedly matched (this is achievable with multipatterns (e.g. `#S..`) at the beginning and/or end of mentioned pattern holes), bounding multipatterns will be matched until the boundary of the enclosing code block. See instance `moveIntoNextBlock` in the next section for an example.

If we move statements in a block, it is possible to introduce unwanted variable shadowings, therefore to alter the original binding. The control logic of the scheme, as can be seen in its **shadowed references** clause, automatically compensates this by appending the original name qualification (line 12) to the shadowed reference.

This scheme was designed with the *block* equivalence level in mind. Its contract requires the block-equivalence of matching and replacement patterns considering preconditions and automatic name qualification.

**Lambda refactoring scheme.** As Schäfer et al. [15] suggest in their work, lambda functions are practical because of their ability to act either as data or as code, making it possible to destruct error-prone refactorings which modify both data and control flow at once into multiple smaller, cleaner transformations. Unfortunately, the formal semantics we plan to base the verification on does not support lambda functions, therefore we have to use interfaces and anonymous class instances instead.

We have identified two variants of the lambda scheme: one for introducing new and one for modifying existing lambda interfaces and instances. Here we present the latter.

```
1  lambda refactoring <name>
2      <matching lambda-pattern>
3      ----------------------------
4      <replacement lambda-pattern>
5  top level definition
6      <interface definition (#F) for the matching lambda-pattern>
7      ------------------------------------------------------------
8      <interface definition for the replacement lambda-pattern>
9  when
10     #F.references().size() = 1
11     and #F.references().contains(target)
12     and <optional preconditions>
```

The main task here is to automatically update underlying interface definitions. For example, we expect the framework to propagate changes between the matching and replacement lambda applications to the corresponding interface, denoted by metavariable `#F` in the description of the scheme. As we want to keep the transformation scope local (we are only discussing special method calls), in the preconditions we verify that the lambda interface is not used anywhere else, i.e. it is only referenced once (line 10) and that one reference is the target of the refactoring (line 11). In accordance with this, the scheme is based on the *local* level of equivalence and its contract demands the matching and replacement patterns to be locally equivalent considering preconditions and interface versions.

**Class refactoring scheme.** The class scheme was designed for refactorings that modify classes and class members. We constructed three variants in this category: one for introducing new methods, one for lifting methods and one for lifting fields. The reason behind excessive concretization was mainly the complexity of preconditions: we wanted to hide them from users inside the scheme. In the following we discuss the variant intended to add new methods into the enclosing class.

```
1  class refactoring <name>
2      <matching pattern>
3      ------------------
4      #name(#args..)
5  target
6      <optional target pattern>
7  definition in class
8      #visibility #type #name(#params..) #body
9  when
10     /* omitted for the sake of readability */
11     and <optional preconditions>
```

Structurally, the scheme looks quite simple: a new method is introduced into the enclosing class, and the matching pattern is replaced with a corresponding function call. Missing arguments (e.g. `#name`, `#body`, etc.) must be inferred from concrete instances. However, the scheme's true complexity is encoded into its preconditions which we omitted here for the sake of readability. In short, we have to guarantee that the new definition will not cause compiler errors (names are unique, in case of overrides visibility and types are correct, etc.) and also want to check that the predefinition is not reachable (see Section 5.4).

Naturally, this scheme uses the *class* level of equivalence (which, in this case, is technically a partial ordering ($\preccurlyeq$), see Section 4.2). However, due to the exhaustive preconditions, its contract only requires the matching and replacement patterns to be locally equivalent. Of course when checking this we assume that the preconditions hold and that the new method definition – which is called in the replacement pattern – has been inserted to the enclosing class.

## 5.6 Defining Scheme Instances

Using the schemes introduced in the previous section, we can define the decomposed microrefactorings as scheme instances. At the end of this part, we also demonstrate how the base refactoring can be rebuilt from microsteps in a composite specification.

**Local refactorings.** The only local refactoring is the one which appends a new, empty code block after its target statement.

```
1 local refactoring introduceEmptyBlockAfter()
2     #s
3     -------
4     #s ; {}
```

**Block refactorings.** There are two block instances: one for moving a statement into its subsequent block (`moveIntoNextBlock`) and one for handling declarations differently during the process (`moveToTop`). Here we show only the first one, but the latter could be easily constructed as well.

```
1 block refactoring moveIntoNextBlock()
2     target ; { #S.. } ; #S'..
3     -------------------------
4     { target ; #S.. } ; #S'..
5 when
6     isSingle(target)
7     and (isVariableDeclaration(target) ->
8         not isReferencedIn(target.declaredVariable(), #S'..))
```

Note how the transformation is expressed using only pattern matching. As we can see from the precondition, the block-matching feature helps to obtain the surrounding context without the use of semantic functions. Here we require the target to be a single statement (line 6) – as we want to move only one statement at a time, see Section 5.2 –, and also if it is a declaration (line 7), its declared variable should not be referenced in the remainder of the block (line 8) – since for these references, the original declaration would become invisible if we moved it into the sub-block.

**Lambda refactorings.** In total, we have defined three lambda refactorings. One introduces a void (`wrapInVoidLambda`), the other one constructs a value-returning lambda (`wrapInValueLambda`). Now we present the third one (`extractInVariables`), which makes data dependencies of an existing lambda explicit.

```
1 lambda refactoring extractInVariables()
2     new #F() { public #type #name() #body }.#name()
3     ----------------------------------------------------------------
4     new #F() { public #type #name(#inVars..) #body }.#name(#inVars..)
5 when
6     #inVars.. = #body.variableReads().filter(#read :
7             isBefore(#read.variable().declaration(), target))
8         .map(#read : #read.variable()).reduce()
```

In the first pattern, metavariable `#F` will be matched to the underlying interface of the targeted "lambda"-application. Here you can also see that metavariables might even be assigned in preconditions. In this particular example, variables read in the body of the target lambda (`#body.variableReads()`), but declared before (not inside) it (line 7), are collected into and later used through metavariable `#inVars...` We also take advantage of the fact that the collected variable names can be used both as formal and actual parameters. (The last line of the precondition is a technicality: we have to convert the filtered variable reads to the read variables, and also eliminate duplications (`reduce()`) from the resulting collection, as it will be used as a parameter/argument list.)

**Class refactorings.** Out of the three class refactoring instances, we show the most interesting one, that is which extracts a lambda to a new method (`extract`). The other two (one for lifting methods and one for lifting fields, both named `lift`) can be mechanically specified without significant extra content.

```
1  class refactoring extract(#visibility, #newName)
2      new #F() { public #type #name(#params..) #body }.#name(#args..)
3      ----------------------------------------------------------------
4      #newName(#args..)
5  definition in class
6      #visibility #type #newName(#params..) #body
7  when
8      isSubsetOf(#body.dataAccesses().map(#access : #access.target()),
9          union(#params.., target.enclosingClass().fields(),
10             #body.localVariables()))
```

Similarly to the previous example, metavariable `#F` in the first pattern will be matched to the underlying interface of the targeted "lambda"-application. The difference here is that we have to check whether the lambda to be extracted is truly independent from its surroundings, that is, it does not reference variable-like entities from outside its parameters, body and accessible fields. In other words, the referenced variables (`#access.target()`) of its body's data accesses (`#body.dataAccesses()`) should form a subset of the union of its parameters, local variables and accessible fields of the enclosing class (line 9-10). This instance also has two parameters, the name and visibility of the new method.

**Composite refactorings** Finally, we can reconstruct the initial *lift segment* refactoring in the composite definition of `lift`.

```
1  composite refactoring lift(#visibility, #name)
2  do
3      extract(#visibility, #name)
4      #extractedMethod = target.enclosingMethod()
5      #extractedMethod.cascadedLift()
6  when
7      isSegment(target)
```

Note that here, `extract` and `cascadedLift` are further composite refactorings built from scheme instances mentioned above. In this specification it is also shown how selectors and combinators can be used to imperatively control the application of transformations: in line 4, we store a reference to the method extracted in the previous line in a metavariable, then in the next line we lift it together with its dependencies by applying `cascadedLift` through the referencing metavariable.

## 5.7 Example

In this section, the previously discussed scheme instances and composite refactorings are demonstrated by the stepwise transformation of a concrete Java program (see Figure 1). Each step is presented by a code fragment pair showing the before-after state, followed by a short explaination of the applied transformation. Code highlighted in blue denotes the target of the refactoring on the left, and the currently modified parts on the right.

The first step is to apply instance `lift` with the *package* visibility and function name `h` as parameters. Refactoring `lift` is composite – within it, `extract` is called, which is also composite, and its first step is `introduceEmptyBlockAfter`.

```
1  class A {}
2  class B extends A {
3      int a, b;
4      void f() {
5          int x = 1;
6          a = x;
7          g();
8          int y;
9          a = y;
10     }
11     void g() { a = b = 0; }
12 }
```

```
1  class A {}
2  class B extends A {
3      int a, b;
4      void f() {
5          int x = 1;
6          a = x;
7          g();
8          int y;
9          {}
10         a = y;
11     }
12     void g() { a = b = 0; }
13 }
```

After that, `moveIntoNextBlock` is applied, but it does not succeed, because the scope of a declaration that is used in the enclosing block cannot be reduced. The `moveToTop` rule is tried next, which will be performed successfully.

```
1  class A {}
2  class B extends A {
3      int a, b;
4      void f() {
5          int x = 1;
6          a = x;
7          g();
8          int y;
9          {}
10         a = y;
11     }
12     void g() { a = b = 0; }
13 }
```

```
1  class A {}
2  class B extends A {
3      int a, b;
4      void f() {
5          int y;
6          int x = 1;
7          a = x;
8          g();
9          {}
10         a = y;
11     }
12     void g() { a = b = 0; }
13 }
```

The next statement is moved to the target block using `moveIntoNextBlock`.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        g();
        {}
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        {
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

We move the first statement of the originally selected segment to a block using `moveIntoNextBlock` as well.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        {
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        {
            a = x;
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

Since the resulting block does not contain a `return` statement, we can use the `wrapInVoidLambda` rule to convert it to a lambda. Note that this also creates the corresponding interface.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        {
            a = x; g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { public void apply() {
            a = x; g();
        } }.apply();
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(); }
```

In the next step, the input parameters of the generated lambda are extracted by refactoring `extractInVariables`. `x` is one such parameter, as it is a local variable declared outside of the lambda, but field `a` can be accessed inside the class even with the current unqualified reference.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { void apply() {
            a = x; g();
        } }.apply();
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(); }
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { void apply(int x) {
            a = x; g();
        } }.apply(x);
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(int x); }
```

The last step in `extract segment` is to convert the lambda to a method with the `extract` rule. Although removing the interface that is no longer used is formally not a part of this rule, such a refactoring could be easily defined, thus we omit it from the example code to improve readability.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() {
            public void apply(int x) {
                a = x; g();
            }
        }.apply(x);
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(int x); }
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

Since the first part of `lift`, which is responsible for segment extraction, has been completed, we now proceed to the second subtransformation. This is the – also composite – `cascadedLift` that lifts the method created in the previous phase to the superclass. First, field `a` used in `h` is lifted using the `lift` rule.

```
class A {}
class B extends A {
    int a;
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a;
}
class B extends A {
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

In the next step, method `g` – referenced in `h` – is targeted, on which we recursively call the composite `cascadedLift` refactoring. Its first step is to select field `b` used in `g` and lift it with the `lift` rule.

```
class A {
    int a;
}
class B extends A {
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a;
    int b;
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

Since `g` has no more dependencies from its enclosing class, we can lift it to the superclass with the `lift` rule.

```
class A {
    int a, b;
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a, b;
    void g() { a = b = 0; }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void h(int x) { a = x; g(); }
}
```

With the previous step, we completed the transformations needed to lift the dependencies of `h`, so now we can lift `h` itself using the `lift` rule. Thus we performed the second, method-lifting part of our initial refactoring. Since this was the last one, in this step the whole transformation terminates successfully.

```
class A {
    int a, b;
    void g() { a = b = 0; }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a, b;
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
}
```

# 6 Future Work

In previous sections we presented a general outline for the adaptation process. Although we aimed to make the case study as constructive as possible, comprehensive support for the object-oriented paradigm is yet to be realized. Here we share two natural continuations of our research which may improve upon this aspect.

## 6.1 More Schemes and Case Studies

The main concept behind the discussed framework is the notion of refactoring schemes. Therefore it would be beneficial to examine the scheme construction method in finer detail. For example, it would be interesting to conduct more case studies and to compare the different schemes obtained from them. Moreover, the relationship between the top-down and bottom-up approaches also raise additional questions. For example, could results from the two be unified?

## 6.2 Verification

After an established set of refactoring schemes is constructed, research could proceed with formal verification. Considering the current verification backend, this would mean that almost all language artifacts, including schemes, levels of equivalence and metatheory would need to be formalized in a model compatible with the chosen operational semantics of the target language. Ideally, schemes could be verified manually by structural induction, while scheme instantiation, that is, conformity to scheme contracts would become automatically verifiable.

# 7 Conclusion

In this paper we presented a proposal for adapting a domain-specific refactoring language from the functional to the object-oriented programming paradigm, using Java instead of Erlang as a representative.

As part of this task, we briefly introduced the original refactoring framework and discussed its description language as well as its verification technique. We also gave an overview of related research.

Then we approached the problem from a high-level perspective, presenting our reasoning about how the adaptation process shall be carried out. We showed how and why the choice of target language and paradigm arose, then discussed how a multilayered definition of equivalence, or even a partial ordering can help to characterize the behavior-preserving property of refactorings in a more intuitive way. We also presented two iterative methods for synthesizing new transformation schemes in the form of the top-down and bottom-up approaches.

Using the latter, we conducted a complex case study where we showed the decomposition of a compound refactoring rule called *lift segment*. With the goal of reconstructing this transformation inside the adapted framework, we began to discuss how different parts of the system should be extended to achieve this target.

In this process, we added new elements to the description language, identified suitable semantic functions and predicates for the target language metatheory (including the notion of inter- and intrahierarchy-reachability) and proposed a set of generalized refactoring schemes. To conclude the case study, we presented formal, scheme-based definitions for decomposed building blocks of the original refactoring, and demonstrated them by the stepwise transformation of a concrete program. Finally, we listed future research directions.

Based on the case study, we conclude that the first steps towards adapting the scheme-based refactoring approach to OOP have been successful: we were able to express a complex Java refactoring in the modified language. As part of this, we found a suitable decomposition for this transformation, and then we were able to generalize schemes from the resulting microsteps. By constructing an appropriate program equivalence, a description language and a metatheory, we managed to make the identified schemes definable. We have seen that these schemes are already suitable for expressing the initial base refactoring. Their generality obviously still falls short, but we hope that a more comprehensive scheme library can be built with the presented technique in the future.

## References

[1] Aiken, Alexander. Cool: A Portable Project for Teaching Compiler Construction. *SIGPLAN Not.*, 31(7):19–24, July 1996. DOI: 10.1145/381841.381847.

[2] Bogdanas, Denis and Roşu, Grigore. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA, 2015. ACM. DOI: 10.1145/2676726.2676982.

[3] Bravenboer, Martin, Kalleberg, Karl Trygve, Vermaas, Rob, and Visser, Eelco. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72:52–70, 2008. DOI: 10.1016/j.scico.2007.11.003.

[4] Ciobâcă, Ştefan, Lucanu, Dorel, Rusu, Vlad, and Roşu, Grigore. A Language-Independent Proof System for Full Program Equivalence. *Formal Aspects of Computing*, 28(3):469–497, mar 2016. DOI: 10.1007/s00165-016-0361-7.

[5] Corliss, Marc L., Furcy, David, Davis, Joshua, and Pietraszek, Lori. Bantam Java Compiler Project: Experiences and Extensions. *J. Comput. Sci. Coll.*, 25(6):159–166, June 2010. URI: http://dl.acm.org/citation.cfm?id=1791129.1791160.

[6] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN: 0201485672.

[7] Garrido, Alejandra and Meseguer, Jose. Formal Specification and Verification of Java Refactorings. *Proceedings - Sixth IEEE International Workshop on*

*Source Code Analysis and Manipulation, SCAM 2006*, pages 165–174, 2006. DOI: 10.1109/SCAM.2006.16.

[8] Horpácsi, Dániel, Kőszegi, Judit, and Horváth, Zoltán. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In Lisitsa, Alexei, Nemytykh, Andrei P., and Proietti, Maurizio, editors, Proceedings of the Fifth International Workshop on *Verification and Program Transformation,* Uppsala, Sweden, 29th April 2017, Volume 253 of *Electronic Proceedings in Theoretical Computer Science*, pages 92–108. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.253.8.

[9] Horpácsi, Dániel, Kőszegi, Judit, and Thompson, Simon. Towards Trustworthy Refactoring in Erlang. In Hamilton, Geoff, Lisitsa, Alexei, and Nemytykh, Andrei P., editors, Proceedings of the Fourth International Workshop on *Verification and Program Transformation,* Eindhoven, The Netherlands, 2nd April 2016, Volume 216 of *Electronic Proceedings in Theoretical Computer Science*, pages 83–103. Open Publishing Association, 2016. DOI: 10.4204/EPTCS.216.5.

[10] Kalleberg, Karl Trygve. *Abstractions for Language-Independent Program Transformations.* PhD thesis, University of Bergen, Bergen, Norway, 2007. URI: http://hdl.handle.net/1956/3287.

[11] Leitão, António. A Formal Pattern Language for Refactoring of Lisp Programs. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 186–192, 2002. DOI: 10.1109/CSMR.2002.995803.

[12] Li, Huiqing and Thompson, Simon. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering*, pages 501–515, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-28872-2_34.

[13] Opdyke, William F. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645. URI: http://hdl.handle.net/2142/72072.

[14] Roşu, Grigore and Ştefănescu, Andrei. From Hoare Logic to Matching Logic Reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, Volume 7436 of *LNCS*, pages 387–402. Springer, Aug 2012. DOI: 10.1007/978-3-642-32759-9_32.

[15] Schäfer, Max, Verbaere, Mathieu, Ekman, Torbjörn, and de Moor, Oege. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag. DOI: 10.1007/978-3-642-03013-0_17.

[16] Schäfer, Max and de Moor, Oege. Specifying and Implementing Refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 286–301, New York, NY, USA, 2010. Association for Computing Machinery. DOI: 10.1145/1869459.1869485.

[17] Sommerville, Ian. *Software Engineering.* Addison-Wesley Publishing Company, USA, 9th edition, 2010. ISBN: 0137035152.

[18] Stefănescu, Andrei, Park, Daejun, Yuwen, Shijiao, Li, Yilong, and Roşu, Grigore. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 74–91, New York, NY, USA, 2016. ACM. DOI: 10.1145/2983990.2984027.

[19] Verbaere, Mathieu, Ettinger, Ran, and Moor, Oege. JunGL: a Scripting Language for Refactoring. In *Proceedings – International Conference on Software Engineering*, Volume 2006, pages 172–181, 01 2006. DOI: 10.1145/1134311.

[20] Visser, Eelco and Benaisse, Zine-el-Abidine. A Core Language for Rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, 1998. DOI: 10.1016/s1571-0661(05)80027-1.

Acta Cybernetica **25** (2020) 847–876.

# A Modern Look at GRIN, an Optimizing Functional Language Back End*

Péter Dávid Podlovics[ab], Csaba Hruska[c], and Andor Pénzes[d]

**Abstract**

GRIN is short for Graph Reduction Intermediate Notation, a modern back end for lazy functional languages. Most of the currently available compilers for such languages share a common flaw: they can only optimize programs on a per-module basis. The GRIN framework allows for interprocedural whole program analysis, enabling optimizing code transformations across functions and modules as well.

Some implementations of GRIN already exist, but most of them were developed only for experimentation purposes. Thus, they either compromise on low level efficiency or contain ad hoc modifications compared to the original specification.

Our goal is to provide a full-fledged implementation of GRIN by combining the currently available best technologies like LLVM, and evaluate the framework's effectiveness by measuring how the optimizer improves the performance of certain programs. We also present some improvements to already existing components of the framework. Some of these improvements include a typed representation for the intermediate language and an interprocedural program optimization, the dead data elimination.

**Keywords:** GRIN, compiler, whole program optimization, intermediate representation, dead code elimination

## 1 Introduction

Over the last few years, the functional programming paradigm has become more popular and prominent than it was before. More and more industrial applications emerge, the paradigm itself keeps evolving, existing functional languages are being refined day by day, and even completely new languages appear. Yet, it seems the corresponding compiler technology lags behind a bit.

*The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: peter.d.podlovics@gmail.com, ORCID: 0000-0002-4848-883X
[c]E-mail: csaba.hruska@gmail.com, ORCID: 0000-0002-6168-1570
[d]E-mail: andor.penzes@gmail.com, ORCID: 0000-0002-6221-4579

DOI: 10.14232/actacyb.282969

Functional languages come with a multitude of interesting features that allow us to write programs on higher abstraction levels. Some of these features include higher-order functions, laziness and sophisticated type systems based on SystemFC [29], some even supporting dependent types. Although these features make writing code more convenient, they also complicate the compilation process.

Compiler front ends usually handle these problems very well, but the back ends often struggle to produce efficient low level code. The reason for this is that back ends have a hard time optimizing code containing *functional artifacts*. These functional artifacts are the by-products of high-level language features mentioned earlier. For example, higher-order functions can introduce unknown function calls and laziness can result in implicit value evaluation which can prove to be very hard to optimize. As a consequence, compilers generally compromise on low level efficiency for high-level language features.

Moreover, the paradigm itself also encourages a certain programming style which further complicates the situation. Functional code usually consists of many smaller functions, rather than fewer big ones. This style of coding results in more composable programs, but also presents more difficulties for compilation, since optimizing individual functions only is no longer sufficient.

In order to resolve these problems, we need a compiler back end that can optimize across functions as well as allow the optimization of laziness in some way. Also, it would be beneficial if the back end could theoretically handle any suitable front end language.

In this paper we present a modern look at the GRIN framework. We explain some of its core concepts, and also provide a number of improvements to it. The results are demonstrated through a modernized implementation of the framework[1]. The main contributions presented in the paper are the following.

1. Extension of the heap points-to analysis with more accurate basic value tracking
2. Specification of a type inference algorithm for GRIN using the extended heap points-to analysis
3. Implementation of an LLVM back end for the GRIN framework
4. Extension of the dead data elimination transformation with typed dummification and an overview of an alternative transformation for producer-consumer groups
5. Implementation of an Idris front end for the GRIN framework

## 2 Graph Reduction Intermediate Notation

GRIN is short for *Graph Reduction Intermediate Notation*. GRIN consists of an intermediate representation language (IR in the following) as well as the entire

[1] Almost the entire GRIN framework has been reimplemented. The only exceptions are the simplfyifing transformations which are no longer needed by the new code generator that uses LLVM as its target language.

compiler back end framework built around it. GRIN tries to resolve the issues highlighted in Section 1 by using interprocedural whole program optimization.

## 2.1 General overview

Interprocedural program analysis is a type of data-flow analysis that propagates information about certain program elements through function calls. Using interprocedural analyses instead of intraprocedural ones, allows for optimizations across functions. This means the framework can handle the issue of large sets of small interconnecting functions presented by the composable programming style.

Whole program analysis enables optimizations across modules. This type of data-flow analysis has all the available information about the program at once. As a consequence, it is possible to analyze and optimize global functions. Furthermore, with the help of whole program analysis, laziness can be made explicit. In fact, the evaluation of suspended computations in GRIN is done by an ordinary function called `eval`. This is a global function uniquely generated for each program, meaning it can be optimized just like any other function by using whole program analysis.

Finally, since the analyses and optimizations are implemented on a general intermediate representation, many other languages can benefit from the features provided by the GRIN back end. The intermediate layer of GRIN between the front end language and the low level machine code serves the purpose of eliminating functional artifacts from programs such as closures, higher-order functions and even laziness. This is achieved by using optimizing program transformations specific to the GRIN IR and functional languages in general. The simplified programs can then be optimized further by using conventional techniques already available. For example, it is possible to compile GRIN to LLVM and take advantage of an entire compiler framework providing a huge array of very powerful tools and features.

