Towards Generic Refactoring

ARTICLE · APRIL 2002

DOI: 10.1145/570186.570188 · Source: arXiv

1 AUTHOR:

Ralf Lämmel
Universität Koblenz-Landau
114 PUBLICATIONS **2,361** CITATIONS

Towards Generic Refactoring

1st February 2008

Ralf Lämmel
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

We study program refactoring while considering the language or even the programming paradigm as a parameter. We use typed functional programs, namely Haskell programs, as the specification medium for a corresponding refactoring framework. In order to detach ourselves from language syntax, our specifications adhere to the following style. (I) As for primitive algorithms for program analysis and transformation, we employ generic function combinators supporting generic traversal and polymorphic functions refined by ad-hoc cases. (II) As for the language abstractions involved in refactorings, we design a dedicated multi-parameter class. This class can be instantiated for abstractions as present in various languages, e.g., Java, Prolog or Haskell.

1998 ACM Computing Classification System: D.1.1, D.1.2, D.2.1, D.2.3, D.2.13, D.3.1, I.1.1, I.1.2, I.1.3
Keywords and Phrases: refactoring, program transformation, reuse, generic programming, frameworks, functional programming

1. Introduction

Refactoring The very term refactoring has recently been pushed a lot in the context of object-oriented programming, but the related idea of semantics-preserving program transformation is as old as high-level programming. A program refactoring is typically meant to improve the internal structure of a program [7], be it to make the program more comprehensible, to enable its reuse, or to prepare a subsequent adaption. In a broader sense, one might also include program transformation in the sense of refinement or optimization. Let us consider a standard refactoring, namely the *extraction* of an abstraction from a given program. Extraction (say, folding) introduces a name for a previously anonymous piece of code. Obviously, the established abstraction creates potential for reuse. Also, the extracted functionality is maybe more concisely documented by the abstraction, or more accessible for a subsequent adaptation. Depending on the language which we want to deal with, different kinds of code fragments and abstractions are relevant. Here is a list of some classes of languages, corresponding syntactical domains involved in extraction, and references to previous work on program transformation with relevance for refactoring:

Class of languages	*Focused fragment*	*Extracted abstraction*	*References*
XML/DTD	content particle	element type	[16]
Logic programming	literal	predicate	[25, 24]
Preprocessing	code fragment	macro	[6, 11]
Functional programming	expression	function	[24, 1, 13]
OO programming	statement	method	[21, 7, 17]
Syntax definition	EBNF phrase	nonterminal	[23, 14]

Genericity One might wonder what the commonalities of program refactorings (such as extraction) are if we attempt to consider the language or even the programming paradigm as a parameter. To address these problems, we develop a language-independent refactoring framework. A language-independent formulation of refactoring has not been suggested or attempted before, but it turns out to be informative and useful. As a first indication, the commonalities, which we are able to capture, are of the following kind:

- There are general notions of focus, scope, and abstraction.
- One can navigate through programs, e.g., nested lists of abstractions.
- There is an interface for name analyses.
- A refactoring can be described by a number of steps of the following kind:
  - Identification of fragments of a certain type and location;
  - Destruction, analysis, and construction;
  - Checking for pre- and postconditions;
  - Placing, removing or replacing a focus.
- There are parameters for language-specific ingredients.

The refactoring framework is specified in Haskell 98 [8] with one common extension which is used for convenience, namely functional dependencies [9]. We rely on the *Strafunski* style of generic functional programming [17] (joint work of the author with Joost Visser; see `http://www.cs.vu.nl/Strafunski/`). This approach is based on generic function combinators including combinators for generic traversal and update of polymorphic functions by ad-hoc cases. The interested reader is referred to [15] for the foundation of generic programming with *Strafunski*-like function combinators.