## 2.2 A small example

As a brief introduction to the GRIN language, we will show how a small functional program can be encoded in GRIN. We will use the following example program: `(add 1) (add 2 3)`. The `add` function simply takes two integers, and adds them together. This means, that the program only makes sense in a language that supports partial function application, due to `add` being applied only to a single argument. We will also assume, that the language has lazy semantics. We can see the GRIN code generated from the above program in Program code 2.1.

The first thing we can notice is that GRIN has a monadic structure, and syntactically it is very similar to low-level Haskell. The second one, is that it has data constructors (`CInt`, `Fadd`, etc). We will refer to them as *nodes*. Thirdly, we can see four function definitions: `grinMain`, the main entry point of our program; `add`, the function adding two integers together; and two other functions called `eval` and `apply`. Lastly, we can see `_prim_int_add` and the `store`, `fetch` and `update` operations, which do not have definitions. The first one is a primitive operation, and the last three are intrinsic operations responsible for graph reduction. We can

```
grinMain =
  a <- store (CInt 1)
  b <- store (CInt 2)
  c <- store (CInt 3)

  r <- store (Fadd b c)
  suc <- pure (P1_add a)
  apply suc r

add x y =
  (CInt x1) <- eval x
  (CInt y1) <- eval y
  r <- _prim_int_add x1 y1
  pure (CInt r)
```

```
eval p =
  v <- fetch p
  case v of
    (CInt _n)   -> pure v
    (P2_add)    -> pure v
    (P1_add _x) -> pure v
    (Fadd x2 y2) ->
      r_add <- add x2 y2
      update p r_add
      pure r_add

apply f u =
  case f of
    (P2_add) ->
      pure (P1_add u)
    (P1_add z) -> add z u
```

Program code 2.1: GRIN code generated from `(add 1) (add 2 3)`

also view `store`, `fetch` and `update` as simple heap operations: `store` puts values onto the heap, `fetch` reads values from the heap, and `update` modifies values on the heap.

The GRIN program is always a first order, strict, defunctionalized version of the original program, where laziness and partial application are expressed explicitly by `eval` and `apply`. A lazy function call can be expressed by wrapping its arguments into an `F` node. As can be seen, the `add 2 3` expression is compiled into the `Fadd 2 3` node. Whenever a lazy value needs to be evaluated, the GRIN program will call the `eval` function, which will force the given computation and update the stored value (so that it is not computed twice), or it will just return the value if it is already in weak head normal form. For a partial function call, the GRIN program will construct a `P` node, and call the `apply` function. The number in the `P` node's tag indicates how many arguments are still missing to the given function call. The `apply` function will take a partially applied function (a `P` node), and will apply it to a given argument. The result can be either another partially applied function, or the result of a saturated function call.

The definitions of `eval` and `apply` are uniquely generated for each program by the GRIN back end. As we can see, they are just ordinary GRIN functions, which means the compiler can analyze and optimize them. For a more detailed description, the reader can refer to [5,6].

# 3 Related Work

This section will introduce the reader to the state-of-the-art concerning functional language compiler technologies and whole program optimization. It will compare these systems' main goals, advantages, drawbacks and the techniques they use.

## 3.1 The Glasgow Haskell Compiler

GHC [13] is the de facto Haskell compiler. It is an industrial strength compiler supporting Haskell2010 with a multitude of language extensions. It has full support for multi-threading, asynchronous exception handling, incremental compilation and software transactional memory.

GHC is the most feature-rich stable Haskell compiler. However, its optimizer part is lacking in two respects. Firstly, neither of its intermediate representations (STG and Core) can express laziness explicitly using the syntax of the language. This means, in order to generate optimal machine code, the code generator cannot use only the AST of the program, but also has to rely on the previously calculated strictness analysis result. This makes the code generation phase more complicated. Secondly, GHC only supports optimization on a per-module basis by default, and only optimizes across modules after inlining certain specific functions. This can drastically limit the information available for the optimization passes, hence decreasing their efficiency. The following sections will show alternative compilation techniques to resolve the issues presented above.

## 3.2 Clean compiler

The Clean compiler [25] is also an industrial-grade compiler, supporting concurrency and a multitude of platforms. It uses the abstract ABC machine as it's evaluation model. The ABC machine is a stack machine which uses three different stacks: the Argument stack, the Basic value stack and the Code stack. The Clean compiler performs no optimizations on the ABC machine level, since defining code transformations on a stack-based representation would be quite inconvenient. Instead, the driving design principle behind the ABC machine is that it should be easy to generate native machine code from it. In the present days, this task is often accomplished by LLVM, which not only guarantees performance, but also provides a higher level intermediate representation. Nonetheless, the Clean compiler generates performant code for most major platforms.

The main difference between Clean and Haskell lies in the type systems. Clean uses uniqueness typing, a concept similar to linear typing. A function argument can be marked unique, which means that it will be used only a single time in the function definition. This allows the compiler to generate destructive updates on that argument after it has been used. The efficiency of Clean programs is largely not attributed to code optimizations, but rather to the fact that the programmer writes mutable code to begin with. Uniqueness typing introduces controlled mutability which can highly increase the efficiency of Clean programs.

## 3.3 GRIN

Graph Reduction Intermediate Notation is an intermediate representation for lazy[1] functional languages. Due to its simplicity and high expressive power, it was utilized by several compiler back ends.

### 3.3.1 Boquist

The original GRIN framework was developed by U. Boquist, and first described in the article [6], then in his PhD thesis [5]. This version of GRIN used the Chalmers Haskell-B Compiler [2] as its front end and RISC as its back end. The main focus of the entire framework is to produce highly efficient machine code from high-level lazy functional programs through a series of optimizing code transformations. At that time, Boquist's implementation of GRIN already compared favorably to the existing Glasgow Haskell Compiler of version 4.01.

The language itself has very simple syntax and semantics, and is capable of explicitly expressing laziness. It only has very few built-in instructions (`store`, `fetch` and `update`) which can be interpreted in two ways. Firstly, they can be seen as simple heap operations; secondly, they can represent graph reduction semantics [24]. For example, we can imagine `store` creating a new node, and `update` reducing those nodes.

GRIN also supports whole program optimization. Whole program optimization is a compiler optimization technique that uses information regarding the entire program instead of localizing the optimizations to functions or translation units. One of the most important whole program analyses used by the framework is the heap-points-to analysis, a variation of Andersen's pointer analysis [1].

### 3.3.2 UHC

The Utrecht Haskell Compiler [10] is a completely standalone Haskell compiler with its own front end. The main idea behind UHC is to use attribute grammars to handle the ever-growing complexity of compiler construction in an easily manageable way. Mainly, the compiler is being used for education, since utilizing a custom system, the programming environment can be fine-tuned for the students, and the error messages can be made more understandable.

UHC also uses GRIN as its IR for its back-end part, however the main focus has diverted from low level efficiency, and broadened to the spectrum of the entire compiler framework. It also extended the original IR with synchronous exception handling by introducing new syntactic constructs for `try/catch` blocks [11]. Also, UHC can generate code for many different targets including LLVM [17], .Net, JVM and JavaScript.

[1] Strict semantics can be expressed as well.

### 3.3.3 JHC

JHC [15] is another complete compiler framework for Haskell, developed by John Meacham. JHC's goal is to generate not only efficient, but also very compact code without the need of any runtime. The generated code only has to rely on certain system calls. JHC also has its own front end and back end just like UHC, but they serve different purposes.

The front end of JHC uses a very elaborate type system called the pure type system [4,30]. In theory, the pure type system can be seen as a generalization of the lambda cube [3], in practice it behaves similarly to the Glasgow Haskell Compiler's Core representation. For example, similar transformations can be implemented on them.

For its intermediate representation, JHC uses an alternate version of GRIN. Meacham made several modifications to the original specification of GRIN. Some of the most relevant additions are mutable variables, memory regions (heap and stack) and throw-only IO exceptions. JHC's exceptions are rather simple compared to those of UHC, since they can only be thrown, but never caught.

JHC generates completely portable ISO C, which for instance was used to implement a NetBSD sound driver in high-level Haskell [21].

### 3.3.4 LHC

The LLVM Haskell Compiler [9] is a Haskell compiler made from reusable libraries using JHC-style GRIN as its intermediate representation. As its name suggests, it generates LLVM IR code from the intermediate GRIN.

## 3.4 Other Intermediate Representations

GRIN is not the only IR available for functional languages. In fact, it is not even the most advanced one. Other representations can either be structurally different or can have different expressive power. For example GRIN and LLVM are both structurally and expressively different representations, because GRIN has monadic structure, while LLVM uses basic blocks, and while GRIN has sum types, LLVM has vector instructions. In general, different design choices can open up different optimization opportunities.

### 3.4.1 MLton

MLton [32] is a widely used Standard ML compiler. It also uses whole program optimization, and focuses on efficiency.

MLton has a wide array of distinct intermediate representations, each serving a different purpose. Each IR can express a certain aspect of the language more precisely than the others, allowing for more convenient implementation of the respective analyses and transformations. They use a technique similar to defunctionalization called 0CFA, a higher-order control flow analysis. This method

serves a very similar purpose to defunctionalization, but instead of following function tags, it tracks function closures. Also, 0CFA can be generalized to k-CFA, where $k$ represents the number of different contexts the analysis distinguishes. The variant used by MLton distinguishes zero different contexts, meaning it is a *context insensitive* analysis. The main advantage of this technique is that it can be applied to higher-order languages as well.

Furthermore, MLton supports contification [12], a control flow based transformation, which turns function calls into continuations. This can expose a lot of additional control flow information, allowing for a broad range of optimizations such as tail recursive function call optimization.

As for its back end, MLton has its own native code generator, but it can also generate LLVM IR code [18].

### 3.4.2 Intel Research Compiler

The Intel Labs Haskell Research Compiler [19] was a result of a long running research project of Intel focusing on functional language compilation. The project's main goal was to generate very efficient code for numerical computations utilizing whole program optimization.

The compiler reused the front end part of GHC, and worked with the external Core representation provided by it. Its optimizer part was written in MLton and was a general purpose compiler back end for strict functional languages. Differently from GRIN, it used basic blocks which can open up a whole spectrum of new optimization opportunities. Furthermore, instead of whole program defunctionalization (the generation of global `eval`), their compiler used function pointers and data-flow analysis techniques to globally analyze the program. They also supported synchronous exceptions and multi-threading.

One of their most relevant optimizations was the SIMD vectorization pass [23]. Using this optimization, they could transform sequential programs into vectorized ones. In conjunction with their other optimizations, they achieved performance metrics comparable to native C [22].

# 4 Compiling to LLVM

LLVM is a collection of compiler technologies consisting of an intermediate representation called the LLVM IR, a modularly built compiler framework and many other tools built on these technologies. This section discusses the benefits and challenges of compiling GRIN to LLVM.

## 4.1 Benefits and Challenges

The main advantage LLVM has over other CISC and RISC based languages lies in its modular design and library based structure. The compiler framework built around LLVM is entirely customizable and can generate highly optimized low level machine code for most architectures. Furthermore, it offers a vast range of tools

and features out of the box, such as different debugging tools or compilation to WebAssembly.

However, compiling unrefined functional code to LLVM does not yield the results one would expect. Since LLVM was mainly designed for imperative languages, functional programs may prove to be difficult to optimize. The reason for this is that functional artifacts or even just the general structuring of functional programs can render conventional optimization techniques useless.

While LLVM acts as a transitional layer between architecture independent, and architecture specific domains, GRIN serves the same purpose for the functional and imperative domains. Figure 4.1 illustrates this domain separation. The purpose of GRIN is to eliminate functional artifacts and restructure functional programs in a way so that they can be efficiently optimized by conventional techniques.

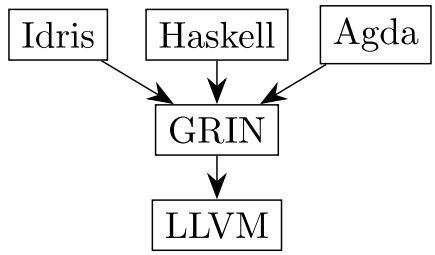



Figure 4.1: Possible representations of different functional languages

The main challenge of compiling GRIN to LLVM has to do with the discrepancy between the respective type systems of these languages: GRIN is untyped, while LLVM has static typing. In order to make compilation to LLVM possible[1], we need a typed representation for GRIN as well. Fortunately, this problem can be circumvented by implementing a type inference algorithm for the language. To achieve this, we can extend an already existing component of the framework, the heap points-to data-flow analysis.

## 4.2 Heap points-to Analysis

Heap points-to analysis (HPT in the following), or pointer analysis is a commonly used data-flow analysis in the context of imperative languages. The result of the analysis contains information about the possible variables or heap locations a given pointer can point to. In the context of GRIN, it is used to determine the type of data constructors (or nodes) a given variable could have been constructed with. The result is a mapping of variables and abstract heap locations to sets of data constructors.

[1]As a matter of fact, compiling untyped GRIN to LLVM *is* possible, since only the registers are statically typed in LLVM, the memory is not. So in principle, if all variables were stored in memory, generating LLVM code from untyped GRIN would be plausible. However, this approach would prove to be very inefficient.

The original version of the analysis presented in [5] and further detailed in [6] only supports node level granularity. This means, that the types of literals are not differentiated, they are unified under a common "basic value" type. Therefore, the analysis cannot be used for type inference as it is. In order to facilitate type inference, HPT has to be extended, so that it propagates type information about literals as well. This can be easily achieved by defining primitive types for the literal values. Using the result of the modified algorithm, we can generate LLVM IR code from GRIN.

However, in some cases the monomorphic type inference algorithm presented above is not sufficient. For example, the Glasgow Haskell Compiler has polymorphic primitive operations. This means, that despite GRIN being a monomorphic language, certain compiler front ends can introduce external polymorphic functions to GRIN programs. To resolve this problem, we have to further extend the heap points-to analysis. The algorithm now needs a table of external functions with their respective type information. These functions *can* be polymorphic, hence they need special treatment during the analysis. When encountering external function applications, the algorithm has to determine the concrete type of the return value based on the possible types of the function arguments[1]. Essentially, it has to fill all the type variables present in the type of the return value with concrete types. This can be achieved by unification. Fortunately, the unification algorithm can be expressed in terms of the same data-flow operations HPT already uses.

### 4.3 Type Information from the Surface Language

Another option would be to use type information provided by the surface language. This approach might seem convenient, but it has three major disadvantages. The first one is that this solution would need to address each front end language separately, since they might have different type systems. Secondly, requiring type information from the front end would rule out dynamically typed languages. Lastly, the surface language's type system tells us about the *semantics* of the program, however we need information about the *data representation* to efficiently analyze, optimize, and generate machine code from GRIN programs. The two concepts might seem familiar at first, but the type-based control flow analysis yields a lot less precise result than the heap-points-to analysis (slightly modified 0-CFA) [27].

In object oriented languages, type-based control flow analysis is sometimes used to make the general pointer analysis more precise. In certain cases, type information can help to filter out impossible cases calculated by the pointer analysis (e.g.: when using interfaces). For functional languages, this approach only works for strict data structures. For example, if we have a strict list, we know that it has been constructed with either `Nil` or `Cons`. However, if the list is lazy, it still might be a thunk referring to any function that returns a list. This means, that in the defunctionalized GRIN program, the list can not only have a `CNil` or a `CCons` tag, but also any `F` tag belonging to a function that returns a list. Consequently, the

[1] This concrete type always exists, since all inputs to the program have concrete types (which are propagated through the program), and we know the entire program at compile time.

set of possible tags for a given lazy type would have to include all those `F` tags as well. This would hinder the type-based analysis considerably inaccurate.

# 5 Dead Code Elimination

Dead code elimination is one of the most well-known compiler optimization techniques. The aim of dead code elimination is to remove certain parts of the program that neither affect its final result nor its side effects. This includes code that can never be executed, and also code which only consists of irrelevant operations on dead variables. Dead code elimination can reduce the size of the input program, as well as increase its execution speed. Furthermore, it can facilitate other optimizing transformation by restructuring the code.

## 5.1 Dead Code Elmination in GRIN

The original GRIN framework has three different type of dead code eliminating transformations. These are dead function elimination, dead variable elimination and dead function paramater elimination. In general, the effectiveness of most optimizations solely depends on the accuracy of the information it has about the program. The more precise information it has, the more agressive it can be. Furthermore, running the same transformation but with additional information available, can often yield more efficient code.

In the original framework, the dead code eliminating transformations were provided only a very rough approximation of the liveness of variables and function parameters. In fact, a variable was deemed dead only if it was never used in the program. As a consequence, the required analyses were really fast, but the transformations themselves were very limited.

## 5.2 Interprocedural Liveness Analysis

In order to improve the effectiveness of dead code elimination, we need more sophisticated data-flow analyses. Liveness analysis is a standard data-flow analysis that determines which variables are live in the program and which ones are not. It is important to note, that even if a variable is used in the program, it does not necessarily mean it is live. See Program code 5.1.

In the first example, we can see a program where the variable `n` is used, it is put into a `CInt` node, but despite this, it is obvious to see that `n` is still dead. Moreover, the liveness analysis can determine this fact just by examining the function body locally. It does not need to analyze any function calls. However, in the second example, we can see a very similar situation, but here `n` is an argument to a function call. To calculate the liveness of `n`, the analysis either has to assume that the arguments of `foo` are always live, or it has to analyze the body of the function. The former decision yields a faster, but less precise *intraprocedural* analysis, the latter results in a bit more costly, but also more accurate *interprocedural* analysis.

```
1 main =
2   n <- pure 5
3   y <- pure (CInt n)
4   pure 0
```

(a) Put into a data constructor

```
1 main =
2   n <- pure 5
3   foo n
4 foo x = pure 0
```

(b) Argument to a function call

Program code 5.1: Examples demonstrating that a used variable can still be dead

By extending the analysis with interprocedural elements, we can obtain quite a good estimate of the live variables in the program, while minimizing the cost of the algorithm. Using the information gathered by the liveness analysis, the original optimizations can remove even more dead code segments.

# 6 Dead Data Elimination

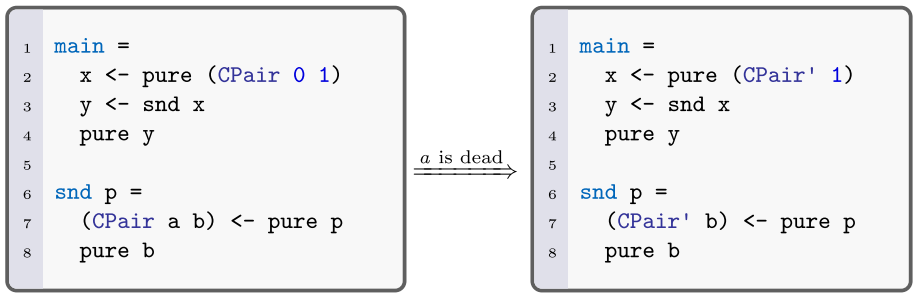
Conventional dead code eliminating optimizations usually only remove statements or expressions from programs; however, *dead data elimination* can transform the underlying data structures themselves. Essentially, it can specialize a certain data structure for a given use-site by removing or transforming unnecessary parts of it. It is a powerful optimization technique that — given the right circumstances — can significantly decrease memory usage and reduce the number of executed heap operations.

Within the framework of GRIN, it was Remi Turk, who presented the initial version of dead data elimination in his master's thesis [31]. His original implementation used intraprocedural analyses and an untyped representation of GRIN. We extended the algorithm with interprocedural analyses, and improved the "dummification" process (see Sections 6.4 and 6.5). In the following we present a high level overview of the original dead data elimination algorithm, as well as detail some of our modifications.

## 6.1 Dead Data Elimination in GRIN

In the context of GRIN, dead data elimination removes dead fields of data constructors (or nodes) for both definition- and use-sites. In the following, we will refer to definition-sites as *producers* and to use-sites as *consumers*. Producers and consumers are in a *many-to-many* relationship with each other. A producer can define a variable used by many consumers, and a consumer can use a variable possibly defined by many producers. It only depends on the control flow of the program. Program code 6.1 illustrates dead data elimination on a very simple example with a single producer and a single consumer.

As we can see, the first component of the pair is never used, so the optimization can safely eliminate the first field of the node. It is important to note, that




(a) Before the transformation

(b) After the transformation

Program code 6.1: A simple example for dead data elimination

the transformation has to remove the dead field for both the producer and the consumer. Furthermore, the name of the node also has to be changed to preserve type correctness, since the transformation is specific to each producer-consumer group. This means, the data constructor `CPair` still exists, and it can be used by other parts of the program, but a new, specialized version is introduced for any optimizable producer-consumer group[1].

Dead data elimination requires a considerable amount of data-flow analyses and possibly multiple transformation passes. First of all, it has to identify potentially removable dead fields of a node. This information can be acquired by running liveness analysis on the program (see Section 5.2). After that, it has to connect producers with consumers by running the *created-by data-flow analysis*. Then it has to group producers together sharing at least one common consumer, and determine whether a given field for a given producer can be removed globally, or just dummified locally. Finally, it has to transform both the producers and the consumers.

## 6.2 Created-by Analysis

The created-by analysis, as its name suggests is responsible for determining the set of producers a given variable-was possibly created by. For our purposes, it is sufficient to track only node valued variables, since these are the only potential candidates for dead data elimination. Analysis example 6.1 demonstrates how the algorithm works on a simple program.

The result of the analysis is a mapping from variable names to set of producers grouped by their tags. For example, we could say that "variable `y` was created by the producer `a` given it was constructed with the `CTrue` tag". Naturally, a variable can be constructed with many different tags, and each tag can have multiple producers. Also, it is important to note that some variables are their own producers. This is

[1] Strictly speaking, a new version is only introduced for each different set of live fields used by producer-consumer groups.

```
1 null xs =
2   y <- case xs of
3     (CNil) ->
4       a <- pure (CTrue)
5       pure a
6     (CCons z zs) ->
7       b <- pure (CFalse)
8       pure b
9   pure y
```

(a) Input program

| Var | Producers |
|---|---|
| xs | $\{CNil[\ldots], CCons[\ldots]\}$[1] |
| a | $\{CTrue[\mathsf{a}]\}$ |
| b | $\{CFalse[\mathsf{b}]\}$ |
| y | $\{CTrue[\mathsf{a}], CFalse[\mathsf{b}]\}$ |

(b) Anyalsis result

Analysis example 6.1: An example demonstrating the created-by analysis

because producers are basically definitions-sites or bindings, identified by the name of the variable on their left-hand sides. However, not all bindings have variables on their left-hand side, and some values may not be bound to variables. Fortunately, this problem can be easily solved by a simple program transformation.

## 6.3 Grouping Producers

On a higher abstraction level, the result of the created-by analysis can be interpreted as a bipartite directed graph between producers and consumers. One group of nodes represents the producers and the other one represents the consumers. A producer is connected to a consumer if and only if the value created by the producer can be consumed by the consumer. Furthermore, each component of the graph corresponds to one producer-consumer group. Each producer inside the group can only create values consumed by the consumers inside the same group, and a similar statement holds for the consumers as well.

## 6.4 Transforming Producers and Consumers

As mentioned earlier, the transformation applied by dead data elimination can be specific for each producer-consumer group, and both the producers and the consumers have to be transformed. Also, the transformation can not always simply remove the dead field of a producer. Take a look at Figure 6.1.

As we can see, producers $P_1$ and $P_2$ share a common consumer $C_2$. Let's assume, that the shared value is a `CPair` node with two fields, and neither $C_1$, nor $C_2$ uses the first field of that node. This means, the first field of the `CPair` node is locally dead for producer $P_1$. Also, suppose that $C_3$ *does* use the first field of that node, meaning it is live for $P_2$, hence it cannot be removed. In this situation, if the transformation were to remove the locally dead field from $P_1$, then it would lead

[1]For the sake of simplicity, we will assume that `xs` was constructed with the `CNil` and `CCons` tags. Also its producers are irrelevant in this example.

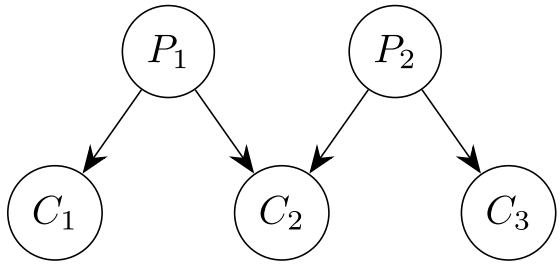



Figure 6.1: Producer-consumer group

to a type mismatch at $C_2$, since $C_2$ would receive two `CPair` nodes with different number of arguments, with possibly different types for their first fields. In order to resolve this issue the transformation has to rename the tag at $P_1$ to `CPair'`, and create new patterns for `CPair'` at $C_1$ and $C_2$ by duplicating and renaming the existing ones for `CPair`. This way, we can avoid potential memory operations at the cost of code duplication.

In fact, even the code duplication can be circumvented by introducing the notion of *basic blocks* to the intermediate representation. Basic blocks allow us to transfer control between different code segments meanwhile maintaining the same local environment (local variables). This means, we can share code between the different alternatives of a case expression. We still need to generate new alternatives (new patterns), but their right-hand sides will be simple jump instructions to the basic blocks of the original alternative's right-hand side.

## 6.5 The `undefined` value

Another option would be to only *dummify* the locally dead fields. In other words, instead of removing the field at the producer and restructuring the consumers, the transformation could simply introduce a dummy value for that field. The dummy value could be any placeholder with the same type as the locally dead field. For instance, it could be any literal of that type. A more sophisticated solution would be to introduce an undefined value. The `undefined` value is a placeholder as well, but it carries much more information. By marking certain values undefined instead of just introducing placeholder literals, we can facilitate other optimizations down the pipeline. However, each `undefined` value has to be explicitly type annotated for the heap points-to analysis to work correctly. Just like the other approach mentioned earlier, this alternative also solves the problem of code duplication at the cost of some modifications to the intermediate representation. Previously we needed structural extensions facilitating code sharing (basic blocks), now we had to introduce a new basic value (typed `undefined`).

# 7 Idris Front End

Currently, our compiler uses the Idris compiler as its front end. The infrastructure can be divided into three components: the front end, that is responsible for generating GRIN IR from the Idris byte code; the optimizer, that applies GRIN-to-GRIN transformations to the GRIN program, possibly improving its performance; and the back end, that compiles the optimized GRIN code into an executable.

## 7.1 Front end

The front end uses the bytecode produced by the Idris compiler to generate the GRIN intermediate representation. The Idris bytecode is generated without any optimizations by the Idris compiler. The code generation from Idris to GRIN is really simple, the difficult part of refining the original program is handled by the optimizer.

## 7.2 Optimizer

The optimization pipeline consists of three stages, as can be seen in Figure 7.1. In the first stage, the optimizer iteratively runs the so-called *regular optimizations*. These are the program transformations described in Urban Boquist's PhD thesis [5]. A given pipeline of these transformations are run until the code reaches a fixed-point, and cannot be optimized any further. This set of transformation are not formally proven to be confluent, so theoretically different pipelines can result in different fixed-points[1]. Furthermore, some of these transformations can work against each other, so a fixed-point may not always exist. In this case, the pipeline can be caught in a loop, where the program returns to the same state over and over again. Fortunately, these loops can be detected, and the transformation pipeline can be terminated.

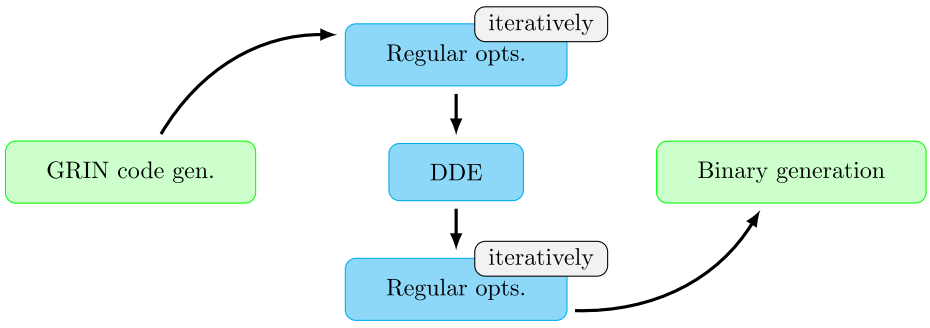



Figure 7.1: Idris compilation pipeline

[1] Although, experiments suggest that these transformations *are* confluent.

Following that, in the second stage, the optimizer runs the *dead data elimination pass*. This pass can be quite demanding on both the memory usage and the execution time due to the several data-flow analyses it requires, and the unrefined implementation. As a consequence, the dead data elimination pass is executed only a single time during the entire optimization process. Since the dead data elimination pass can enable other optimizations, the optimizer runs the regular optimizations a second time right after the DDE pass. This also means, that the liveness analysis could collect more precise information about certain variables, which implies that another pass of DDE could optimize the GRIN program even further. However, in order to run the DDE pass multiple times its implementation has to be improved (see section 9).

### 7.3 Back end

After the optimization process, the optimized GRIN code is passed onto the back end, which then generates an executable using the LLVM compiler framework. The input of the back end consists of the optimized GRIN code, the primitive operations of Idris and a minimal runtime (the latter two are both implemented in C). Currently, the runtime is only responsible for allocating heap memory for the program, and at this point it does not include a garbage collector.

The first task of the back end is to compile the GRIN code into LLVM IR code which is then optimized further by the LLVM Modular Optimizer [34]. Currently, the back end uses the default LLVM optimization pipeline. After that, the optimized LLVM code is compiled into an object file by the LLVM Static Compiler [33]. Finally, Clang links together the object file with the C-implemented primitive operations and the runtime, and generates an executable binary.

## 8 Results

In this section, we present the initial results of our implementation of the GRIN framework. The measurements presented here can only be considered preliminary, given the compiler needs further work to be comparable to systems like the Glasgow Haskell Compiler or the Idris compiler [7]. Nevertheless, these statistics are still relevant, since they provide valuable information about the effectiveness of the optimizer.

### 8.1 Measured programs

The measurements were taken using the Idris front end and LLVM back end of the compiler. Each test program — besides "Length" — was adopted from the book *Type-driven development with Idris* [8] by Edwin Brady. These are small Idris programs demonstrating a certain aspect of the language.

"Length" is an Idris program, calculating the length of a list containing the natural numbers from 1 to 100. This example was mainly constructed to test how

the dead data elimination pass can transform the inner structure of a list into a simple natural number (see Section 6).

## 8.2 Measured metrics

Each test program went through the compilation pipeline described in Section 7, and measurements were taken at certain points during the compilation. The programs were subject to three different types of measurements.

- Static, compile time measurements of the GRIN code.
- Dynamic, runtime measurements of the interpreted GRIN code.
- Dynamic, runtime measurements of the executed binaries.

The compile time measurements were taken during the GRIN optimization passes, after each transformation. The measured metrics were the number of `store`s, `fetch`es and function definitions. These measurements ought to illustrate how the GRIN code becomes more and more efficient during the optimization process. The corresponding diagrams for the static measurements are Diagrams 8.0b to 8.0b. On the horizontal axis, we can see the indices of the transformations in the pipeline, and on the vertical axis, we can see the number of the corresponding syntax tree nodes. Reading these diagram from left to right, we can observe the continuous evolution of the GRIN program throughout the optimization process.

The runtime measurements of the interpreted GRIN programs were taken at three points during the compilation process. First, right after the GRIN code is generated from the Idris byte code; second, after the regular optimization passes; and finally, at the end of the entire optimization pipeline. As can be seen on Figure 7.1, the regular optimizations are run a second time right after the dead data elimination pass. This is because the DDE pass can enable further optimizations. To clarify, the third runtime measurement of the interpreted GRIN program was taken after the second set of regular optimizations. The measured metrics were the number of executed function calls, case pattern matches, `store`s and `fetch`es. The goal of these measurements is to compare the GRIN programs at the beginning and at the end of the optimization pipeline, as well as to evaluate the efficiency of the dead data elimination pass. The corresponding diagrams for these measurement are Diagrams 8.0a to 8.0a.

The runtime measurements of the binaries were taken at the exact same points as the runtime measurements of the interpreted GRIN code. Their goal is similar as well, however they ought to compare the generated binaries instead of the GRIN programs. The measured metrics were the size of the binary, the number of executed user-space instructions, stores, loads, total heap memory usage (in bytes) and execution speed (in milliseconds)[1]. The binaries were generated by the LLVM back end described in Section 7.3 with varying optimization levels for the LLVM Optimizer. The optimization levels are indicated in the corresponding tables: Ta-

[1]The execution speed was measured by averaging the result of 1000 measurements.

bles 8.1 to 8.4. Where the optimization level is not specified, the default, `O0` level was used. As for the LLVM Static Compiler and Clang, the most aggressive, `O3` level was set for all the measurements.

There are also measurements for the binaries generated by the Idris compiler. These were compiled using the highest (`O3`) optimization level and the C back end. For these executables, the size is not included, because Idris compiles a full-fledged runtime system into the binary. Since our Idris back end only has a mnimal runtime yet, the sizes of the binaries are not comparable. However, all other metrics are, because during these measurements, Idris' garbage collector was never triggered. This can be accomplished by configuring the initial size of the heap memory through the runtime system of Idris. This allows us to compare Idris and GRIN binaries despite the *yet* non-implemented garbage collector for GRIN.

## 8.3 Measurement setup

All the measurements were performed on a machine with `Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz` processor and `Ubuntu 18.04 bionic` operating system with `4.15.0-46-generic` kernel. The Idris compiler used by the front-end is of version 1.3.1, and the LLVM used by the back end is of version 7.

The actual commands for the binary generation are detailed in Program code 8.1. That script has two parameters: `N` and `llvm-in`. `N` is the optimization level for the LLVM Optimizer, and `llvm-in` is the LLVM program generated from the optimized GRIN code.

```
opt-7 -ON <llvm-in> -o <llvm-out>
llc-7 -O3 -relocation-model=pic -filetype=obj -o <object-file>
clang-7 -O3 prim_ops.c runtime.c <object-file> -s -o <executable>
```

Program code 8.1: Commands for binary generation

As for the runtime measurements of the binary, we used the `perf` tool, the runtime of Idris and the minimal runtime of GRIN. The `perf` command can be seen in Program code 8.2 which was used to count the number of executed user space instructions, stores, loads and to measure the execution speeds. The runtimes were used to determine the memory usage, and to make sure that Idris' garbage collector is never triggered.

```
perf stat -e cpu/mem-stores/u -e "r81d0:u" -e instructions:u
↪  <executable>
```

Program code 8.2: Command for runtime measurements of the binary

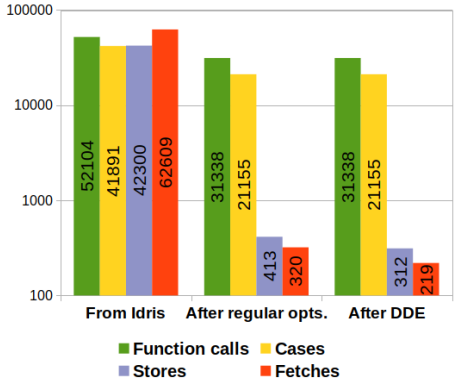
## 8.4 Length

The first thing we can notice on the runtime statistics of the GRIN code, is that the GRIN optimizer significantly reduced the number of heap operations, as well as the number of function calls and case pattern matches. Moreover, the DDE pass could further improve the program's performance by removing additional heap operations.

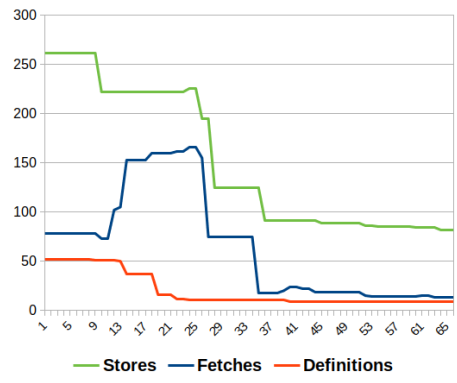
The compile time statistics demonstrate an interesting phenomena. The number of `stores` and function definitions continuously keep decreasing, but at a certain point, the number of `fetches` suddenly increase by a relatively huge margin. This is due to the fact that the optimizer usually performs some preliminary transformations on the GRIN program *before* inlining function definitions. This explains the sudden rise in the number of `fetches` during the early stages of the optimization process. Following that spike, the number of heap operations and function definitions gradually decrease until the program cannot be optimized any further.

Diagram 8.1: Length - GRIN statistics

(a) Runtime




(b) Compile time




The runtime statistics for the executed binary are particularly interesting. First, observing the `O0` statistics, we can see that the regular optimizations substantially reduced the number of executed instructions and memory operations, just as we saw with the interpreted GRIN code. Also, it is interesting to see that the DDE optimized binary did not perform any better than the regularly optimized one; however, its size decreased by more than 20%.

We can also notice the huge memory usage difference between the Idris program and the GRIN programs that were only optimized by LLVM but not by GRIN. This because the rather simple code generation scheme of the Idris front end as discussed in 7.1. However, after running the optimizations, the optimized GRIN programs consume considerably less memory, and have better execution times as well.

It is worth noting that the Idris binary executed significantly more instructions, and performed a lot more stores and loads than the unoptimized GRIN binary, yet

it had a better execution time. The excessive number of memeory operations can be explained by Idris' calling convention. The function arguments are always psuhed onto the stack by the caller, and popped by the callee. This results in a lot of stack memory stores and loads which are reflected in the measurements. However, since the stack memory operations are quite fast, they have no significant impact on the execution times.

As for the high number of executed instructions, we can only hypothesize that it's caused by the Idris runtime system. Idris uses the runtime system to allocate memory through multiple function calls. In GRIN, the memory operations are kind of "inlined" into the generated LLVM code. This might mean that the binaries generated by the Idris compiler could execute a lot more instructions for every memory operation.

Table 8.1: Length - CPU binary statistics

| Stage | Size | Instructions | Stores | Loads | Memory | Time |
|---|---|---|---|---|---|---|
| `idris` | - | 2822725 | 366880 | 1064977 | 9440 | 0.838 |
| `normal-O0` | 23928 | 769588 | 212567 | 233305 | 674080 | 1.993 |
| `normal-O3` | 23928 | 550065 | 160252 | 170202 | 674080 | 1.056 |
| `regular-opt` | 19832 | 257397 | 14848 | 45499 | 8200 | 0.463 |
| `dde-O0` | 15736 | 256062 | 14243 | 45083 | 5776 | 0.525 |
| `dde-O3` | 15736 | 284970 | 33929 | 54555 | 5776 | 0.461 |

Also, it should be pointed out that the aggressively optimized DDE binary performed much worse than the `O0` version. This is because the default optimization pipeline of LLVM is designed for the C and C++ languages. As a consequence, in certain scenarios it may perform poorly for other languages. In the future, we plan to construct a better LLVM optimization pipeline for GRIN.

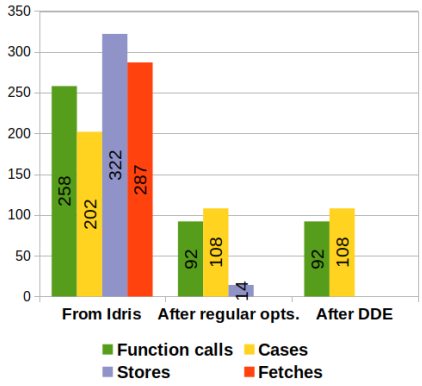
## 8.5 Exact length

For the GRIN statistics of "Exact length", we can draw very similar conclusions as for "Length". However, closely observing the statistics, we can see, that the DDE pass completely eliminated *all* heap operations from the program. In principle, this means, that all the variables can be put into registers during the execution of the program. In practice, some variables will be spilled onto stack, but the heap will never be used.

The binary statistics show that the optimized GRIN programs really do not use any heap memory. As for the other measured metrics, we do not see any major improvements.

Diagram 8.1: Exact length - GRIN statistics

(a) Runtime




(b) Compile time

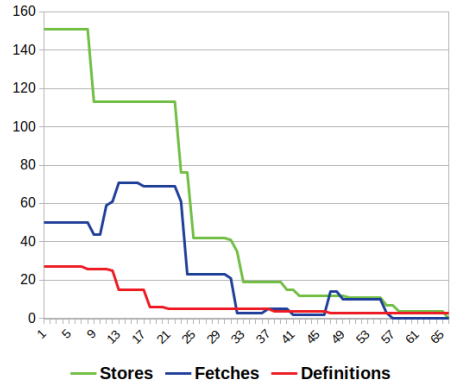



Table 8.2: Exact length - CPU binary statistics

| Stage | Size | Instructions | Stores | Loads | Memory | Time |
|---|---|---|---|---|---|---|
| `idris` | - | 260393 | 23320 | 68334 | 1888 | 0.516 |
| `normal-O0` | 18800 | 188469 | 14852 | 46566 | 4112 | 0.464 |
| `normal-O3` | 14704 | 187380 | 14621 | 46233 | 4112 | 0.455 |
| `regular-opt` | 10608 | 183560 | 13462 | 45214 | 112 | 0.451 |
| `dde-O0` | 10608 | 183413 | 13431 | 45189 | 0 | 0.453 |
| `dde-O3` | 10608 | 183322 | 13430 | 44226 | 0 | 0.448 |

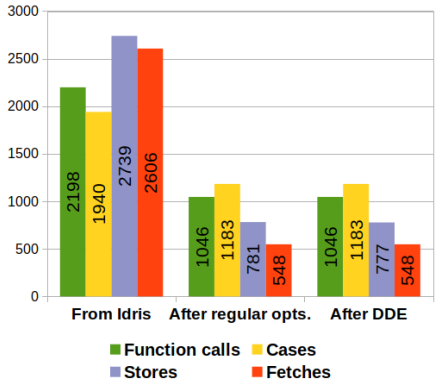
## 8.6 Type level functions

The GRIN statistics for this program may not be particularly interesting, but they demonstrate that the GRIN optimizations work for programs with many type level computations as well.

The binary statistics look promising for "Type level functions". Almost all measured performance metrics are strictly decreasing, which suggests that even the default LLVM optimization pipeline can work for GRIN. Also, the optimized GRIN programs use almost half as much memory as the Idris program.

Diagram 8.1: Type level functions - GRIN statistics

(a) Runtime




(b) Compile time

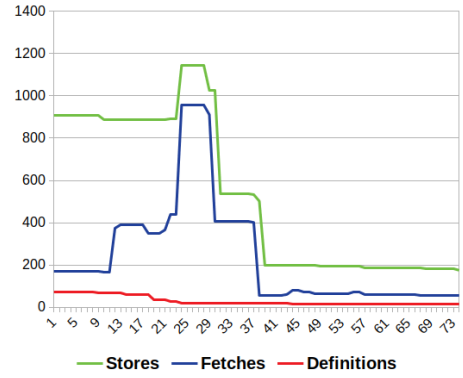



Table 8.3: Type level functions - CPU binary statistics

| Stage | Size | Instructions | Stores | Loads | Memory | Time |
|---|---|---|---|---|---|---|
| `idris` | - | 525596 | 70841 | 158363 | 29816 | 0.637 |
| `normal-O0` | 65128 | 383012 | 49191 | 86754 | 44212 | 0.581 |
| `normal-O3` | 69224 | 377165 | 47556 | 84156 | 44212 | 0.536 |
| `regular-opt` | 36456 | 312122 | 34340 | 71162 | 15412 | 0.516 |
| `dde-O0` | 32360 | 312075 | 34331 | 70530 | 15236 | 0.532 |
| `dde-O3` | 28264 | 309822 | 33943 | 70386 | 15236 | 0.513 |

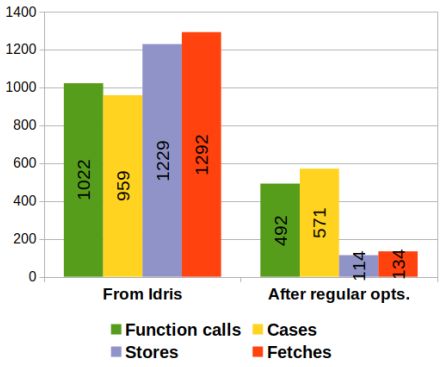
## 8.7 Reverse

Unlike, the previous programs, "Reverse" could not have been optimized by the dead data elimination pass. The pass had no effect on it. Fortunately, the regular optimizations alone could considerably improve both the runtime and compile time metrics of the GRIN code.

The binary statistics are rather promising. The binary size decreased by a substantial margin and the number of executed memory operations has also been reduced by quite a lot. Furthermore, the optimized GRIN programs use less than one third of the memory that the Idris program uses.

Diagram 8.1: Reverse - GRIN statistics

(a) Runtime




(b) Compile time

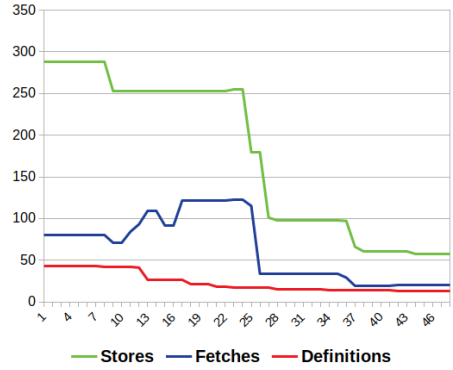



Table 8.4: Reverse - CPU binary statistics

| Stage | Size | Instructions | Stores | Loads | Memory | Speed |
|---|---|---|---|---|---|---|
| `idris` | - | 350215 | 37893 | 101040 | 7656 | 0.576 |
| `normal-O0` | 27112 | 240983 | 25018 | 58253 | 18640 | 0.498 |
| `normal-O3` | 31208 | 236570 | 23808 | 56617 | 18640 | 0.481 |
| `regular-opt-O0` | 14824 | 222085 | 19757 | 53125 | 2384 | 0.467 |
| `regular-opt-O3` | 14824 | 220837 | 19599 | 52827 | 2384 | 0.454 |

## 8.8 General conclusions

In general, the measurements demonstrate that the GRIN optimizer can considerably improve the performance metrics of a given GRIN program. The regular optimizations themselves can usually produce highly efficient programs, however, in certain cases the dead data elimination pass can facilitate additional optimizations, and can further improve the performance.

The results of the binary measurements indicate that the GRIN optimizer performs optimizations orthogonal to the LLVM optimizations. This supports the motivation behind the framework, which is to transform functional programs into a more manageable format for LLVM by eliminating the functional artifacts. This is backed up by the fact, that none of the fully optimized `normal` programs could perform as well as the regularly or DDE optimized ones. Also, it is interesting to see, that there is not much difference between the `O0` and `O3` default LLVM optimization pipelines for GRIN. This motivates further research to find an optimal pipeline for GRIN.

Finally, it is rather surprising to see, that the dead data elimination pass did not really impact the performance metrics of the executed binaries, but it significantly

reduced their size. Firstly, it might be unorthodox to expect speedup from dead code elimination; however, dead data elimination does not only remove unused code, but it transforms the underlying data representations that the program uses. For instance, it could reduce the size of nodes such that they fit into fewer registers, which could help the register allocator, and thus improve the performance of the program. Also, it could remove the elements of a list, leaving only its spine, thus reducing the initial number of heap operations required to allocate the list. Finally, it could help the garbage collector by not allocating unused heap objects as well as reducing the size of the memory map it has to traverse.

Not seeing any performance gains can be explained by the fact, that most of these programs are quite simple, and do not contain any compound data structures. Dead data elimination can shine when a data structure is used in a specific way, so that it can be locally restructured for each use site. However, when applying it to simple programs, we can obtain sub par results.

Nevertheless, the binary size reduction is still notable, and demonstrates that even for simple programs, dead data elimination can still have a significant impact.

# 9 Future Work

Currently, the framework only supports the compilation of Idris, but we are working on supporting Haskell by integrating the Glasgow Haskell Compiler as a new front end. As of right now, the framework *can* generate GRIN IR code from GHC's STG representation, but the generated programs still contain unimplemented primitive operations. The main challenge is to somehow handle these primitive operations. In fact, there is only a small set of primitive operations that cannot be trivially incorporated into the framework, but these might even require extending the GRIN IR with additional built-in instructions.