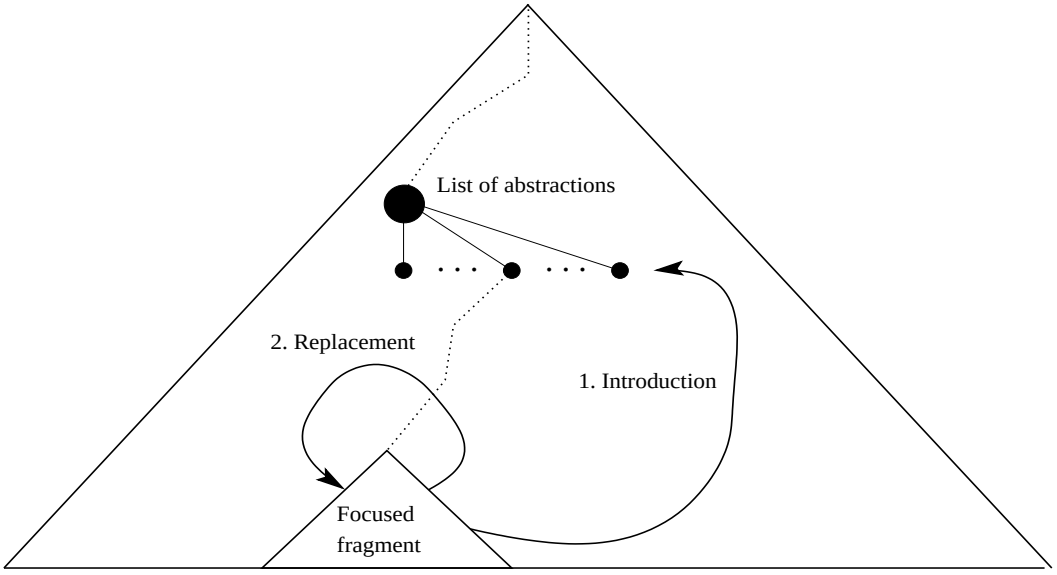

Figure 1: Illustration of extraction

Running example In Figure 1, we illustrate some abstract properties of extraction. The figure shows the skeleton of a syntax tree. We assume that there is a focused code fragment which is meant to constitute the body of a new abstraction. The first major step of extraction is to construct an abstraction from the focused piece of code, and to add it to the relevant scope. Adding the new abstraction leads to an intermediate result of extraction. We use the term *introduction* to denote the process of adding an abstraction. In fact, introduction is another refactoring. We assume that abstractions are hosted in possibly nested lists of abstractions. This clearly implies that we need to be prepared for nested scopes. The proper scope for the new abstraction is the next list of abstractions *above* the focus. The introduction refactoring has to be restricted not to add an abstraction which interferes with existing ones. Once, the abstraction has been added, the focus can be replaced by an application of the abstraction. We have to take free names in the focused fragment into account. These names constitute the formal parameters of the new abstraction, and the actual parameters of the application for focus replacement. At certain points along a refactoring, we might have to check specific properties of some fragments.

To give a language-specific example of extraction, consider the extraction of a Java method. In this case, the focused piece of code is a statement. One has to check that the potential compound statement does not contain a `return` statement since the `return` statement will lead to a different control-flow once placed in another method. One also has to check that there are no assignments to non-local variables since these side effects would not be propagated. The focused statement constitutes the body of the extracted method. The arguments of the emerging method are retrieved by a free variable analysis on the focused piece of code. The focused statement will be finally replaced by a method invocation.

Schedule In Section 2, the Strafunski style of generic functional programming is briefly recalled. This style is crucial for our specification of language-independent refactorings. In Section 3, the framework for refactoring is worked out. We aim at a concise specification where the framework is equipped with a number of hot spots using different parameterization techniques. In Section 4, an instantiation of the framework for a Java-like language is worked out. In Section 5, the paper is concluded.

Acknowledgement I am very grateful for the collaboration with Joost Visser in the *Strafunski* project. The author was supported in part by the Dutch research organisation NWO in the project 612.014.006 ("*Generation of program transformation systems*"). The author gave presentations on generic refactoring at the CWI meeting on language design assistants, action semantics, modular language definitions in Amsterdam, The Netherlands, August 24, 2001, and at the 56th IFIP WG 2.1 meeting ("*Algorithmic languages and calculi*") on Ameland, The Netherlands, September 10-14, 2001.