Besides the addition of built-in instructions, the GRIN intermediate representation can be improved further by introducing the notion of function pointers and basic blocks. Firstly, the original specification of GRIN does not support modular compilation. However, extending the IR with function pointers can help to achieve incremental compilation. Each module could be compiled separately with indirect calls to other modules through function pointers, then by using different data-flow analyses and program transformations, all modules could be optimized together incrementally. In theory, if the entire program is available for analysis at compile time, incremental compilation could produce the same result as whole program compilation. In practice, the LLVM compiler already uses link-time optimizations which implement a very similar idea.

Secondly, the original GRIN IR has a monadic structure which can make it difficult to analyze and transform the control flow of the program. In certain cases it would be beneficial to explicitly transfer control from one program point to another. There two main use cases for this: code sharing (see section 6.4) and explicit tail recursion. Fortunately, replacing the monadic structure of GRIN with basic blocks can resolve both of these issues.

Whole program analysis is a powerful tool for optimizing compilers, but it can be quite demanding on execution time. This being said, there are certain techniques to speed up these analyses. The core of the GRIN optimizer is the heap points-to analysis, an Andersen-style inclusion based pointer analysis [1]. This type of data-flow analysis is very well researched, and there are several ways to improve the algorithm's performance. Firstly, cyclic references could be detected and eliminated between data-flow nodes at runtime. This optimization allows the algorithm to analyze millions of lines of code within seconds [14]. Secondly, the algorithm itself could be parallelized for both CPU and GPU [20], achieving considerable speedups. Furthermore, some alternative algorithms could also be considered. For example, Steengaard's unification based algorithm [28] is a less precise analysis, but it runs in almost linear time. It could be used as a preliminary analysis for some simple transformations at the beginning of the pipeline. Finally, Shapiro's algorithm [26] could act as a compromise between Steengaard's and Andersen's algorithm. In a way, Shapiro's analysis lies somewhere between the other two analyses. It is slower than Steengaard's, but also much more precise; and it is less precise than Andersen's, but also much faster.

Another way to improve on the execution time of the analyses is to drastically improve their implementations. Currently, the analyses are implemented manually as abstract interpretations, and are not optimized further in any way. However, they could reimplemented in well-established, industrial-strength program analysis frameworks. One option would be the Soufflé Datalog compiler [16]. It uses Datalog to define logic-based program analyses, then compiles them to highly-parallelized C++ code. Soufflé facilitates implementing highly scalable data-flow analyses for whole program compilation.

## 10 Conclusions

In this paper we presented a modern look at GRIN, an optimizing functional language back end originally published by Urban Bouquist.

We gave an overview of the GRIN framework, and introduced the reader to the related research on compilers utilizing GRIN and whole program optimization. Then we gave an extension for the heap points-to analysis with more accurate basic value tracking. This allowed for defining a type inference algorithm for the GRIN intermediate representation, which then was used in the implementation of the LLVM back end. Following that, we detailed the dead data elimination pass and the required data-flow analyses, originally published by Remi Turk. We also presented an extension of the dummification transformation which is compatible with the typed representation of GRIN by extending the IR with the `undefined` value. Furthermore, we gave an alternative method for transforming producer-consumer groups by using basic blocks. Our last contribution was the implementation of the Idris front end.

We evaluated our implementation of GRIN using simple Idris programs taken from the book *Type-driven development with Idris* [8] by Edwin Brady. We mea-

sured the optimized GRIN programs, as well as the generated binaries. It is important to note, that the measurements presented in this paper can only be considered preliminary, given the compiler needs further work to be comparable to other systems. Nevertheless, these statistics are still relevant, since they provide valuable information about the effectiveness of the optimizer. The results demonstrate that the GRIN optimizer can significantly improve the performance of GRIN programs. Furthermore, they indicate that the GRIN optimizer performs optimizations orthogonal to the LLVM optimizations, which supports the motivation behind the framework. As for dead data elimination, we found that it can facilitate other transformations during the optimization pipeline, and that it can considerably reduce the size of the generated binaries.

All things considered, the current implementation of GRIN brought adequate results. However, there are still many promising ideas left to research.

# References

[1] Andersen, Lars Ole. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] Augustsson, Lennart. Haskell B. User Manual, 1992. Programming Methodology Group Report, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden.

[3] Barendregt, Henk P. *Lambda calculi with types*. In *Handbook of logic in computer science*, Volume 2, pages 117–309. Oxford: Clarendon Press, 1992.

[4] Berardi, Stefano. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. *Technical report, Carnegie-Mellon University (USA) and Universita di Torino (Ita1y)*, 1988.

[5] Boquist, Urban. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology and Göteborg University, 1999.

[6] Boquist, Urban and Johnsson, Thomas. The GRIN project: A highly optimising back end for lazy functional languages. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, IFL '96, pages 58–84, Berlin, Heidelberg, 1997. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647975.743083.

[7] Brady, Edwin. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. DOI: 10.1017/S095679681300018X.

[8] Brady, Edwin. *Type-driven development with Idris*. Manning Publications Company, 2017.

[9] David Himmelstrup. LLVM Haskell Compiler. URL: http://lhc-compiler.blogspot.com/.

[10] Dijkstra, Atze, Fokker, Jeroen, and Swierstra, S. Doaitse. The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM. DOI: 10.1145/1596638.1596650.

[11] Douma, Christof. Exceptional GRIN. Master's thesis, Utrecht University, Institute of Information and Computing, 2006.

[12] Fluet, Matthew and Weeks, Stephen. Contification Using Dominators. *SIGPLAN Not.*, 36(10):2–13, 2001. DOI: 10.1145/507669.507639.

[13] Hall, Cordelia V., Hammond, Kevin, Partain, Will, Peyton Jones, Simon L., and Wadler, Philip. The Glasgow Haskell Compiler: A Retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 62–71, London, UK, 1993. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647557.729914.

[14] Hardekopf, Ben and Lin, Calvin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *ACM SIGPLAN Notices*, 42(6):290–299, 2007. DOI: 10.1145/1273442.1250767.

[15] John Meacham. Jhc. URL: http://repetae.net/computer/jhc/jhc.shtml.

[16] Jordan, Herbert, Scholz, Bernhard, and Subotić, Pavle. Soufflé: On synthesis of program analyzers. In Chaudhuri, Swarat and Farzan, Azadeh, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[17] Lattner, Chris and Adve, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, 2004.

[18] Leibig, Brian Andrew. An LLVM Back-end for MLton. Technical report, Department of Computer Science, B. Thomas Golisano College of Computing and Information Sciences, 2013. URL: https://www.cs.rit.edu/~mtf/student-resources/20124_leibig_msproject.pdf.

[19] Liu, Hai, Glew, Neal, Petersen, Leaf, and Anderson, Todd A. The Intel Labs Haskell Research Compiler. *SIGPLAN Not.*, 48(12):105–116, 2013. DOI: 10.1145/2578854.2503779.

[20] Mendez-Lojo, Mario and Burtscher, Martin and Pingali, Keshav. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 107–116. ACM, 2012. DOI: 10.1145/2145816.2145831.

[21] Okabe, Kiwamu and Muranushi, Takayuki. Systems Demonstration: Writing NetBSD Sound Drivers in Haskell. *SIGPLAN Not.*, 49(12):77–78, 2014. DOI: 10.1145/2775050.2633370.

[22] Petersen, Leaf, Anderson, Todd A., Liu, Hai, and Glew, Neal. Measuring the Haskell Gap. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, pages 61:61–61:72, New York, NY, USA, 2014. ACM. DOI: 10.1145/2620678.2620685.

[23] Petersen, Leaf, Orchard, Dominic, and Glew, Neal. Automatic SIMD Vectorization for Haskell. *SIGPLAN Not.*, 48(9):25–36, September 2013. DOI: 10.1145/2544174.2500605.

[24] Peyton Jones, Simon. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/.

[25] Plasmeijer, Rinus and Eekelen, Marko Van. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.

[26] Shapiro, Marc and Horwitz, Susan. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1997. DOI: 10.1145/263699.263703.

[27] Shea, Ryan. Alternate control-flow analyses for defunctionalization in the MLton Compiler, 2016.

[28] Steensgaard, Bjarne. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996. DOI: 10.1145/237721.237727.

[29] Sulzmann, Martin, Chakravarty, Manuel MT, Jones, Simon Peyton, and Donnelly, Kevin. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007. DOI: 10.1145/1190315.1190324.

[30] Terlouw, Jan. Een nadere bewijstheoretische analyse van GSTT's, 1989. Manuscript (in Dutch).

[31] Turk, Remi. A modern back-end for a dependently typed language. Master's thesis, Universiteit van Amsterdam, 2010.

[32] Weeks, Stephen. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 1–1, New York, NY, USA, 2006. ACM. DOI: 10.1145/1159876.1159877.

[33] LLVM Static Compiler. URL: https://llvm.org/docs/CommandGuide/llc.html.

[34] Modular LLVM Analyzer and Optimizer. http://llvm.org/docs/CommandGuide/opt.html.

Acta Cybernetica **25** (2022) 877–895.

# Visualisation of Jenkins Pipelines

Ádám Révész[ab] and Norbert Pataki[ac]

**Abstract**

Continuous Integration (CI) is an essential approach in modern software engineering. CI tools help merging the recent commits from the developers, thus the bugs can be realized in an early phase of development and integration hell can be avoided. Jenkins is the most well-known and most widely-used CI tool.

Pipelines become first-class citizen in Jenkins 2. Pipelines consist of stages, such as compiling, building Docker image, integration testing, etc. However, comprehensive Jenkins pipelines are hard to see through and understand. In this paper, we argue for a modern visualisation of Jenkins pipelines. We present our solution for making Jenkins pipelines comprehensible on the dashboard.

**Keywords:** Jenkins, pipeline, visualisation, CI

## 1 Introduction

We think most of us still remember those excuses from the dark ages on failing production releases which sounded like "It has been working on my workstation". Maybe fresh starters also facing these nowadays before getting introduced to the magical realms of continuous integration and continuous delivery tools, the heroes of the software development lifecycle, the saviours of software quality.

As the time being developer community is transforming into DevOps community [10]. A community where developers and administrators live together, learning from each-other and devote themselves to the whole software development lifecycle from the very first stages where design changes being introduced to the production release and monitoring. The common focus is on software quality and productivity. These collaborations made developers and systems engineers to think together on patterns, solutions and tools to streamline development, testing, delivery, logging and monitoring. Mentioning a few of their finest productions are infrastructure as code which enables us to create declarative scripts for infrastructure creation, initialisation and management, able to take under version control. Containerisation

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Hungary
[b]E-mail: adamrevesz@gmail.com, ORCID: 0000-0002-8375-7767
[c]E-mail: patakino@elte.hu, ORCID: 0000-0002-7519-3367

DOI: 10.14232/actacyb.284211

and orchestration platforms enable us to compactly pack applications and define runtime interfaces between them [1]. Remote execution and configuration management systems like Ansible which gives us the ability to declare deployments, configuration upgrades and so on [5]. Logging and monitoring systems like Prometheus and Grafana to for metrics export and visualisation. Elasticsearch, Logstash and Kibana stack for application and system log gathering and analysing [7]. Both of them including alarm systems for multiple level of events to notify teams when a future failure is predicted or in worse case a failure has happened. Remote triggered, most of the time VCS change triggered static analyser tools like SonarCube and CodeChecker to detect code vulnerabilities, smells, anti-patterns, showing test coverage for each version of the product and more. Code community platforms like GitHub where open source developers can work together, share their code and experiences [3].

We could continue this list with so much more excellent tools make our daily work productive, developments well trackable and leverage quality [16]. All the tools listed are very important in the SDLC of services and microservices the most of the industry works with nowadays. All linked by one important type of tools, the CI/CD tools [12].

Continuous Integration (CI) and Continuous Delivery (CD) systems get triggered on version control system changes, run compilation and builds, run tests, passes to static analysers then tag and publishes artifacts with results attached, deploy environments and artifacts into them. Delivers the product to every environment, let it be integration or manual testing, even production servers, app stores, etc. [15]

Jenkins is one of the most popular CI tools of choice [6]. Even though it has recently added pipeline view in the reimagined UI called Blue Ocean, it still misses some points when it comes down to intuition compared to e.g. Microsoft Azure DevOps pipeline view. We think such tool is aimed to both tech members and management ones of a software team. A more intuitive UI could ease the discussions over processes with clients and even inside the team.

In this paper, we are talking about visualisation of CI/CD automated workflows – so called pipelines, discussing visual elements, inspecting and comparing existing solutions of multiple vendors, introducing our take on Jenkins pipeline visualisation concept through our proof of concept implementation.

The rest of this paper is organized as follows. In Section 2, we present the approach of CI and CD. We consider how the CI/CD tools can be utilized by non-tech members in Section 3. The existing solutions are discussed in Section 4. We present our approach in Section 5. We demonstrate our design and legend in Section 6. Finally, this paper concludes in Section 7.

# 2 Continuous Integration and Continuous Delivery

## 2.1 Continuous Integration

Continuous Integration itself is a group of tools and workflows allowing multiple developers/teams to work on multiple features or fixes concurrently on the same product without violating product global quality contracts [14].

The mentioned contracts could be defined by sets semantic rules, test plans, automated tests including unit, property, regression and UI tests, code quality metrics [13].

The evaluation of these compliances takes longer time as the complexity of the product grows. The complexity is a multidimensional measure including (but not limited to)

- number of components (could be separated into artifacts)
- number of features and their tests
- number of supported platforms - environments
- number of concurrent work in progress versions
- number of build, test, analysis (...) tools used for artifact creation validation and verification

Most of the cases, the human resource cost is the largest factor of a software project budget. Making every team member locally run integration tests before commiting changes to the source is inefficient, not mentioning the decrease of developer experience (DX) which causes decreasing productivity of each individual developer, which again shows inefficiency [4].

A continuous integration tool or system satisfies the following requirements:

- isolated workspace
- programmable with one or more scripting language
- has secure storage to store credentials for- and has access to
  - source code repositories
  - artifactory repositories
  - integration environments
  - auto testing systems
  - static analysers
- build and environments matching the supported env(s)
  - build tools

  - platform
  - third party resources
- interface exposing progress
- has logs available for each run
- publishes results
- ensures transparency
- has configurable notification system

In modern terms of continuous integration systems, a workflow executed by CI systems is called pipeline [8].

Pipelines can be defined using user interfaces provided by the CI tool, or editing the pipeline definition itself as code.

Pipelines can be written in script or builder languages e.g. Bash, Maven or Gradle or CI specific languages (mainly domain specific languages over existing languages). Modern CI tools support declarative pipeline definitions which are cleaner, more expressive (e.g. Jenkinsfile). Strategies on where to store pipeline definitions may vary depending on software project size.

## 2.2 Continuous Delivery

Continuous delivery is – nomen est omen – about delivering the product of software project(s) to multiple environments [9]. Continuous delivery is often seen along continuous integration since both of them serving the automation purpose and work with the same exact projects.

In most cases, CI and CD tasks are executed by the same system. The CD often picks up the workflow where CI left but in the majority of the cases there is no clean cut between CI and CD since e.g. in the web service development fields the integration test executed on separate – integration – environment(s), but the deployment of the product to the particular system is the job of CD systems [2].

A continuous delivery tool or system satisfies the same requirements as CI tools excluding the following:

- ability and tools for building the product
- has access and credentials for
  - auto testing systems
  - static analysers

In the same time, has additional requirements:

- has access and credentials for:
  - integration environment

– all other target environments and artifact repositories

Continuous integration and continuous delivery systems are making daily work more effective, give team members flow experience by enabling them to focus better on their tasks instead of manually doing all the work described above or waiting for the results between all the iterations made.

# 3 Further Potentials in CI/CD Tools

Inspecting the core capabilities of the CI tools what can be observed is the target audience is the tech members of software project teams. Tech members are testers, developers, administrators, architects, anyone who's field of work is related to computer science or software engineering.

Having a further consideration these tools could be useful for non-tech members also. There should be a method, an interface of communicating the useful information of the pipelines. This kind of information could be shared on as natural interface as it can be, so the ideal choice is a graphical user interface (GUI), a visualisation of the pipelines.

Of course, we are not inventing the idea of GUIs nor the visualisation of pipelines since there are already popular implementations in the industry but we are proposing a general concept of this kind of visuals, and of course creating a proof of concept implementation for the widely applied open source CI/CD tool, Jenkins.

## 3.1 Essential Definitions on Pipelines

Observing *pipelines*, let them be either CI or CD pipelines the pipeline means a sequence of processes triggered by source code changes in VCS repositories or manually. These processes can be executed on multiple systems – build or release tools – yielding state changes and artifacts if any.

A *pipeline script* is a piece of code interpreted by the observed pipeline executor system – build system. Each step of the *pipeline* is represented as commands in the pipeline script, optionally broken into stage.

A *stage* is a named sequence of commands. Mostly used for leveraging association between its commands, representing the stage as a single step in a higher abstraction layer. Stage as a meta-information can and should be used by visualisation tools. Progress of each stage could be represented on a dynamic visualisation as the ratio between finished and yet to be completed commands of the actual stage.

A *command* can be a pipeline variable declaration and definition (functions included), function and shell invocation.

A *job* marks an execution instance of a pipeline, most commonly identified by the pipeline identifier ($x$) and an incremented integer ($n$), representing the $n$th run of $x$ pipeline.

## 3.2 Useful Information

Let us consider the following team members as non-tech users are interested in pipelines:

- project managers
- delivery managers
- SCRUM masters
- business analysts
- product owners

The assumption regarding the above listed members have been introduced to flow charts, business process management visualisations come natural.

Based on the assumed knowledge of the non-tech members their common requirements against a pipeline visualization could be the display of the following aspects:

- static elements:
  - stages of execution
  - count of steps in each stage
  - average time of each stage execution
  - indicators of manual approval on stages
  - indicators of (quality) gates if any on each stage
  - indicator of relation between stages (defining order)
- dynamic elements:
  - progress of each stage
  - indicators of active stages
  - actual time of execution
  - indicator of successful stages
  - indicators of failed stages
  - indicators of satisfied quality gates
  - indicators of unsatisfied quality gates
  - indicators of manually approved stages
  - indicators of manually disapproved stages
  - indicator of manual abortion
  - indicator of termination

# 4 Existing Solutions

## 4.1 GitLab CI

GitLab is a popular Git server implementation which evolved in the years and now contains multiple collaboration and flow tools. GitLab can be self hosted and offers hosting solution also [17]. It is a popular choice for small projects.

GitLab introduced its own CI tool called GitLab CI. Each project can define declarative pipelines in their source code repository written in YAML.

GitLab CI can define conditions as gates for stages but as the time being this paper written it has not developed manual approval of stages yet.

GitLab CI visual dashboard shows:

- stages
- all steps in stages
- indicators of progress
- indicators of success or failure of steps

This dashboard shows too much detail for a non-tech user even though those tasks can be groupped. Most of them are not interested in this level of details rather just stage level.

## 4.2 Microsoft Azure DevOps Pipelines

Microsoft Azure DevOps Pipelines is a great tool for creating CI/CD pipelines. This product is Azure hosted and has limited resources in free tier. The main reason of being listed discussed this section is the intuitive, well-designed, elegant UI it has. Microsoft clearly shows its experience in making business and development tools. The UI (Figure 1) shows:

- artifacts being deployed
- source revision being used
- all stages
- progress of running stage (Ring 1c, circle progress on the bottom)
- succeeded stage (Ring 1a, green tick and label)
- partly succeeded stage (Ring 1b, red warning icon and label)
- failed stage (Ring 1b - Test, red x and label)
- manual pre-approved stage (Ring 1a, human siluette with tick on the left)
- satisfied precondition of stage (Ring 1a, green gate on the left)
- pending manual approval (Ring 1a Test, blue action button with tick)

Note the UI applies colors on the top of each stage 'cards' indicating the state.

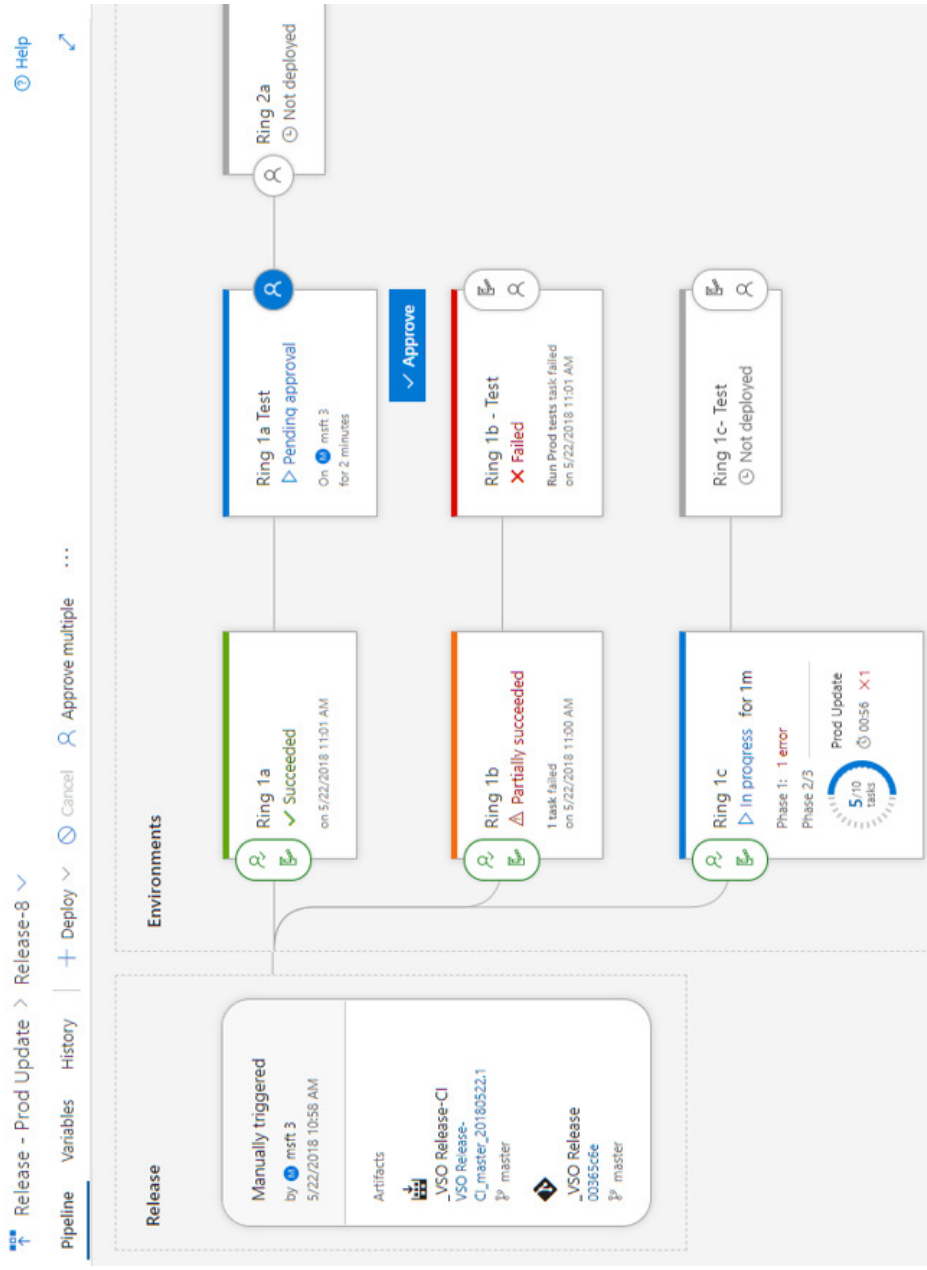



Figure 1: Pipeline visualisation of Microsoft Azure DevOps Pipelines

### 4.3 Jenkins – Blue Ocean

Jenkins is one of the most favourite CI/CD systems, it is available for on premise hosting, has powerful plugin collection and configuration system, battle tested.

Blue Ocean is an installable plugin for Jenkins with a reimagined, modern UI. Blue Ocean also has a web GUI for creating and editing pipelines which results in declarative Jenkins pipeline files.

Blue Ocean has a clear visualisation (Figure 2) of pipelines [11].

- stages
- item count of stages (only in details view)
- indicator of active stages (not shown on image, spinning refresh icon)
- indicator of success-failure of stage (Build)
- indicator of unsatisfied preconditions of stage (Promote-release)
- indicator of manually disapproved stage (UAT)

Note that the indicator of satisfied preconditions and manual approves are missing due to API limitations (discussed in section 5.3.1) e.g., manual testing stage is controlled by our manual approval.

## 5 Our Solution

Our proof of concept implementation is an in-browser application, relying on Jenkins REST API and Jenkins Workflow API. For rendering UI objects, the implementation applies scalable vector graphics (SVG).

### 5.1 Chosing Grounds

In the beginning, we decided between three ways of collecting data of the workflows.

The approach of parsing scripted pipelines has not been used since scripted pipelines have free form, a really flexible structure, hard to evaluate the order of domain-specific language (DSL) calls.

Parsing declarative Jenkins pipelines seemed to be easier due to the strict form, but would be a bigger task to integrate with the DSL. With this solution we would have to depend directly on the pipeline script which is a design flaw for a ui application, or we would have to implement a separate service. The other option is to create a Jenkins plugin which also has to use the DSL lib, but has access to pipeline scripts. Making the plugin would be a big task, itself since we have to persist the metadata somehow, but Jenkins still does not have (or we have not found) notifications on pipeline script updates (maybe SVC triggered pipelines could do it, but that would be real meta pipelining).

The approach chosen is to rely on the Jenkins REST APIs since this approach can have the lightest implementation, allowing us to create an in-browser POC.

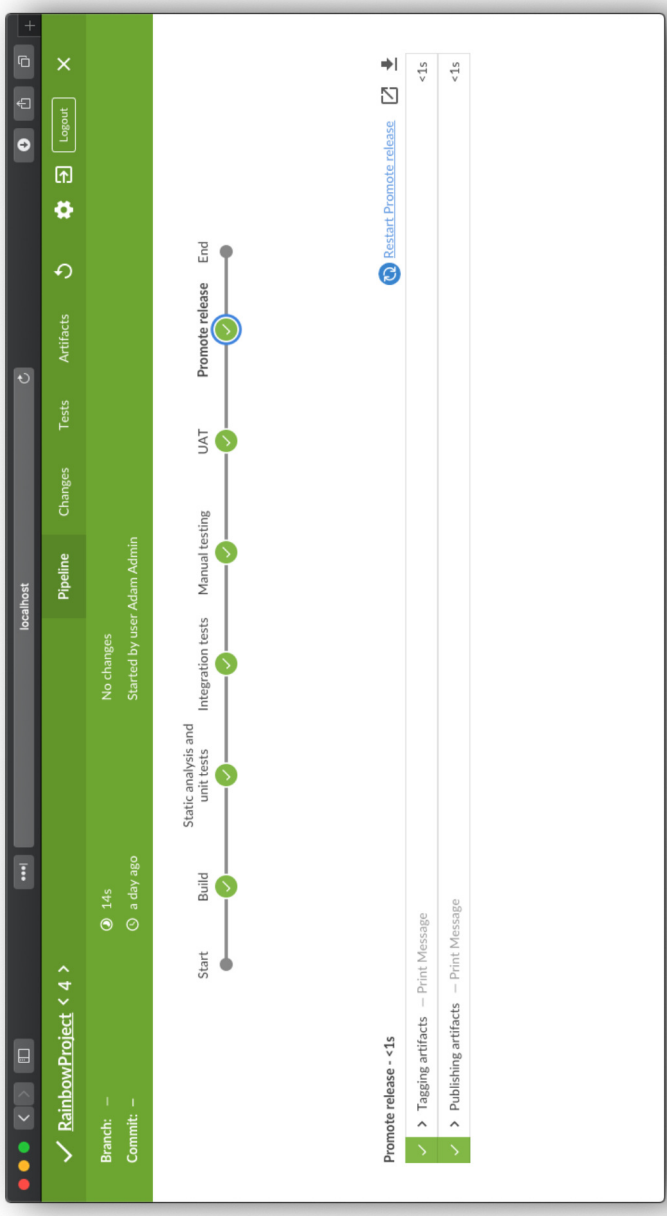



Figure 2: Pipeline visualisation of Jenkins - Blue Ocean

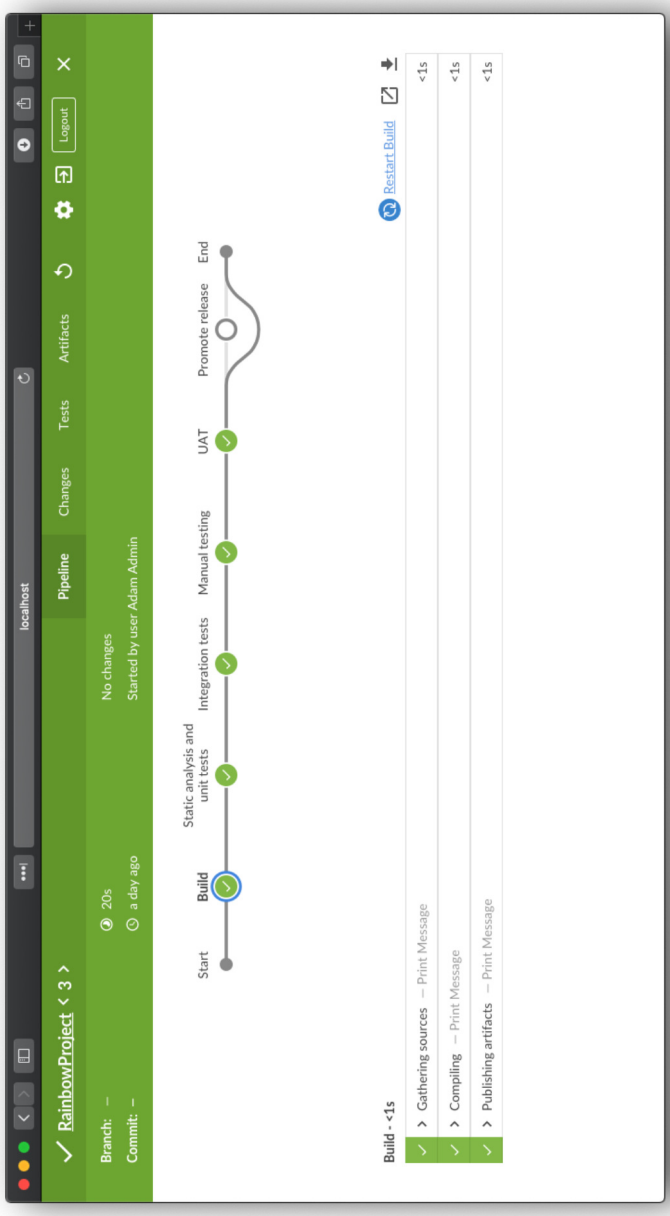



Figure 3: Pipeline visualisation of Jenkins - Blue Ocean


Figure 4: Pipeline visualisation of Jenkins - Blue Ocean

Jenkins REST API itself gives the ability of creating and triggering builds, Jenkins Workflow Plugin (on its new name Pipeline Plugin) introduces the Pipeline interpreter and runtime.

Scalable vector graphics (SVG) is chosen because it has document object model, so easily editable with in-browser technologies like JavaScript dynamically. The format is also a well known image format, suitable for dropping into presentations and printed documents.

## 5.2 Applied Pipeline Script

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Gathering sources'
        echo 'Compiling'
        echo 'Publishing artifacts'
      }
    }
    stage('Code analysis') {
      steps {
        echo 'Running code analysis'
        echo 'Running unit tests'
        echo 'Tagging artifacts'
        echo 'Publishing results'
      }
    }
    stage('Integration tests') {
      steps {
        echo 'Deploying artifacts to integration env'
        echo 'Running regression tests'
        echo 'Running UI tests'
      }
    }
    stage('Manual testing') {
      input {
        message 'Should we deploy?'
        ok 'Yes, go ahead'
      }
      steps {
        echo 'Deploying artifacts to QA environment'
        echo 'Running regression tests'
        echo 'Running UI tests'
        echo 'Manual testing in progress'
```

```
      }
    }
    stage('UAT Deploy') {
      input {
        message 'Should we deploy?'
        ok 'Yes, go ahead'
        submitter "admin"
      }
      steps {
        echo 'Deploying artifacts to UAT environment'
        echo 'Running smoke tests'
        echo 'Manual testing in progress'
      }
    }
    stage('Release Promotion') {
      when {
        not {
          tag 'release-*'
        }
      }
      steps {
        echo 'Tagging artifacts'
        echo 'Publishing artifacts'
      }
    }
  }
}
```

This pipeline represents a common CI/CD pipeline. Since the build and deployment themselves are not relevant in this paper, the execution of tasks are mocked by echo commands.

The subject has three types of stages and their combinations:

- simple stage
- stage with manual approval
- stage with precondition

## 5.3 Exploring Jenkins REST API

Using the original Jenkins UI or Blue Ocean the following important behaviour can be observed:

- Pipeline visualisations do not get updated until a successful build run.

- Failed build runs can apply partial updates.

The static structure of the pipeline is not persisted. Only the runtime yielded metadata gets captured.

### 5.3.1 Limitations

This conclusion is reflected in the API endpoints also. The build pipeline structure is not exposed. Only the results of actual build run and their stages, steps, etc.

The missing indication of satisfied preconditions of stages and manual approves gives reason to inspect the meta data could be obtained about such gates.

Inspecting simple stages in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status
- Current step
- Previous steps

Inspecting manual approved stages in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- The first step is the “Wait for interactive input” step
- Rest of the steps

Inspecting manual disapproved stages (Figure 4) in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- The first and only step is the “Wait for interactive input” step
- Status successful

Conclusion of manual approval required stages is the state of the approval – in case of at least one step is defined in the stage – that the status of the approval can be determined from the API response.

Inspecting stages with satisfied precondition in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status
- Current step
- Previous steps

Inspecting stages with unsatisfied precondition in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status is "NOT_EXECUTED"

Conclusion with satisfied preconditions there is no difference from a simple stage (Figure 2) in the result of the API call. With unsatisfied there is a difference in status (Figure 3) but the reason cannot be retrieved from the response nor the logs. Based on this we have found no point to explicitly mark preconditioned stages because between two runs we cannot determine whether a precondition is removed or satisfied. We have chosen no information over false information. We plan to mark unsatisfied stages with "avoiding" the stage with the line representing the execution.

# 6 User Interface Design

The main goal of the User Interface (Figure 5) is to give a high-level overview of the pipeline (and the progress) at the first glance. The visualisation has to be static, readable without interaction (no clicks for detailed view).

This enables the user to grab the image to documentations, serve as test / deployment evidence.

The card design enables us to attach more information to a stage than those a name and an icon can tell. Attaching the information of corresponding data into a single shape leverages the natural association than a detail view on a different part of the screen or document.

Since the visuals can be dynamically changed during execution, we wanted to add an exclusive start and end point. The first success or failed status stage connected directly to the start point is always the first stage of the execution.

### 6.1 Legend

In the proof of concept implementation, we did not want to introduce third party iconsets, and using characters is way easier. Emojis are part of the Unicode charset handy enough to use.

Legend of the icons (Figure 5):

- Thumbs up (Manual test, in the left circle) indicates manually approved stage
- Thumbs up (Not shown, its place is the same as Thumbs up icon) indicates manually disapproved stage
- Heavy check mark (Manual test, under the stage name) indicates completed stage
- Cross mark (Not shown, its place is the same as Heavy check icon) indicates failed stage
- Raised hand, AKA Stop (Release promotion, under the stage name) indicates pending manual approval
- Hourglass not done (Release promotion, under the stage name) indicates in progress stage

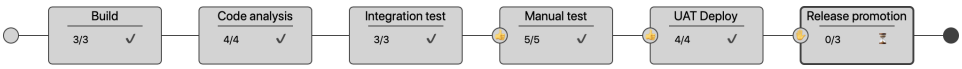



Figure 5: Pipeline visualisation of our POC

## 7 Conclusion

In this article, we have stated general requirements against pipeline visualisations, compared to state of the art implementations.

With our POC implementation, we have discussed approaches of a possible Jenkins pipeline visualisation tool implementations, pointed out reliability issues of runtime evaluated pipeline, and lack of expressiveness of the current APIs (and stored metadata).

This paper is intended to be constructive. We both admit the incredible work of the communities of all the pieces of software we have discussed.

The POC implementation left places to improve:

- using different colors for cards (stages) in different state
- apply design language e.g. flat, material or Blue Ocean

- develop Jenkins plugin, since that is the official way of creating handy Jenkins extensions
- maybe try to add to Blue Ocean as alternative visualisation
- try out SDL level integration

We recommend this paper to CI/CD tool developers, especially the Blue Ocean developer team, we hope they can find useful thoughts in this paper.

## References

[1] Bernstein, David. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014. DOI: 10.1109/MCC.2014.51.

[2] Chen, Lianping. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, Mar 2015. DOI: 10.1109/MS.2015.27.

[3] Dabbish, Laura, Stuart, Colleen, Tsay, Jason, and Herbsleb, Jim. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM. DOI: 10.1145/2145204.2145396.

[4] Fagerholm, Fabian and Münch, Jürgen. Developer experience: Concept and definition. In *Proceedings of the International Conference on Software and System Process*, ICSSP '12, pages 73–77, Piscataway, NJ, USA, 2012. IEEE Press. DOI: 10.1109/ICSSP.2012.6225984.

[5] Hochstein, Lorin and Moser, Rene. *Ansible: Up and Running Automating Configuration Management and Deployment the Easy Way.* O'Reilly Media, Inc., 2nd edition, 2017.

[6] Jenkins. https://jenkins.io/.

[7] Lahmadi, Abdelkader and Beck, Frédéric. Powering Monitoring Analytics with ELK stack. In *9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015)*, 2015. URL: https://hal.inria.fr/hal-01212015/file/slides-ELK.pdf.

[8] Lehtonen, Timo, Suonsyrjä, Sampo, Kilamo, Terhi, and Mikkonen, Tommi. Defining metrics for continuous delivery and deployment pipeline. In Nummedennmaa, Jyrki, Sievi-Korte, Outi, and Mäkinen, Erkki, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST)*, number 1525 in CEUR Workshop Proceedings, pages 16–30, Aachen, 2015. URL: http://ceur-ws.org/Vol-1525/paper-02.

[9] Leppänen, Marko, Mäkinen, Simo, Pagels, Max, Eloranta, Veli-Pekka, Itkonen, Juha, Mäntylä, Mika V., and Männistö, Tomi. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, Mar 2015. DOI: 10.1109/MS.2015.50.

[10] Lwakatare, Lucy Ellen, Kuvaja, Pasi, and Oivo, Markku. Dimensions of DevOps. In Lassenius, Casper, Dingsøyr, Torgeir, and Paasivaara, Maria, editors, *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings*, pages 212–217. Springer International Publishing, Cham, 2015. DOI: 10.1007/978-3-319-18612-2_19.

[11] Pathania, Nikhil. *Declarative Pipeline Development Tools*. In *Beginning Jenkins Blue Ocean: Create Elegant Pipelines With Ease*, pages 191–209. Apress, Berkeley, CA, 2019. DOI: 10.1007/978-1-4842-4158-5_5.

[12] Révész, Ádám and Pataki, Norbert. Integration heaven of nanoservices. In *Proceedings of the 21 th International Multi-Conference INFORMATION SOCIETY, IS'2018*, Volume Volume G: Collaboration, Software and Services in Information Society, pages 43–46, 2018. URL: http://library.ijs.si/Stacks/Proceedings/InformationSociety/2018/IS2018_Volume_G%20-%20CSS.pdf.

[13] Roche, James. Adopting DevOps practices in quality assurance. *Commun. ACM*, 56(11):38–43, November 2013. DOI: 10.1145/2524713.2524721.

[14] Schaefer, Andreas, Reichenbach, Marc, and Fey, Dietmar. Continuous integration and automation for DevOps. In Kim, Kon Haeng, Ao, Sio-Iong, and Rieger, B. Burghard, editors, *IAENG Transactions on Engineering Technologies: Special Edition of the World Congress on Engineering and Computer Science 2011*, pages 345–358. Springer Netherlands, Dordrecht, 2013. DOI: 10.1007/978-94-007-4786-9_28.

[15] Soltesz, Stephen, Pötzl, Herbert, Fiuczynski, Marc E., Bavier, Andy, and Peterson, Larry. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007. DOI: 10.1145/1272998.1273025.

[16] Steingartner, William, Perháč, Ján, and Biliňski, Alexander. A visualizing tool for graduate course: Semantics of programming languages. *IPSI BgD Transactions on Internet Research*, 15(2):52–58, 2019.

[17] van Baarsen, Jeroen. *GitLab Cookbook*. Packt Publishing, 2014.

Acta Cybernetica **25** (2022) 897–908.

# Instantiation of Java Generics

Péter Soha[ab] and Norbert Pataki[ac]

**Abstract**

Type parametrization is an essential construct in modern programming languages. On one hand, Java offers generics, on the other hand, C++ provides templates for highly reusable code. The mechanism between these constructs differs and affects usage and runtime performance, as well. Java uses type erasure, C++ deals with instantiations.

In this paper, we argue for an approach in Java which is similar to C++ template construct. We evaluate the runtime performance of instantiated code and we present our tool which is able to use Java generics as templates. This tool generates Java source code. We present how this approach improves the usage of Java generics.

**Keywords:** Java, generic, instantiation, template

## 1 Introduction

Nowadays we can choose from many different programming paradigms and languages and all have unique advantages and disadvantages. We have to think different when programming in *Java* or *Clean*, when we use the *object-oriented* or the *functional* paradigm. Sometimes we wish that some elements of a language would be supported by another language though. One of these useful tools is the construct of reusable and parametrizable code which significantly reduces the repetition of the code. For this, the *Java* offers the *generics* that uses a runtime polymorphic solution with some transformations at compilation time. On the other hand, the *C++* language provides the *templates*.

Java is considered as a verbose language, therefore Java source typically contains boilerplate code [10]. To overcode the boilerplate, many libraries have been developed, such as Project Lombok [1]. Lombok decreases the quantity of boilerplate code by generating Java code from annotations. However, this solution is not able to generate similar new classes from templates.

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: sohaur@inf.elte.hu, ORCID: 0000-0003-1556-8267
[c]E-mail: patakino@inf.elte.hu, ORCID: 0000-0002-7519-3367

DOI: 10.14232/actacyb.284073

The generics in Java fit into the object-oriented realm, but they have runtime overhead that can be reduced with compile-time instantiation. The required information can be found in the source, therefore performance can be improved without any limitation.

In this paper, we analyze how the template mechanism can be attempered in the Java programming language. We take into consideration how different languages provide type parametrization. Earlier, we have proposed an alternative syntax for instantiable templates for improved runtime performance [12]. At this time, we deal with standard Java code. For keeping the improved runtime performance, we developed a tool which aims at the instantiation of Java generics. Our tool works in the Java realm, therefore no external dependency is taken advantage of. We present our tools and evaluate this approach.

The rest of this paper is organized as follows. In Section 2, we present the different approaches related to type parametrization. After, we briefly present our existing solution in Section 3. In Section 4, we present our new approach for the instantiation of generics. Further options are presented in Section 5. In Section 6, we evaluate the performance of our method. In Section 7, we briefly present our further aims. Finally, this paper is concluded in Section 8.

# 2 Theoretical Background

## 2.1 About Java Generics

The constructs of generic programming paradigm were introduced in Java 1.5 in 2004 [2]. The main idea behind this step was the support of Java programmers with a tool to avoid duplications, and help to write type safe code which is abstract enough to fit as many situations as possible without any modifications [6]. To reach this, all generic code has to contain a type variable section named *parameter list* where the programmer can declare the usable types. This construct has a considerable merit. Since *Generics* applies dynamically typed, a class or function can be used with totally different types without recompile the code but keep type safety. To reach that, *Java* offers the runtime polymorphism, which means the following:
If `X` is a subtype of `T`, every occurance of `T` can be replaced with the objects of `X`. This makes the usage *Generics* versatile but it has its own cost [5].

Restrictions on the type parameters require bounded generic type parameters. If one defines an upper bound for a generic parameter, only its subclasses can be used as generic argument type.

Because of the *type erasure*, we have to deal with the some factors which can affect the performance:

- At compile time, all type parameters will be deleted and replaced with their first bound or `Object` if it is unbounded [15].
- To ensure type safety, *JVM* will generate type casts.

- To preserve polymorphism, bridge methods will be generated as well.

And exactly here is the weakness of the generics. The Java compiler has to modify the written code and insert runtime parts which can significally decrease the speed and effectiveness and may increase the heap memory consumption. Because of the dynamic typing, the construction must be as abstract as possible even if it is not necessary at all. As we showed, sometimes a quazi-statically typed solution can be faster. Moreover, subtype checking is quite difficult in Java [7].

Listing 1: Java generics example

```
1  public class Generic<T> {
2      private T element;
3
4      public Generic(T value) {
5          element=value;
6      }
7
8      public T get() {
9          return element;
10     }
11 }
```

## 2.2 Templates in C++

C++ provides templates for efficient type parametrization [4]. A template is a code snippet that is parametrized and the C++ compiler instantiates with different arguments at compilation time. Let us consider the following example that is quite similar to the previous one:

Listing 2: C++ template example

```
1  template <class T>
2  class Template {
3  private:
4      T element;
5
6  public:
7      Template( const T& t ): element( t ) { }
8
9      const T& get() const { return element; }
10 };
```

The compiler cannot instantiate the template and cannot generate corresponding low-level code unless the template argument is known, thus the template itself is not compilable code from the view of typical C++ compilers (e.g. g++, clang++).

The compiler generates specific code when the template is instantiated. For instance, in case of `Template<int>`, the compiler generates code from the template by substituted `T` with `int`. This construct enables to instantiate templates with arbitrary, previously unknown classes. The compiler is aware of the called functions related to the template parameter, so it can optimize many calls [9]. Moreover, type safety is an essential aspect of templates. On the other hand, code bloat of binary code may appear and template instantiation increases compilation time.

C++ provides function templates and class templates [13]. However, a template can be parametrized not only types but integral constant values, pointers, pointer-to-members, etc. If the compiler does know what the argument is, it is able to generate code. However, string literals are not supported to be template arguments [14].

C++ templates offer an interesting approach. In case of class templates, one can write special implementation for specific class parameters with partial and full specializations. Utilization of this possibility leads us to C++'s template metaprogramming feature that is a Turing-complete subset of C++ [11]. Metaprograms are beneficial since they can speed-up the execution time, enable the development of active libraries that make decisions at compilation time and evaluate compile-time asserts.

# 3 Previous Works

To create templates in Java, we investigated two different ways to instantiate a generic class. In our previous paper, we offered a new keyword and a different class structure to transform a generic class to a template one called *Java Template* [12]. This constructed originated from the C++ templates. It was quite useful and comfortable but already had limitations. In this case, we had to manually rewrite the existing code and declare the template variables. The work was getting more complex when we tried to transform some of the standard containers, and that was the inspiration to try a totally different way and instead of transforming code just create a tool which can work with pure *Generics* without any structural modification (later we will see that it is impossible because of the context-sensitive parts).

First of all we summarize the results of the *Java Templates* and after all we show how can we instantiate the generic classes directly.

## 3.1 Templates and Packages

The *Java Template* is a very similar class construct to C++ templates, but mixing the benefits of the *Generics*. At the beginning of the class definition, there is the `template` keyword followed by the identifiers of type parameters. To insure the instantiation, we restricted the following:

- If `T` is primitive type or literal of primitive types (String literals inclusive) there is no limitation.

- If `T` is an object type, we recommend to use the fully qualified name to avoid any importing issue.

At this point, we have some other requirements by Java that we have to meet. Every Java class has its own and unique identifier called *fully qualified name*. This is a composition of the name of the class and its place of the package hierarchy. The Java compiler does not allow the programmer to create two different classes in the same package with the same name. And well, if we use generics, we have to create only one class to many types, but with templates (since it is statically typed) every single type needs its own instance. To solve this issue, we decided that the package which contains the template instance will build up from the actual parameter types. For this, we made some restrictions:

- If `T` is a primitive type, or a literal of primitve types (String literals included) we add a prefix to it.
- If `T` is an object type, we use the fully qualified name, by escaping the separators with backslash.
- All package names must be able to generate with only of the previous two rules.

To avoid any OS specific issue with package names, the maximum length should not exceed 260 characters.

### 3.2 Instantiation

The *Java Templates* have a special way to instantiation since this construct is not the part of Java. For this, we developed a tool, which can tokenize the source and turn it into a standard C++ macro. Considering that the macro is a low-level language feature, to guarantee the success, need some restrictions:

- The number of declared type parameters at most the number of given parameters.
- For object types, the *fully qualified name* is preferred.
- A formal variable name can be replaced with a given literal if, and only if the specific variable name is only occurs at right hand side of an expression or where literals are allowed by Java.

## 4 New Results

Since the transformation and templatize not really decent when the class structure getting more complex, we had to have find another way. The new idea was that instead of creating a new language feature, and depend on external tools (especially `g++` compiler for the *Java Templates*) we shall use only what Java gives us. In our

new tool, we totally left the `g++` and other external dependecies and build up the preprocessing with pure Java. In this version, we introduced a *typetable* in the lexer tool, which acts as a field memory and store the formal and actual parameters in an associative container (list of key-value pairs). This makes the identification and replace lot easier and let us use this information in the future. The main steps now are the following:

- Load the source and localize the generic type declarations.
- Parse the type placeholders and store them in the *typetable*. To ensure correct run, we made a check to determine that the tool get enough arguments, because as we discussed, the number of actual types must be greater than or equal to the placeholders. In this step, the generic bounds are irrelevant because they will be replaced with an exact type, which will be an upper bound of the acceptable object at that point (this can be possible because of the Liskov substitution principle).
- Assemble the package of the class. This step is the first which clearly gains advantage by using the *typetable*. To create the package, we read the values one-by-one, and concatenate to the package name. To meet the rules of package name declared in Java, we have to replace the dot character to backslash when the current type is an object. For this step, we implement a minor feature as the support of generic type arguments. To reach this, we treat the characters of the *diamond operator* as dot, and also escape it with backslash.

With these three steps, we get the same result as with *Java Templates* which needed six steps for this. In the next sections, we show small examples to both methods and compare them.

## 4.1 Examples

First, we create a basic implementation of a `Stack` with the `Java Templates`. To give information to the lexer tool, we had to use the `template` keyword to declare the formal parameters. Since we are not restricted by the Java grammatic rules, we can use arrays typed by placeholders because the final type will be known at last before compile time.

Listing 3: Java template example

```
1 template(T)
2 class Stack {
3     private T[] elements = new T[10];
4     private int c = 0;
5     // ...
6     public void push(T item) { elements[c++] = item; }
7     public T pop() { return elements[--c]; }
8 }
```

Now, the same stack implemented with the toolkit of *Java Generics*. Since our improved tool can work with standardized generic classes, we do not need any special keyword or language feature, because we can extract all necessary information right from the source. However (as we discussed this problem in the next section) this is a special example. In this case, we have only one `Object` array (which the only possibility because the grammatic rules), but we could prepare our tool to handle this, since there are no ambiguous types. Note, if we want to declare another `Object` array, we may run into anomalies because all these will be replaced with a specific type which given as argument.

Listing 4: Java generic class example

```
1 class Stack<T> {
2     private Object[] elements = new Object[10];
3     private int c = 0;
4     // ...
5     public void push(T item) { elements[c++] = item; }
6     public T pop() { return (T)elements[--c]; }
7 }
```

Finally, both versions of our tool will produce the exactly same Java class. This class is now statically typed (as much as Java allows it), and as one may see in section 6, in some cases, the speed-up is considerable.

Listing 5: Generated Java class

```
1 class Stack {
2     private int[] elements = new int[10];
3     private int c = 0;
4     //...
5     public void push(int item) { elements[c++]=item; }
6     public int pop() { return elements[--c]; }
7 }
```

## 4.2 Compilation with our Tool

We demonstrate the precompiling process of the generic with our tool. A straightforward generic class is `Pair` that we use for presentation.

In this example, the two generic parameters are K and V which denote the key and value types. For instantiation, the tool requires the following arguments in order:

- Tha Java source file contains the implementation of `Pair` generic class.
- Restriction for this step that the file may contain only one top level class which has to match the name of type.

- The actual type of `K` which if it is an object type then it must be the fully qualified name.
- The actual type of `V` that has the same condition as `K`.

Listing 6: Java Pair generic class example

```
1  public class Pair<K, V> {
2      private K key;
3      private V value;
4
5      public K getKey() { return key; }
6      public V getValue() { return value; }
7
8      public void setKey(K _key) { key = _key; }
9      public void setValue(V _value) {
10         value = _value;
11     }
12 }
```

Now let the actual parameters are `int` and `boolean`, therefore we use the hereinafter command for the instantiation:

```
$java Tool Pair.java int boolean
```

After compilation, we have the instantiated Pair class in a unique path which ensures to avoid the ambiguous references. This class contains the following:

Listing 7: Generated class example

```
1  public class Pair {
2      private int key;
3      private boolean value;
4
5      public int getKey() { return key; }
6      public boolean getValue() { return value; }
7
8      public void setKey(int _key) { key = _key; }
9      public void setValue(boolean _value) {
10         value = _value;
11     }
12 }
```

This tool is now a standalone component of the building process right before the compilation. We are working on an improved integration of compilation task.

This tool at the moment has two major deficiencies. The first one is the requirement of explicit enumeration of all actual types, although if we want to replace for example the `T extends Number` parameter, it shall be guessed and replace `T` with `Number`. This feature is useful every time when the bound is a class/typename, so

giving command line arguments is necessary only when we want to use primitives or the bound is an interface. The second one is a more serious limitation, since only one class can be given to the tool but in enterprise environment there are hundreds of classes in every single project and instantiate one by one requires countless hours. So we want to introduce a JSON-like format which contains cases of every single preprocessing task. A task describes the name of the file which will be instantiated and the actual type parameters.

# 5 Improvements

Although the new methods are really beneficial, we find some questions which still waiting for to being aswered. One of them is the problem of the ambiguous field typing. Let us suppose that we created a generic container class that uses an `Object` array to store elements. Now we declare another `Object` array for reasons. If we use the new tool, the clear way is to replace the storage array's type from `Object` to the given one (let it be now `int`). But we also have the other array named `otherarray` which has a different (but not less important) functionality, and it must remain `Object`. In this case, we have the following possible solutions:

1. Replace `Object` with `int`: In this case, the functionality of `otherarray` will be damaged, since the role of the `Object` array is context-sensitive.

2. Train the tool to identify the possible fields: This way is significantly more complex because of the context-sensitive grammar rules and the chance of mistakes even more higher than the previous solution.

3. Pass the chosen identifiers as tool arguments: Now, we can decide in every case whether the current field is modifiable. Although from the point of the input this is one of the best choice, with the extra arguments can make the usage of the tool more complicated.

4. In Java, one can use annotations for the member declarations. Members can be distinguished by their annotations.

# 6 Measurements

In this section, we present the performance of our solution. We focus on the runtime performance because this property is more important than the duration of compilation time. The preprocessor has I/O-intensive tasks, so its performance depends on the storage device [3]. Moreover, a compiler support approach would be more effective in which the instantiation is executed on constructed abstract syntax tree.

We have evaluated how the proposed approach affects the runtime. We have started a cloud-based virtual machine with Ubuntu 16.04 LTS operating system

image and Java 8 JVM installed. We evaluate two different scenarios with high number of test cases. We use our stack data structure implementations.

The first scenario is using stack that contains integers. The generic implementation must be used with `Integer`s, the instantiated generic version can be instantiated with `int`. This approach avoids the autoboxing between `int` and `Integer` and overhead of many memory allocations can be eliminated.

We have instantiated the stack with `Integer` in the second scenario. In this case, the template and generic parameter is exactly the same. However, the template stack itself knows that it contains `Integer`, not `Object`, so less runtime validations are needed in this case, as well.

Our approach performed better in both scenarios. The performance is improved significantly in the first scenario. The average running time of the long-term performance test has beeen reduced to 2.63% of the generic approach with our template mechanism. We measure this speed-up when the size of stack was 8000000. We fulfilled the stack with 8000000 `push` operations and after we used `pop` functions until the stack becomes empty. High amount of dynamic memory allocation and autoboxing conversion can be avoided with instantiated generic in this case. The results were rather balanced when both stacks contain `Integer` objects. In the second scenario, the average running time has been reduced to 82.645% with the proposed approach. This means more than 20% speed-up in the execution without any special instantiation and special ones are able to speed-up the execution significantly. However, more effective code can be generated with more specific approaches [8].

The speed-up is significant, therefore we should realize what are the main reasons behind this effect, cache consistency or other JVM runtime optimazations. As future work, we evaluate the proposed approach with more generics and explore how the instantion can be utilized much more effectively.

# 7 Future Work

As we discussed earlier, the most difficult issue that waiting for solution is the problem of context-sensitive code parts. Of course not just the ambiguous fields, but in Java Standard especially, since the container classes take advantage of the toolkit of the *Generics*. First of all, we have to explore and classify the parts which can lead to anomalies or errors if we directly transform them. Comprehensive evalutaion is necessary, as well. In the future, we want to implement an integrated development environment (IDE) plugin which wraps our solutions and provides wide support to the users. All in all, our most ambitious goal is to become part of the Java language.

# 8 Conclusion

Type parametrization is an essential construct in modern programming languages with different backgrounds. For instance, C++ provides templates that are instan-

tiated by the compiler during compilation. Java offers generics that are based on type erasure. According to the measurements, the runtime performance can be significantly better when templates are in-use.

Previously, we created Java templates and a tool that instantiates them. A new syntax was offered that was not compatible with existing code bases. Therefore, our aim became the instantiation of Java generics: standard Java generics to instantiate with a new tool. In this paper, we introduced the background of our proof-of-concept tool and we have measured and evaluated the runtime efficiency of the proposed approach. The instantiated generic performs significantly better compared to the standard solution.

## References

[1] Project Lombok. https://projectlombok.org/.

[2] Arnold, Ken, Gosling, James, and Holmes, David. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[3] Babati, Bence, Pataki, Norbert, and Porkoláb, Zoltán. C/C++ preprocessing with modern data storage devices. In *Proceedings of the 13th IEEE International Scientific Conference on Informatics*, pages 36–40. IEEE, 2015. DOI: 10.1109/Informatics.2015.7377804.

[4] Burrus, Nicolas, Duret-Lutz, Alexandre, Duret-Lutz, Re, Geraud, Thierry, Lesage, David, and Poss, Raphael. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, 2003. URL: https://www.lrde.epita.fr/wiki/Publications/burrus.03.mpool.

[5] Dragan, Laurentiu and Watt, Stephen M. Performance analysis of generics in scientific computing. In *Proceedigs of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 93–100, 2005. DOI: 10.1109/SYNASC.2005.56.

[6] Ghosh, Debasish. Generics in Java and C++: A comparative model. *ACM SIGPLAN Notes*, 39(5):40–47, 2004. DOI: 10.1145/997140.997144.

[7] Grigore, Radu. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM. DOI: 10.1145/3009837.3009871.

[8] Horváth, Gábor, Pataki, Norbert, and Balassi, Márton. Code generation in serializers and comparators of Apache Flink. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages,*

*Programs and Systems*, ICOOOLPS'17, pages 5:1–5:6, New York, NY, USA, 2017. ACM. DOI: 10.1145/3098572.3098579.

[9] Meyers, Scott. *Effective STL*. Addison-Wesley, 2001.

[10] Nam, Daye, Horvath, Amber, Macvean, Andrew, Myers, Brad, and Vasilescu, Bogdan. MARBLE: Mining for boilerplate code to identify API usability problems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 615–627, 2019. DOI: 10.1109/ASE.2019.00063.

[11] Porkoláb, Zoltán. Functional programming with C++ template metaprograms. In Horváth, Zoltán, Plasmeijer, Rinus, and Zsók, Viktória, editors, *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, pages 306–353, Berlin, Heidelberg, 2010. Springer. DOI: 10.1007/978-3-642-17685-2_9.

[12] Soha, Péter and Pataki, Norbert. Effective type parametrization in Java. *AIP Conference Proceedings*, 2116(1):350007, 2019. DOI: 10.1063/1.5114360.

[13] Stroustrup, Bjarne. *The C++ Programming Language (special edition)*. Addison-Wesley, 2000.

[14] Szűgyi, Zalán, Sinkovics, Ábel, Pataki, Norbert, and Porkoláb, Zoltán. *C++ Metastring Library and Its Applications*. In *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 461–480. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-18023-1_15.

[15] Torgersen, Mads, Hansen, Christian Plesner, Ernst, Erik, von der Ahé, Peter, Bracha, Gilad, and Gafter, Neal. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1289–1296, New York, NY, USA, 2004. ACM. DOI: 10.1145/967900.968162.

Acta Cybernetica **25** (2022) 909–921.

# Improved Loop Execution Modeling in the Clang Static Analyzer*

Péter Szécsi[ab], Gábor Horváth[ac], and Zoltán Porkoláb[ad]

**Abstract**

The LLVM Clang Static Analyzer is a source code analysis tool which aims to find bugs in C, C++, and Objective-C programs using symbolic execution, i.e. it simulates the possible execution paths of the code. Currently the simulation of the loops is somewhat naive (but efficient), unrolling the loops a predefined constant number of times. However, this approach can result in a loss of coverage in various cases.

This study aims to introduce two alternative approaches which can extend the current method and can be applied simultaneously: (1) determining loops worth to fully unroll with applied heuristics, and (2) using a widening mechanism to simulate an arbitrary number of iteration steps. These methods were evaluated on numerous open source projects, and proved to increase coverage in most of the cases. This work also laid the infrastructure for future loop modeling improvements.

**Keywords:** static analysis, symbolic execution, loop modeling

## 1 Introduction

During software development it is natural to make mistakes. Consequently, writing various test cases is required in order to validate the behavior of the program. In addition to the costs of test writing, it is possible that the developers fail to cover all possible critical cases. Furthermore, test writing and running often happens later than code development, but the costs of error correction increases proportionally to elapsed time [2]. This proves that testing alone is not necessarily sufficient to ensure code quality.

The static analysis tools offer a different approach for code validation [6, 1]. Moreover, they can potentially check for some characteristics of the code – which

*This study was supported by the ÚNKP-17-2 New National Excellence Program of the Hungarian Ministry of Human Capacities and by the EFOP-3.6.2-16-2017-00013

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: ps95@caesar.elte.hu, ORCID: 0000-0001-9156-1337

[c]E-mail: xazax@caesar.elte.hu, ORCID: 0000-0002-0834-0996

[d]E-mail: gsd@caesar.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.283176

cannot be verified by testing – e.g. the adherence to conventions. Unfortunately, it is impossible to detect every bug using static analysis [8] without a large number of spurious warnings. Static analyzer tools might not be able to discover some bugs (these are called false negatives) or report correct code snippets as incorrect (false positives). As the developer's time is one of the most valuable resource in the industry and reports of the automated tools are evaluated manually, industrial tools aim to keep the ratio of the false positive reports low while still be able to find real bugs.

The purpose of the Clang Static Analyzer is to find bugs by performing a symbolic execution [4, 3] on the code. During symbolic execution, the program is being interpreted, on a function-by-function basis, without any knowledge about the runtime environment. It builds up and traverses an inner model of the execution paths, called *ExplodedGraph*, for each analyzed function.

The Static Analyzer – as it is indicated by its name – build around the Clang compiler [5]. An important technical note is that the building of the `ExplodedGraph` is based on the *Control Flow Graph* (`CFG`) of the functions. The `CFG` represents a source-level, intra-procedural control flow of a statement. This statement can potentially be an entire function body, or just a single expression. The `CFG` consists of `CFGBlocks` which are simply containers of statements. The `CFGBlocks`s essentially represent the basic blocks of the code but can contain some extra custom information. Although basic blocks and `CFGBlocks` are technically different, in the rest of the article the term basic blocks will be used for `CFGBlocks` as well for the sake of easier understanding and better illustration.

Thus during the analysis – based on the function CFGs – an `ExplodedGraph` is built up. A node of this graph (called `ExplodedNode`) contains a `ProgramPoint` (which determines the location) and a `State` (which contains any known information at that point). Its paths from the root to the leaves are modeling the different execution paths of the analyzed function. Whenever the execution encounters a branch, a corresponding branch will be created in the `ExplodedGraph` during the simulated interpretation. Hence, branches lead to an exponential number of `ExplodedNode`s. This combinatorial explosion is handled in the Static Analyzer by stopping the analysis when given conditions are fulfilled. Terminating the analysis process may cause loss of potential true positive results, but it is indispensable for maintaining a reasonable resource consumption regarding the memory and CPU usage.

These conditions are modeled by the concept of budget. The budget is a collection of constraints on the shape of the `ExplodedGraph` including:

1. The maximum number of traversed nodes in the `ExplodedGraph`. If this number is reached then the analysis of the simulated function stops.

2. The size of the simulated call stack. When a function call is reached then the analysis continues in its body as if it was inlined to the place of call (interprocedural). There are several heuristics that may control the behavior of inlining process. For example the too large functions are not inlined at all, and the really short functions are not counted in the size of call stack.

3. The number of times a function is inlined. The idea behind this constraint is that the more a function is analyzed, the less likely it is that a bug will appear in it. If this number is reached then that function will not be inlined again in this `ExplodedGraph`.

4. The number of times a basic block is processed during the analysis. This constraint limits the number of loop iterations. When this threshold is reached the currently analyzed execution path is aborted. The budget expression can be used in two ways. Sometimes it means the collection of the limitations above, sometimes it refers to one of these limitations. This will always be distinguishable from the context.

## 2 Motivation

Currently, the analyzer handles loops quite simply. It unrolls them 4 times by default and then cuts the analysis of the path where the loop would have been unrolled more than 4 times. This behavior is enforced by the above presented basic block visiting budget.

One of the problems with this approach to loop modeling is the loss of coverage. Specifically, in cases where the loop is statically known to make more than 4 steps, the analyzer do not analyze the code following the loop. Thus, the naive loop handling (described above) could lead to entirely unchecked code. Listing 1 shows a small example exercising this behavior.

```
1 void foo() {
2   int arr[6];
3   for (int i = 0; i < 6; i++) {
4     arr[i] = i;
5   }
6   /*rest of the function*/
7 }
```

Listing 1: Since the loop condition is known at every iteration, the analyzer will not check the 'rest of the function' part in the current state.

According to the budget rule concerning the basic block visit count, the analysis of the loop stops in the fourth iteration even if the loop condition is simple enough to see that unrolling the whole loop would not be too much extra work relatively. Running out of the budget implies (in this case) that the rest of the function body remains unanalyzed, which may lead to not finding potential bugs. Another problem can be seen on Listing 2:

```
int num();
void foo() {
  int n = 0;
  for (int i = 0; i < num(); ++i) {
    ++n;
  }
  /*rest of the function, n < 4 */
}
```

Listing 2: The loop condition is unknown but the analyzer will not generate a simulation path where n $\geq$ 4 (which can result coverage loss).

This code fragment results in an analysis which keeps track of the values of `n` and `i` variables (this information is stored in the State). In every iteration of the loop the values are updated accordingly. Note that updating the State means that a new node is inserted into the `ExplodedGraph` with the new values. Since the body of the `num()` function is unknown, the analyzer can not find out its return value. Thus it is considered as unknown. This circumstance makes the graph to split into two branches. The first one belongs to the symbolic execution of the loop body assuming that the loop condition is true. The other one simulates the case where the condition is false and the execution continues after the loop. This process is done for every loop iteration, however, at the 4th time, assuming the condition is true, the path will be cut according to the budget rule. Even though the analyzer generates paths to simulate the code after the loop in the above described case, the value of variable `n` will always be less than 4 on these paths and the rest of the function will only be checked with this assumption. This can result in coverage loss as well, since the analyzer will ignore the paths where n is more than 4.

# 3 Proposed Solution

In this section two solutions are presented to resolve the above mentioned limitations on symbolic execution of loops in the Clang Static Analyzer. It is important to note that these enhancements are incremental in the sense that the analyzer falls back to the original method on examples which are too complex to handle at the moment. For the sake of simplicity we will use a "division by zero" bug to illustrate the analyzer's behavior in the following examples.

## 3.1 Loop Unrolling Heuristics

We have identified heuristics and patterns (such as loops with small number of branches and small known static bound) in order to find specific loops which are worth to be completely unrolled. This idea is inspired by the following example:

```
void foo() {
  for (int i = 0; i < 6; i++) {
    /*simple loop which does not
```

```
4     change 'i' or split the state*/
5   }
6   int k = 0;
7   int l = 2/k; // Division by zero
8 }
```

Listing 3: Complete unrolling of the loop makes it possible to find the division by zero error.

Currently, a loop has to fulfill the following conditions in order to be unrolled:

1. The loop condition should arithmetically compare a variable – which is known at the beginning of the loop – to a literal (like: `i < 6` or `6 ≥ i`)
2. The loop should modify the loop variable only once per iteration in its body and the difference needs to be constant. (This way the maximum number of steps can be estimated.)
3. There is no alias created to the loop variable.
4. The estimated number of steps should be less than 128. (Simulating loops which takes thousands of steps because they could single handedly exhaust the budget.)
5. The loop must not generate new branches or use `goto` statements.

By using this method, the bug on the Listing 3 example is found successfully.

## 3.2 Loop Widening

The final aim of widening is quite the same as the unrolling, to increase the coverage of the analysis. However, it achieves its goal in a very different way. During widening the analyzer simulates the execution of an arbitrary number of iterations. The analyzer already had a widening algorithm which reaches this behavior by discarding all of the known information before the last step of the loop. So the analyzer creates the paths for the first 3 steps and simulate them as usual, but in order to avoid losing the first precise simulation branches, the widening (i.e. the invalidating) happens before the 4th step. This way the coverage will be increased, however, this method is disabled by default, since it can easily result in too much false positives. Consider the example on Listing 4.

```
1 int num();
2 void foo() {
3   bool b = true;
4   for (int i = 0; i < num(); ++i) {
5     /*does not change 'b'*/
6   }
7   int n = 0;
```

```
8   if (b)
9     n++;
10  n = 1/n; // False positive:
11           // Division by zero
12 }
```

Listing 4: Invalidating every known information (even those which are not modified by the loop) can easily result in false positives.

In this case the analyzer will check that unfeasible path where the variable `b` is false, so `n` is not incremented and lead into a division by zero error. Since this execution path would never be performed while running the analyzed program, it is considered a false positive. Our aim was to give a more precise approach for widening. There was already conversation within the community about some possible enhancements [7].

One of the main principles is that the analysis should still continue after the block visiting budget is exhausted and the information of only those variables should be invalidated which are possibly modified by the loop, e.g. a statement, like `arr[i] = i` where `i` is the loop variable, means that we discard the data on the whole `arr` array but nothing else. For this reason we developed a solution which checks every possible way in which a variable can be modified in the loop. Then these cases are evaluated and if it encounters a modified variable which cannot be handled by the invalidation process (e.g.: a pointer variable), then the loop will not be widened and we return to the conservative method. This mechanism ensures that we do not create nodes that contain invalid states. This approach helps us to cover cases and find bugs like the one illustrated on Listing 5 without reporting false positives presented on Listing 4.

```
1  int num();
2  void foo() {
3    int n = 0;
4    for (int i = 0; i < num(); ++i) {
5      ++n;
6    }
7    if (n > 4) {
8      int k = 0;
9      k = 1/k;  // Division by zero error
10   }
11 }
```

Listing 5: Invalidating the information on only the possible changed variables can result higher coverage (while limiting the number of the found false positives).

The bug is found by invalidating the known information on variable `n` (and `i` as well). This makes the analyzer to create a branch where it checks the body of the `if` statement and finds the bug. However, this solution has its own limitations when dealing with nested loops. Consider the case on Listing 6.

```
int num();
void foo() {
  int n = 0;
  for (int i = 0; i < num(); ++i) {
    ++n;
    for (int j = 0; j < 4; ++j) {
      /*body that does not change n*/
    }
  }
  /*rest of the function, n <= 1 */
}
```

Listing 6: The naive widening method does not handle well the nested loops. In this example the outer loop will not be widened.

In this scenario, when the analyzer first step into the outer loop (so it assumes that `i < num()` is true) and encounter the inner loop, it consumes its (own) block visiting budget. (This implies that it will be widened, although in this case it means that only the inner loop counter (`j`) information is discarded.) After moving on to the next iteration, we may assume that we are on the path where the outer loop condition is true again. Due to the fact that the budget was already exhausted in the previous iteration, the next visit of the first basic block of the inner loop (the condition) means that this path will be completely cut off and not analyzed. This results in the outer loop not reaching the step number where it would been widened. Furthermore, the outer loop will not even reach the 3rd step, even the 2nd is stopped at in its body (as described above). This causes the problem that even though the loop widening method is used, the rest of the function will be analyzed by the assumption `n <= 1`.

In order to deal with the above described nested loop problem, we have implemented a *replay mechanism*. This means that whenever we encounter an inner loop which already consumed its budget, we replay the analysis process of the current step of the outer loop after performing a widening first. This ensures the creation of a path which assumes that the condition is false and simulates the execution after the loop while the possibly changed information are discarded. This way the analyzer will not exclude some feasible path because of the simple loop handling which solves the problem.

An additional note to the widening process is that it makes sense to analyze the branch where the condition is true with the widened State as well. The example on Listing 7 shows a case where this is useful.

```
int num();
void foo() {
  int n = 0;
  int i;
  for (i = 0; i < num(); ++i) {
    if (i == 7) {
      break;
```

```
8       }
9       for (int j = 0; j < 4; ++j) {/* */}
10    }
11    int n = 1 / (7 - i);
12            // ^ Possible division by zero
13  }
```

Listing 7: The replay mechanism successfully helps us to find the possible error the outer loop.

This way the analyzer will produce a path where the value of `i` is known to be 7, so it will be able find the possible division by zero error.

# 4 Evaluation

The effect of the described loop modeling approaches was measured on various popular C/C++ open source projects. These are the following:

| Project | LoC | Language |
|---|---|---|
| TinyXML | 20k | C++ |
| Curl | 21k | C |
| Redis | 40k | C |
| Xerces | 228k | C++ |
| Vim | 540k | C |
| OpenSSL | 550k | C |
| PostgreSQL | 950k | C |
| FFmpeg | 1080k | C |

## 4.1 Coverage and the number of explored paths

Keeping track of these statisics are already part of the analyzer. The coverage percentage is based on the ratio of the visited and the total number of basic blocks in the analyzed functions (instead of the number of visited statements), which results in a small imprecision. It is important to note that the introduced loop modeling methods require having additional loop entrance and exit point information in the `CFG`. This can lead to having more basic blocks in the `CFG` and it can affect the statistics. As a result, even statistics produced by using the current loop modeling approach were measured with this information added to the `CFG`.

The coverage and the number of explored paths are generated for every translation unit and then summarized. This means that header files which are included in more than one translation unit can influence more statistics. However, by using this summarization process consistently for every measurement the results reflect the reality.

The tables presented in this section summarize measurement results using different loop modeling approaches: the current practice (denoted by Normal) and the hereby introduced loop unrolling (Unroll) and loop widening (Widen) methods separately and simultaneously (U+W).

Table 1 shows the coverage difference using the introduced approaches. On most of the projects, analysis coverage was strictly increased by using any of the proposed approaches. The widening method had a stronger influence on the coverage in the average case. However, the complete unroll of specific loops could result in a higher coverage as well (e.g. Curl, Redis). In general, enabling both of them was the most beneficial with respect to the coverage.

Table 2 presents the numbers of analyzed execution paths. As expected, both introduced loop modeling methods resulted in a higher number of simulated paths on (almost) all of the projects. The only exception is the unrolling approach on the FFmpeg project, which caused the budget limiting the number of traversed

Table 1: The code coverage of the analysis on the evaluated projects expressed in percentage

| Project | Normal | Unroll | Widen | U + W |
|---|---|---|---|---|
| TinyXML | 84.2 | 84.2 | 85.1 | 85.1 |
| Curl | 76.2 | 76.9 | 77.7 | 77.2 |
| Redis | 68.5 | 69.1 | 68.5 | 71.3 |
| Xerces | 92.3 | 92.4 | 92.7 | 92.7 |
| Vim | 60.4 | 60.6 | 60.6 | 60.7 |
| OpenSSL | 97.4 | 97.5 | 97.7 | 97.7 |
| PostgreSQL | 76.9 | 77.0 | 76.9 | 76.9 |
| FFmpeg | 86.1 | 86.3 | 87.0 | 86.8 |

Table 2: The numbers of explored execution paths using different loop modeling approaches

| Project | Normal | Unrolling | Widening | U + W |
|---|---|---|---|---|
| TinyXML | 14 452 | 15 460 | 14 765 | 15 773 |
| Curl | 18 272 | 18 577 | 28 835 | 24 279 |
| Redis | 69 857 | 70 097 | 98 446 | 100 929 |
| Xerces | 395 615 | 398 077 | 430 989 | 433 358 |
| Vim | 155 451 | 157 266 | 188 136 | 173 121 |
| OpenSSL | 687 175 | 687 932 | 700 464 | 701 013 |
| PostgreSQL | 382 660 | 383 874 | 453 188 | 419 118 |
| FFmpeg | 466 613 | 458 480 | 571 399 | 521 725 |

`ExplodedNode`s to exhaust earlier, slightly decreasing the number of checked paths. Enabling both of the features resulted in similar or fewer number of explored paths than the runs using only widening. This effect can be explained in two ways: (1) the analyzer prefers to completely unroll loops rather than widen them, which results in a more precise modeling of the state and can exclude unfeasible paths, (2) the simultaneous use of the methods can lead to exhausting the budget on earlier paths, where the analysis will be terminated.

## 4.2 Found bugs

The number of bug reports using the different loop modeling methods can be seen in Table 3. The increase in analysis coverage and in the number of checked paths usually implies an increased number of found bugs, which indeed can be observed on the numbers. However, it is important to note that the upsurge of the number of explored execution paths described in Table 2 considerably outweighs the moderate rise in the number of bug reports. In some cases enabling a proposed feature could result in less results due to two important factors: (1) the more information we collect by precisely analyzing the execution paths does not result in false conclusion, (2) the global budget is exceeded for exploring new paths and some of the earlier checked will be skipped. Unfortunately, case (2) is a possible scenario, however, the increased coverage using the described features shows that this way we still explore more interesting cases. Since the loop widening method creates more new paths by discarding information on the values of variables, it could introduce the risk of analyzing paths that lead to false positives. However, from the results it seems that this was not a problem in practice: relative to the increase in the number of analyzed paths, the number of reports hardly increased. Moreover, based on studying the environment of the found bugs, the ratio of false positive findings was low (beside some clear true positive) among the newly detected bugs.

Table 3: The number of bug reports produced by the analyzer.

| Project | Normal | Unrolling | Widening | U + W |
|---|---|---|---|---|
| TinyXML | 1 | 1 | 3 | 3 (+200%) |
| Curl | 16 | 16 | 16 | 16 (0%) |
| Redis | 55 | 58 | 55 | 59 (+7.27%) |
| Xerces | 62 | 62 | 61 | 61 (-1.61%) |
| Vim | 74 | 74 | 76 | 78 (+5.4%) |
| OpenSSL | 152 | 152 | 153 | 153 (+0.66%) |
| PostgreSQL | 323 | 323 | 327 | 331 (+2.48%) |
| FFmpeg | 425 | 420 | 423 | 454 (+6.82%) |

### 4.3 Analysis time

The running time on different projects is showed in Table 4. Although the widening method lead into more analyzed execution paths, the analysis time increase was more intense after enabling the unrolling process. This is possible due to the fact that unrolling leads to long paths where the `State` usually contains more information (constraints on variable values), which is very expensive in respect of running time. In general there was a manageable increase in the analysis time at all examined projects which suggests a good scalability of the proposed improvements.

Table 4: Average measured time of the analysis expressed in minutes. (Average of 5 runs.)

| Project | Normal | Unrolling | Widening | U + W |
|---|---|---|---|---|
| TinyXML | 0:51 | 0:51 | 0:52 | 0:52 (+2%) |
| Curl | 0:50 | 1:06 | 0:55 | 1:05 (+30%) |
| Redis | 2:06 | 2:11 | 2:28 | 2:10 (+3%) |
| Xerces | 3:38 | 3:34 | 3:37 | 3:39 (+0.5%) |
| Vim | 3:11 | 3:26 | 3:18 | 3:27 (+3%) |
| OpenSSL | 2:04 | 2:22 | 2:13 | 2:19 (+8.3%) |
| PostgreSQL | 7:03 | 8:32 | 7:48 | 7:59 (+13%) |
| FFmpeg | 9:40 | 10:22 | 10:14 | 11:20 (+17%) |

## 5 Future work

The heuristic patterns for completely unrolled loops could be extended to involve loops whose bound is a known variable which is not changed in the body. Furthermore, even more general rules would be beneficial: consider loops where the value variables are known at the beginning and they are affected by a known constant change by every iteration. These improvements have not been implemented yet due to some technical and framework limitations.

During the widening process we invalidate any possibly changed information. However, a change made on a pointer could mean that we need to invalidate all variables due to the lack of advanced pointer analysis. Therefore, introducing pointer analysis algorithms to the analyzer could help to develop a more precise invalidation process.

The infrastructural improvements enable the analyzer to provide entry points for bug finding modules (checkers) on loop entrances/exits and identify the currently simulated loop for every `ExplodedNode`. On top of these entry points new checkers can be implemented.

# 6 Conclusion

Two alternative approaches was introduced for improving the simulation of loops during symbolic execution. These were implemented and subsequently evaluated on various open source projects, with a clear improvement of code coverage in general. The new methods make it possible to explore previously skipped, feasible execution paths, especially when both of them are used in conjunction.

The required changes done to the underlying infrastructure should also ease the implementation of future enhancements. In particular, information tracked by the analysis about location contexts were expanded with additional fields. While code coverage was measured to have increased by an average of 0.8% and the number of explored execution paths by an average of 16%, there was a noticeable performance penalty as well. A general increase in the execution time was observed, with an average of 9.5%. The number of simulated paths also increased proportionally with the time taken, suggesting this time was well spent. In conclusion, if the user does not mind taking ∼10% more time for a more comprehensive analysis, then it is beneficial to enable the proposed feature set by default.

# 7 Acknowledgment

We would like to thank to the members of the `CodeChecker` team at Ericsson for their valuable and helpful suggestions on the paper.

# References

[1] Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, and Engler, Dawson. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010. DOI: 10.1145/1646353.1646374.

[2] Boehm, Barry and Basili, Victor R. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001. DOI: 10.1109/2.962984.

[3] Hampapuram, Hari, Yang, Yue, and Das, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, September 2005. DOI: 10.1145/1108768.1108808.

[4] King, James C. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, 1975.

[5] Lattner, Chris. LLVM and Clang: Next generation compiler technology. URL: https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.html, 2008. Lecture at BSD Conference.

[6] Michael, Zhivich and Robert, K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009. DOI: 10.1109/MSP.2009.56.

[7] Phabricator. Community conversion about loop widening. URL: https://reviews.llvm.org/D12358, 2015.

[8] Rice, Henry G. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

Acta Cybernetica **25** (2022) 923–940.

# Detecting Uninitialized Variables in C++ with the Clang Static Analyzer*

Kristóf Umann[ab] and Zoltán Porkoláb[ac]

**Abstract**

Uninitialized variables have been a source of errors since the beginning of software engineering. Some programming languages (e.g. Java and Python) will automatically zero-initialize such variables, but others, like C and C++, leave their state undefined. While laying aside initialization in C and C++ might be a performance advantage if an initial value cannot be supplied, working with variables is an undefined behaviour, and is a common source of instabilities and crashes. To avoid such errors, whenever meaningful initialization is possible, it should be applied. Tools for detecting these errors run time have existed for decades, but those require the problematic code to be executed. Since in many cases, the number of possible execution paths is combinatoric, static analysis techniques emerged as an alternative to achieve greater code coverage. In this paper, we overview the technique for detecting uninitialized C++ variables using the Clang Static Analyzer, and describe various heuristics to guess whether a specific variable was left in an undefined state intentionally. We implemented and published a prototype tool based on our idea and successfully tested it on large open-source projects. This so-called "checker" has been a part of LLVM/Clang releases since 9.0.0 under the name optin.cplusplus.UninitializedObject.

**Keywords:** C++, static analysis, uninitialized variables

## 1 Introduction

When declaring a variable in program code, we might not be able to come up with a meaningful default value thus leaving the variable uninitialized. This is not an issue if one later assigns said variable before reading it, but such errors can be introduced through, for example, code maintenance. Different languages approach this problem in different ways: Java, Python (and many others) zero- initialize variables by

*This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

[a]Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: szelethus@inf.elte.hu, ORCID: 0000-0002-6679-5614

[c]E-mail: gsd@inf.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.282900

default, while others, like C and C++, leave their values in an undefined state, and working with such variables leads to undefined behaviour at runtime.

Undefined behaviour in C/C++ is any behaviour the standard does not specify. It may occur, among many other sources, when a null pointer is dereferenced, the division is made by zero, or an array is indexed out of its bounds. The real danger of undefined behaviour is that in some cases, the program might behave seemingly correctly, but other times run into runtime errors, e.g. crashing, corrupting opened files or memory regions allocated by other programs. Also, even the same kind of undefined behaviours might manifest in different runtime errors on different executions.

This makes catching undefined behaviour often very hard – in the case of uninitialized variables, zero initialization might occur with a particular compiler, on a particular platform, in a particular build mode, but might also leave the variable hold whatever value that was stored in the memory region to which the variable was assigned (so-called "garbage value"). However, the use of this behaviour could result in some performance enhancement, tempting programmers to not initialize variables, even though this is often a bad approach.

This paper investigates how such variables can be detected using static analysis. Unlike many other available tools, we will focus on non-idiomatic C++ initialization rather then uninitialized value misuse. In Section 2, we will discuss related initialization rules in C++. In Section 3, we overview runtime and static tools that are available for the same problem and sample several techniques of the latter in Section 4 with a highlight on symbolic execution. We will detail the actual implementation of our prototype in Section 5 along with the heuristics we use to emit only reports that will most likely result in incorrect program behaviour, and what other heuristics could be implemented. We evaluate our prototype solution on various large open-source codebases in Section 6 and discuss future works in Section 7. Our paper concludes in Section 8.

## 2 C++ Initialization Rules

The way variables are initialized in C/C++ depends on the variables' type. They can be categorized by whether they are records, arrays, or else. For this paper, we will refer to the latter category as *primitive*. The C++ standard specifies two types of initialization that may result in an indeterministic value [13, p. 221][25], but zero initialization is also relevant in this context. When creating an uninitialized object of type `T`

- default initialization occurs
  - if `T` is a class: default constructor is called, or
  - if `T` is an array, each element is default initialized, or
  - otherwise, no initialization occurs resulting in an indeterminate value,

| Variable declaration | `i`'s initialization |
|---|---|
| `T i;` | default initialization |
| `T i{};` | value initialization (C++11) |
| `T i = T();` | value initialization |
| `T i = T{};` | value initialization (C++11) |
| `T i();` | function declaration (no variable initialization occurs) |

Figure 1: Initialization rules for instatiation of local variables

| `A::A()`'s definition | `A::t`'s initialization |
|---|---|
| `struct A { T t; A() : t() {} };` | value initialization |
| `struct A { T t; A() : t{} {} };` | value initialization (C++11) |
| `struct A { T t; A() {}; };` | default initialization |
| `struct A { T t; A() = default; };` | value initialization |

Figure 2: Initialization rules for data member `A::t` upon instantiating a local `A` typed variable with the default constructor.

- value initialization occurs
  - if T is a class, the object is default initialized after zero initialization if T's default constructor is not user-defined/deleted, or
  - if `T` is an array, each element is value initialized, or
  - otherwise the object is zero initialized,
- zero-initialization occurs, before any other initialization
  - if the object is static or thread-local, or
  - if `T` scalar (number, pointer, enum) set to 0, or
  - if `T` is a class, all subobjects are zero-initialized.

We summarize these rules in Figure 1. and 2. While we did not discuss initialization in great depth in this section, it shows that this issue is not only context-sensitive, it is also confusing, since it can be intertwined with other C++ rules, such as those related to inheritance. Even telling when the compiler will generate a constructor can be difficult [30].

# 3 Related work

When approaching the issue of uninitialized variables, we can sort the already existing solutions into two categories: runtime and static. An ideal solution is both *correct* and *complete*, where none of our reports is incorrect (*false positive*), and we

identify all uninitialized objects. Different techniques tend to emphasize different parts of the requirements mentioned above – Runtime techniques are inherently less prone to report false positives, but lack completeness [14]. Static analysis tools cover far more of the code and offer a more complete but less precise solution [18].

In the context of this paper, we are looking for a solution to detect error-prone lack of initialization and not uninitialized value misuses. Despite this, both due to the scarcity of tools that focus on the former rather than the latter, and the strong relevance of the two problems, we feel it is important to take a short survey of analyzers and techniques that implement rules for either.

### 3.1 Runtime analysis tools

Several papers survey the detection of uninitialized value misuses in runtime analyzers [14, 20, 6]. A common characteristic of said tools is to inspect the program during execution, with a given input. This means that in any given analysis, these tools will only observe a single path of execution. However, this simplification makes these tools far more precise, improving, on that particular path of execution, both the true positive and the true negative findings.

Valgrind is a general dynamic binary instrumentation (DBI) framework [23]. It inspects the executable binary, rather than the source code of a program, which might not even be available. Valgrind offers several tools (also referred to as *plugins*) that can find bugs. It is implemented by its core disassembling a given code block from the binary into an intermediate representation, which is instrumented with analysis code by the plugin, and then converted back into machine code. The resulting *translation* is stored in a code cache to be rerun as necessary. One of these plugins, MemCheck [27], can find uninitialized value misuses using shadow bits [22] for every byte in the application memory: one a bit indicates whether it is addressable, and a bit indicates whether it has a defined value.

Dr. Memory [4] works similarly to MemCheck but is more modern, better optimized and multi-threaded. Its two times faster then MemCheck, but due to concurrent updates of adjacent shadow bits, is more prone to emit false positive and false negative reports [28].

MemorySanitizer [28] offers a different approach to runtime analysis by only solving the problem of uninitialized variable misuses. It generates a modified binary during compilation, skipping disassembly and reassembly entirely. Running the generated binary is far faster and consumes less memory then the solution Valgrind with MemCheck or Dr. Memory offers, but this requires the source code to be available.

### 3.2 Static analysis tools

Static analyzers do not execute the program under analysis, but rather inspect either the source code or the generated binary code. This results in far greater code coverage as they are not restricted to a single path of execution. However, this generality comes at the cost of the analyzer tool having little knowledge about input

values. In parts of the code where such information is crucial, the given analyzer might have to conservatively suppress its reports, or simulate the execution of parts of the code, making assumptions on values. This, considering the complexity of some of the C++ language features (as discussed in Section 2.) can result in a higher number of false positives and false negatives. We will discuss some of the more popular techniques in Section 4. For the remainder of this section, we will sample some of the widely used C/C++ static analyzers.

CppCheck [21] is among the earliest open-source tools with support for C++. It uses AST matching and dataflow analysis[1] to find bugs and code smells. Contrary to many open-source tools for C++ analysis, CppCheck implements its own preprocessor, parser and abstract syntax tree (*AST*). It defines two rules on incorrect initialization in constructors, separately for private and non-private fields.

Infer [5] is a relatively new tool, focusing on scalability and fast execution. It has a unique approach to static analysis, using bi-abduction to perform interprocedural analysis. Infer also runs with cross translation unit analysis enabled by default, and scales significantly better with the number of translation units to analyze compared to other tools such as the Clang Static Analyzer [10]. While it has several checkers to detect uninitialized value misuse, it does not have any that focuses on non-idiomatic C++ object initialization.

The Clang Static Analyzer [18], similarly to CppCheck, is among the more mature static analyzers for C++. Having the benefit of being implemented directly in the Clang compiler and transitively LLVM itself, it can take advantage of several well-tested algorithms and data structures. The Clang Static Analyzer (or *analyzer* for short) was ultimately our choice of project to implement our prototype in and will be discussed in greater detail in Section 5.1.

There are also several commercial static analyzers such as CodeSodar [8], Coverity [29], Klocwork [15], but due to licencing issues we will not compare our results to them.

# 4 An introduction to symbolic execution

Several static analysis techniques may be considered for finding uninitialized variables, each having different strengths and weaknesses in terms of analysis speed, memory or persistent storage consumption. In this section, we introduce symbolic execution through two other approaches, and demonstrate why it is more appropriate for our purpose.

## 4.1 Text-based matching

A possible, though a primitive approach would be to use textual pattern matchers. Let us see through a couple of examples whether we can tackle the problem initialization with regular expressions:

[1] The authors of CppCheck refer to this technique as "valueflow", rather then dataflow.

```
int i;
```

With the regular expression rule `int [A-Za-z]+[A-Za-z0-9]*;` we can catch this error. We can even enhance this regular expression by handling other fundamental types, ignoring whitespaces, C-style comments and the like. By inspecting the preprocessed code, rather than the original source code, we can also handle cases where the preprocessor would generate parts of the expression. However, in Figure 3, we demonstrate that non-trivial cases require a context-sensitive grammar. On that code snippet, `a.i` will be initialized by the of the constructor call, but `a.j` will not be. Having multiple constructors, potentially *after* instantiating the class, inheritance, virtual inheritance, constructor delegation, aggregate initialization make solving even smaller parts of this problem practically impossible with text-based pattern matching.

```
1  struct A {
2    int i;
3    int j;
4    A() : i(0) {}
5  };
6
7  A a;
```

Figure 3: Text-based pattern matching is unable to identify `a.j` as uninitialized.

## 4.2 AST matching

A more sophisticated approach is to utilize the *abstract syntax tree* (AST), which provides far more C++ specific information, especially when coupled with semantic information, such as type information and an identifier table.

For the code snippet on Figure 4a, according to the rules detailed in Section 2, `b1`, `b2` are value initialized while `b3` is default initialized, leaving `b3.data` in an indeterministic state. As discussed earlier, we cannot tell this with regular expressions.

On Figure 4c, we can inspect how Clang constructs the AST for Figure 4a. Using Clang's AST matcher library, we can match the line on which `b3` is defined with the following matcher:

```
declStmt(unless(hasDescendant(
    stmt(anyOf(cxxConstructExpr(requiresZeroInitialization()),
               implicitValueInitExpr())))));
```

This approach is clearly superior to text-based matching because it is context-sensitive and allows us to express C++ specific properties easily. AST matching can be complemented with the retrieval of the matched expression, enabling us to do additional compile-time analysis, such as inspecting the inheritance tree of a

```
1  struct RealSqrt {
2    int data;
3
4
5
6
7  };
8
9  int main() {
10   RealSqrt b1 = RealSqrt();
11   RealSqrt b2{};
12   RealSqrt b3;
13 }
```

(a)

```
1  struct RealSqrt {
2    int data;
3    RealSqrt(int i) {
4      if (i >= 0)
5        data = i;
6    }
7  };
8
9  int main() {
10   int i = rand();
11   if (i >= 0)
12     RealSqrt b(i);
13 }
```

(b)

```
CXXRecordDecl line:1:8 referenced struct A definition
|-DefinitionData
| |-DefaultConstructor trivial
| |-CopyConstructor trivial
| |-MoveConstructor trivial
| |-CopyAssignment trivial
| |-MoveAssignment trivial
| `-Destructor irrelevant trivial
|-FieldDecl col:7 referenced a 'int'
FunctionDecl line:5:5 main 'int ()'
`-CompoundStmt
  |-DeclStmt
  | `-VarDecl col:5 b1 'A' cinit
  |   `-ExprWithCleanups 'A'
  |     `-CXXConstructExpr 'A' 'void (A &&) noexcept'
  |       `-MaterializeTemporaryExpr 'A' xvalue
  |         `-CXXTemporaryObjectExpr 'A' 'void () noexcept' zeroing
  |-DeclStmt
  | `-VarDecl col:5 b2 'A' listinit
  |   `-InitListExpr 'A'
  |     `-ImplicitValueInitExpr 'int'
  |-DeclStmt
  | `-VarDecl col:5 b3 'A' callinit
  |   `-CXXConstructExpr 'A' 'void () noexcept'
```

(c) Simplified AST generated by Clang for Figure Figure 4a.

Figure 4: Code snippets demonstrating the internal representation and analysis techniques used by Clang.

type, gathering all direct (inherited and in-class) fields and checking whether the called constructor is compiler-generated.

While the techniques mentioned above can be used to reduce the number of false positives drastically, not even AST based matching can effectively deal problems requiring path sensitive information.

It is clear that on Figure 4b, `b.data` will not be uninitialized, but we cannot reliably detect that with AST matchers. Even if we do additional compile-time analysis, we cannot reason about runtime values, branches, loops and the like without path sensitive analysis. However, AST-based pattern matching is relatively fast and can be used as a supplement to a better approach.

### 4.3 Symbolic Execution

Symbolic execution [9] is a powerful static analysis technique: it essentially simulates the execution of the program. The implementing tool follows the control flow graph (*CFG*), and evaluates statements within a basic block. Each expression is represented with a symbolic expression that is assigned a value (e.g. assigning `i + 1` with the value `11`), which itself could be symbolic if the value is unknown (e.g. supplied from a file or the command line, randomly generated or received from a translation unit we cannot analyze).

Upon encountering branch statements (when a basic block has more than one outgoing edge), the tool will use a constraint solver to evaluate the condition [16]. If the condition can be proven to be false or true in the given program state, the tool can ignore all but one of the outgoing edges. Otherwise, the analyzer will explore both paths, one on which the condition of it is true, and one where it is false. This introduces the possibility to impose constraints upon symbolic values, such as a pointer check could tell that the pointer value is non-null on a path, even if the precise value is still unknown, and null on the other.

With this technique, we can theoretically explore all execution paths and reason about the values of variables. For Figure 4b, we will note that `A::A()` will only be called if the supplied parameter has a value of 5, and there is no uninitialized value problem made in the program.

Since we can inspect the values in any given program state, handling complex C++ code, such as inheritances, virtual inheritances, constructor delegations come naturally: we could inspect an object after the end of a constructor call.

Clearly, symbolic execution is a far more powerful tool than text- or AST- based pattern matching, hence our decision to use it in our implementation. It is worth noting though that such an analysis is several times slower than compilation.

## 5 Implementation

In this section, we will detail how we implemented our prototype in the Clang Static Analyzer, and some of the heuristics used to guess whether the discovered uninitialized objects were left as such intentionally.

## 5.1 Clang Static Analyzer

Clang is the C/C++ frontend of the LLVM optimizer and code generator and houses the Clang Static Analyzer (*analyzer*). The Clang Static Analyzer implements all three analyses detailed in Section 4. In this project, symbolic execution is divided into two main components: a core that explores paths of execution and evaluates statements, and so-called *checkers* that define bugs or code smells. Our prototype is also a *checker*.

During symbolic execution, the core will simultaneously build an *exploded graph* [12]. This graph describes the entire analysis – nodes could represent new information the analyzer learned while evaluating a statement, such as a new constraint on a symbolic expression, the end of a code block, call to a function, end of a function call, or even checker-specific information (e.g. our prototype creates a new node to note already reported fields). Control statements like `if` and `while` could correlate to several exploded nodes, depending on how many times the condition was evaluated. We call the event when the analyzer constructs multiple child nodes *state splitting*, and it may be introduced by checkers as well. For instance, a dynamic memory modelling checker may split the state when modelling a call to the `malloc` function, creating a path of execution where the result of the function call is `NULL`, and one where the allocation was successful. This implies that the exploded graph grows exponentially, hence the naming.

The exploded graph is not isomorphic with the CFG, but it does have a natural projection to it: each exploded node can be mapped to a statement or an edge in the CFG.

The analyzer categorizes many modelling steps into events. Checkers can subscribe to one or several of these, and are notified by the core should they occur. InnerPointerChecker [17], for instance, is subscribed to the "end of a function call" event to mark the return values of method calls like `std::string::c_str()` as pointers to internal data, and to the "dead symbols" event to mark such data as released after going out of scope.

## 5.2 Representing an object

In C++, the proper initialization of objects of a record type is the responsibility of the special member function called the constructor. The constructor could be user-provided when the programmer specifies it or compiler provided if it is automatically generated. The improper implementation of the user-provided constructor is the primary source of uninitialized value misuse. Therefore, in this research, we are concentrating on validating the construction.

Naturally, any representation must respect fields and fields of fields (*subobjects*), which we will call *direct containment*. Inherited fields are also directly contained. However, non-null valid pointer objects refer to objects that may be uninitialized themselves. We will refer to pointees of pointer fields as *indirectly contained* objects.

```
1  struct A {
2    struct B {
3      int x;
4      int y = 0;
5    };
6
7    B b;
8    int *iptr = new int
       ;
9    B* bptr;
10   A() {}
11 };
```

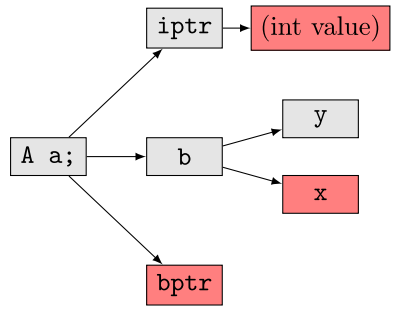



Figure 5: How an object is represented by the checker. Red nodes are uninitialized objects.

With that in mind, our proposed representation is a directed graph, where

- The root of this graph is the object we are analyzing,
- Every other node is also an object that is a union, a non-union record, dereferenceable, an array or of a primitive type,
- The parent of each node is the object that contains it.

It is easy to show that this representation is not always a directed tree: circular linked lists and pointers pointing to themselves are all directed cycles. However, by keeping track of the objects we already analyzed, we can disallow these con- structs, turning this representation into a directed tree.

We can also realize that in this directed tree every leaf is either a null pointer, an undefined pointer, an array, a primitive object, or a pointer that that points to an already analyzed object, meaning that they can be represented as numerical values. This is important, as the analyzer may only mark such objects as uninitialized.

By subscribing our checker to the *end of a function call* event, we can inspect every object after the end of a constructor call. By traversing the above graph, we can detect all directly and indirectly contained objects, and emit reports for each of them.

Our prototype traverses this graph recursively in a depth-first manner, and after each descent to a child node, constructs an informative object to keep track of the path leading to the current node. By the time it reaches a leaf, it has constructed an informative list that contains every information needed to reach it. Should that leaf be in an indeterministic state, a report is emitted, and a helpful warning message is constructed from the informative list. For Figure 5, our prototype would report that `this->b.x`, `this->iptr`'s pointee and `this->bptr` is left uninitialized after the constructor call. However, since the analyzer keeps track of the dynamic type of

the pointers during symbolic execution, the warning message could be incorrect, if we report a derived class' field as uninitialized through its base class' typed pointer. To solve this problem, the checker stores dynamic type information as well in the constructed informative list, so the constructed warning message could contain casts the actual dynamic type.

As constructors are called for fields during construction, we will only run our checker when function call stack does not contain other constructor calls.

## 5.3 Heuristics

In static analysis, we can classify results as true positive, if the analyzer correctly identified a programming error, true negative if the analyzer correctly identified the lack of a programming error, false positives if the analyzer reported a programming error despite the code being correct, and false negatives when the analyzer did not report incorrect code. These definitions, however, describe uninitialized object related reports rather poorly. While correctly identified and reported uninitialized objects are by definition true positives, these objects may have been left as such intentionally. An essential aspect of this problem is that not initializing a variable is not an error, only the reading of an uninitialized value. In fact, when the programmer cannot supply a meaningful default value (e.g. declaring a variable as a buffer), initialization could result in a performance loss.

For this reason, one of the main goals of our research is to identify which uninitialized variables are most likely to result in misuse and undefined behaviour. This implies that we have to reason about the intention of the programmer and suppress some reports, turning them into false negatives. The following section will detail how we try to find an optimal true positive/false negative ratio.

We made our checker configurable, allowing us to enable, disable or fine-tune some of our heuristics for a particular project.

### 5.3.1 Arrays

Before C++11, elements of dynamically allocated arrays could not be initialized. Even stack-allocated arrays are often used as buffers, which was consistent with the results of our findings, so our prototype ignores arrays.

### 5.3.2 No initialized field

Through testing our prototype, we concluded that objects that do not initialize a single one of their fields are often created intentionally. However, this heuristic can result in a higher amount of false negatives than maybe desired, so we made it toggleable.

### 5.3.3 Pointer chasing

Indirect containment raises a philosophical question: Is an object responsible for leaving its pointee object in a fully deterministic state? One perspective we could

```
1  struct PhysicalProperty {
2    int volume, area;
3    enum Kind { VOL, AREA } kind;
4
5    PhysicalProperty(Kind k) : kind(k)
       {
6      switch(k) {
7      case VOL:
8        volume = 0;
9        break;
10     case AREA:
11       area = 0;
12       break;
13     }
14   }
15
16   int getVolume() const {
17     assert(kind == VOL);
18     return volume;
19   }
20
21   int getArea() const {
22     assert(kind == AREA);
23     return area;
24   }
25 };
```

Figure 6: `PhysicalProperty` doesn't initialize all data members, but „guards" against uninitialized value misuse.

take is to guess whether the objects *owns* the pointee. However, ownership is a conceptually popular, but non-standardized concept within C++ [11]. This, and the current faults in the analyzer lead to us to not analyze pointees (or *chasing pointers*) by default.

### 5.3.4 Guarded field analysis

Consider the code snippet in Figure 6. Although `PhysicalProperty` will leave one of its fields uninitialized on every instantiation, we cannot encounter an uninitialized object related error runtime. Similar constructs within the LLVM codebase is very popular and will trigger a report from our checker, despite the lack of a programmer error.

We will call any statement that could prevent the execution from reaching and reading an uninitialized field a *guard*. We call a field *guarded* if every read of it control depends on a guard.

Unfortunately, it is hard to guess compile-time whether a field is guarded, as the argument of if statements might not be correlated to whether the field is initialized. We implemented a primitive heuristic to solve this problem using AST matchers on the object's record definition, analyzing whether the field is public, and is read before a guard in the code. Due to the reasons mentioned above, this is a very rough estimate, and this analysis is disabled by default.

#### 5.3.5 Known to be safely uninitialized fields

In some instances, we might want to ignore particular objects of a particular type intentionally, or if they have a specific variable name. For this reason, our prototype is configurable with a regular expression, and if it matches a variable's name or it's type, we ignore it.

## 6 Evaluation

We evaluated our prototype on several large, open-source C++ projects, such as Rtags [3], LLVM [19], Xerces [1], CppCheck [21], Bitcoin [31] with a variety of configurations. We used the open-source program csa-testbench [24], which helped us compare the results of different configurations of our checker with ease. We used CodeChecker [7] to visualize our results. We also ran CppCheck on said projects and compared its results with the one our solution generated.

We purposely chose projects with diverse design patterns. For example, the LLVM project relies almost purely on its own libraries, and being a compiler, it is very performance-critical. The C++ indexer Rtags uses several third-party libraries and houses a variety of coding styles, some more performance-critical then others. We found that the more performance-critical a project is, the more likely it is that the code takes advantage of not initializing every variable.

As mentioned in Section 5.3, it can be challenging to find good metrics on the quality of the reports. While the authors were researching an algorithm which enforces the idiom of initializing every variable, there are several reports (especially in the code generation libraries of LLVM) that we feel justifies going against it. This implies that judging whether a report is meaningful or not is debatable. For this reason, we categorized the results into three categories: we say a report *useful* if the lack of initialization had probably little to no effect on performance and is error-prone, *questionable* if judging from the code context the lack of initialization is appropriate, and *false positive* if the report was incorrect. We summarized our results in Table 1.

According to the C++ initialization rules, we found only a single false positive where the analyzer disregarded an in-class initialization, though this is a fault of the analyzer's core. We found that the reports from our prototype with the

Table 1: Reports from our prototype in the first 4 columns, and from CppCheck in the last on large, open source projects. Reports are shown in the format "all/**useful**/false positive/questionable".

| | Default | Pointer chasing | Pedantic | Guarded fields ignored | CppCheck |
|---|---|---|---|---|---|
| Rtags | 1/**1**/0/0 | 6/**1**/0/5 | 1/**1**/0/0 | 1/**1**/0/0 | 5/**1**/0/4 |
| LLVM | 46/**18**/1/27 | 88/**18**/1/69 | 48/**20**/1/27 | 43/**18**/1/24 | 4/**1**/0/3 |
| Xerces | 1/**1**/0/0 | 1/**1**/0/0 | 1/**1**/0/0 | 1/**1**/0/0 | 7/**4**/0/3 |
| CppCheck | 0/**0**/0/0 | 0/**0**/0/0 | 0/**0**/0/0 | 0/**0**/0/0 | 0/**0**/0/0 |
| Bitcoin | 5/**5**/0/0 | 5/**5**/0/0 | 5/**5**/0/0 | 5/**5**/0/0 | 12/**5**/0/7 |

default configuration were overwhelmingly useful except for LLVM. Despite most uninitialized variable finds in that project were of little value, even there we were able to find and patch error-prone code.

We found code pattens in nearly all projects that clearly go against idiomatic C++ code. Such code patterns include not letting the compiler generate constructors by defaulting them with `= default`, not initializing variables that are removed in non-debug builds, leaving uninitialized fields of non-POD classes public, or storing auxiliary data such as counters that would be more appropriate as function-local variables.

Contrary to our expectations, both reports that are present with the Pedantic option enabled but not with default configurations were useful, showing constructors that should have been defaulted. Though the number of reports increased significantly after enabling pointer chasing, it mostly lead us to find pointers to buffers that were handled correctly in the class, making them uninteresting in all reports found in LLVM and Rtags. LLVM uses guarded fields extensively, but we were only able to suppress reports based on this information in 3 cases.

Comparatively, CppCheck had the least amount of reports on LLVM except for its own codebase, significantly less than our prototype both in terms of count and useful finds, but had more on Rtags, Xerces and Bitcoin. Shockingly, there was only a single report found by both CppCheck and out prototype in Xerces. This supports the conclusion of other findings on static analyses that to find more bugs, it is better to use multiple tools [2, 33].

It should be noted that unlike our prototype, CppCheck constructs a warning message per uninitialized field, rather than per constructor call, so we regarded multiple warnings originating from the same constructor as one.

This checker, under the name of `optin.cplusplus.UninitializedObject` has been a part of the analyzer since it's 9.0.0 release [32], and has been used by various industrial parties, such as Firefox[2], Apple[3], Google[4] and Ericsson[5].

[2] https://reviews.llvm.org/D45532#1145512
[3] http://lists.llvm.org/pipermail/cfe-dev/2018-August/058905.html
[4] https://reviews.llvm.org/D58573#1477581
[5] https://reviews.llvm.org/D58573#1425837

# 7 Future works

It is hard to find an ideal true positive/false negative ratio to make our reports meaningful enough without suppressing too many of them. More research into new heuristics and improving existing ones is where most of our future projects lay.

Guarded field analysis could benefit from being implemented using dataflow analysis instead of AST matching. While finding a correlation in between the guard statement and whether the field is initialized is theoretically impossible, clever heuristics could help on this matter.

Pointer chasing suffers from some poor modelling techniques within the analyzer's core, which is not directly related to our implementation. Also, heap allocated objects are not yet modelled at all. Improving these within the core and better defining in which cases we want to report uninitialized pointees could eventually enable us to enable pointer chasing by default.

While there are no constructors in C, it is worth investigating whether our C++ prototype could be used for analyzing C code. One approach would be to note when an object of a great enough size is created, and when the function call in which its created ends, analyze that object with our proposed technique.

A popular technique in C++ programming is the *pimpl idiom* [26], where the part of the class' definition is implemented through an opaque pointer. The Clang Static Analyzer, by default, can only analyze a single translation unit at a time, so it may be unable to reason about opaque pointers if the definition of a function or a class lies a translation unit different than what is being analyzed. Cross translation unit (*CTU*) analysis [10] can be used to acquire the definition. It should be investigated whether our prototype is conformant with CTU.

# 8 Conclusion

Static analysis of C/C++ code can be used to detect uninitialized variables, which are a common source of undefined behaviour. While more prone to false positives, static analysis has a far greater code coverage compared to dynamic analysis. We argued against analyzing only fundamental objects and proposed an accurate representation of record objects in the form of a directed tree. Our prototype, implemented in the Clang Static Analyzer, can traverse this graph to detect uninitialized variables for each object after the end of its constructor call. We proposed a variety of heuristics to reduce the number of reports emitted by this prototype, focusing on uninitialized variables most likely to be read. Evaluation of large open-source projects lead us to discover several records that are likely to leave some fields uninitialized unintentionally and are prone to misuse. While we generally found the results of our prototype meaningful, we plan to add new heuristics and enhance existing ones to reduce the further number of uninteresting reports on performance-critical projects.

This checker, under the name of `optin.cplusplus.UninitializedObject` has been a part of the analyzer since it's 9.0.0 release [32], and has been used by various industrial parties, such as Firefox, Apple, Google and Ericsson.

## References

[1] Apache Software Foundation. Apache Xerces. https://xerces.apache.org/.

[2] Arusoaie, Andrei, Ciobâca, Stefan, Craciun, Vlad, Gavrilut, Dragos, and Lucanu, Dorel. A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168. IEEE, 2017. DOI: 10.1109/SYNASC.2017.00035.

[3] Bakken, Anders. Rtags. http://www.rtags.net.

[4] Bruening, Derek and Zhao, Qin. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011. DOI: 10.1109/CGO.2011.5764689.

[5] Calcagno, Cristiano, Distefano, Dino, Dubreil, Jérémy, Gabi, Dominik, Hooimeijer, Pieter, Luca, Martino, O'Hearn, Peter, Papakonstantinou, Irene, Purbrick, Jim, and Rodriguez, Dulma. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015. DOI: 10.4204/eptcs.188.2.

[6] Cifuentes, Cristina, Hoermann, Christian, Keynes, Nathan, Li, Lian, Long, Simon, Mealy, Erica, Mounteney, Michael, and Scholz, Bernhard. Begbunch: Benchmarking for c bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 16–20, 2009. DOI: 10.1145/1555860.1555866.

[7] Ericsson. CodeChecker. https://github.com/Ericsson/codechecker.

[8] GrammaTech. Codesonar. https://www.grammatech.com/products/codesonar.

[9] Hampapuram, Hari, Yang, Yue, and Das, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. *ACM SIGSOFT Software Engineering Notes*, 31(1):52–58, 2005. DOI: 10.1145/1108768.1108808.

[10] Horváth, Gábor, Gera, Zoltán, Krupp, Dániel, Porkoláb, Zoltán, and Szécsi, Péter. Cross translational unit analysis in Clang static analyzer: Prototype and measurements. URL: https://llvm.org/devmtg/2017-03//assets/slides/cross_translation_unit_analysis_in_clang_static_analyzer.pdf, 2017. European LLVM Developers Meeting, Saarbrücken.

[11] Horváth, Gábor and Pataki, Norbert. Categorization of C++ classes for static lifetime analysis. In Eleftherakis, George, Lazarova, Milena, Aleksieva-Petrova, Adelina, and Tasheva, Antoniya, editors, *Proceedings of the 9th Balkan*

*Conference on Informatics, BCI 2019, Sofia, Bulgaria, September 26-28, 2019*, pages 21:1–21:7. ACM, 2019. DOI: 10.1145/3351556.3351559.

[12] Horváth, Gábor, Szécsi, Péter, Gera, Zoltán, Krupp, Dániel, and Pataki, Norbert. Implementation and evaluation of cross translation unit symbolic execution for c family languages. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 428–428. ACM, 2018. DOI: 10.1145/3183440.3195041.

[13] Programming languages – C++. Standard, International Organization for Standardization, Geneva, Switzerland, December 2017.

[14] Jana, Anushri and Naik, Ravindra. Precise detection of uninitialized variables using dynamic analysis-extending to aggregate and vector types. In *2012 19th Working Conference on Reverse Engineering*, pages 197–201. IEEE, 2012. DOI: 10.1109/WCRE.2012.29.

[15] Klocwork. Klocwork. https://www.roguewave.com/products-services/klocwork.

[16] Kovács, Réka and Horváth, Gábor. An initial prototype of tiered constraint solving in the clang static analyzer. *Studia Universitatis Babes-Bolyai, Informatica*, 63(2), 2018. DOI: 10.24193/subbi.2018.2.06.

[17] Kovács, Réka, Horváth, Gábor, and Porkoláb, Zoltán. Detecting C++ lifetime errors with symbolic execution. In *Proceedings of the 9th Balkan Conference on Informatics*, pages 1–6, 2019. DOI: 10.1145/3351556.3351585.

[18] Kremenek, Ted. Finding software bugs with the Clang static analyzer. https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf. LLVM Developers' Meeting, 2008.

[19] Lattner, Chris and Adve, Vikram. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004. DOI: 10.1109/CGO.2004.1281665.

[20] Lu, Shan, Li, Zhenmin, Qin, Feng, Tan, Lin, Zhou, Pin, and Zhou, Yuanyuan. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Volume 5, 2005.

[21] Marjamäki, Daniel. Cppcheck: A tool for static C/C++ code analysis. URL: https://cppcheck.sourceforge.io/, 2013.

[22] Márton, Gábor and Porkoláb, Zoltán. Compile-time function call interception for testing in C/C++. *Studia Universitatis Babes-Bolyai, Informatica*, 63(1), 2018. DOI: 10.24193/subbi.2018.1.02.

[23] Nethercote, Nicholas and Seward, Julian. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 89–100. ACM, 2007. DOI: 10.1145/1250734.1250746.

[24] Nikolett Kovács, Réka, Horváth, Gábor, and Szécsi, Péter. Towards proper differential analysis of static analysis engine changes. In *The 11th Conference of PhD Students in Computer Science*, 2018.

[25] Porkoláb, Zoltán. Multiparadigm programming: Constructors, destructors, operators, 2019. http://gsd.web.elte.hu/lectures/multi/slides/constructor.pdf.

[26] Reddy, Martin. *API Design for C++*. Elsevier, 2011. DOI: 10.1016/c2010-0-65832-9.

[27] Seward, Julian and Nethercote, Nicholas. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit.

[28] Stepanov, Evgeniy and Serebryany, Konstantin. Memorysanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE Computer Society, 2015. DOI: 10.1109/CGO.2015.7054186.

[29] Synopsys, Inc. Coverity. https://scan.coverity.com/.

[30] Szalay, Richárd and Porkoláb, Zoltán. Visualising compiler-generated special member functions of C++ types. In *Proceedings of Collaboration, Software and Services in Information Society*, pages 55–58, 2018.

[31] The Bitcoin Core. Bitcoin Core. https://bitcoincore.org/.

[32] The LLVM Foundation. Clang 9.0.0 release notes, 2019. https://releases.llvm.org/9.0.0/tools/clang/docs/ReleaseNotes.html#static-analyzer.

[33] Zitser, Misha, Lippmann, Richard, and Leek, Tim. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, 2004. DOI: 10.1145/1029894.1029911.

Contents



**ISSN 0324—721 X (Print)**
**ISSN 2676—993 X (Online)**

Editor-in-Chief: Tibor Csendes