2. Generic functional programming

Strafunski-like generic functions [17, 15] enable a combinator-based approach to typeful generic functional programming which is particularly suited for generic program schemes dealing with term traversal. In this section, we briefly recall the style to the extent needed for the subsequent specification of the refactoring framework.

2.1 Generic function types

We want to be able to write generic functions on terms over algebraic datatypes, that is, on term types. We want these functions to be monadic so that we can model partiality, state passing etc. [29]. It turns out that we need two kinds of generic functions, namely type-preserving and type-unifying ones. The former kind of function is suitable for transforming a term while preserving its type whereas the latter is suitable for analysing or reducing a term. Type-preservation adheres to the type scheme

$\forall x.x \to m\ x$ where m is the type-constructor parameter for the monad. Similarly, type-unification for a given type a adheres to the scheme $\forall x.x \to m\ a$. These type schemes should not be interpreted in the restrictive manner of parametric polymorphism. We need generic functions which also allow for generic traversal and ad-hoc cases. In the present paper, we detach ourselves from the modelling details for generic functions in Haskell. We assume that the generic function types for *t*ype-*p*reserving and *t*ype-*u*nifying functions are available via two Haskell datatypes:

$$\begin{array}{lll} \textbf{data} & \mathit{Monad}\ m \Rightarrow \mathit{TP}\ m & = \ \ldots \\ \textbf{data} & \mathit{Monad}\ m \Rightarrow \mathit{TU}\ a\ m & = \ \ldots \end{array}$$

We can formulate the aforementioned intuitive type-schemes as a contract as follows. We assume two dedicated combinators for generic function application—one for type-preserving functions, and another for type-unifying functions:

$$\begin{array}{lll} \mathit{applyTP} & :: & (\mathit{Monad}\ m, \mathit{Term}\ t) \Rightarrow \mathit{TP}\ m \to t \to m\ t \\ \mathit{applyTU} & :: & (\mathit{Monad}\ m, \mathit{Term}\ t) \Rightarrow \mathit{TU}\ a\ m \to t \to m\ a \end{array}$$

In these type declarations, the class *Term* comprises all term types. Let us read the declaration of *applyTP*: If a type-preserving function is applied to a term of type t, then the result is also of type t, or of type $m\ t$ if we are precise and account for the monadic style; similarly for type-unifying function. Note that we need special application combinators because our generic functions are not plain Haskell functions. They are rather opaque terms of type $\mathit{TP}\ m$ and $\mathit{TU}\ a\ m$. We will gradually rehash a few more ordinary function combinators to may use them for generic functions, too.

2.2 Function combinators

In Figure 2, we provide a complete list of all basic functions combinators we need. Let us explain these combinators block-wise. The first block deals with combinators as they are known from (parametric) polymorphic programming. In fact, we can provide parametric polymorphic "prototypes" for the functions combinators in the first block:

$$\begin{array}{ll} \mathit{idP} = \mathit{return} & \text{-- monadic identity} \\ \mathit{constP}\ a = \mathit{const}\ (\mathit{return}\ a) & \text{-- monadic constant function} \\ f\ `\mathit{seqP}`\ g = \lambda x \to f\ x \gg\!= g & \text{-- monadic function composition} \\ f\ `\mathit{letP}`\ g = \lambda x \to f\ x \gg\!= \lambda y \to g\ y\ x & \text{-- monadic let where x is free} \\ \mathit{failP} = \mathit{const}\ \mathit{mzero} & \text{-- monadic failure} \\ f\ `\mathit{choiceP}`\ g = \lambda x \to f\ x\ `\mathit{mplus}`\ g\ x & \text{-- monadic nondetermistic function application} \end{array}$$

The prototypes embody familiar patterns in (monadic) functional programming. The actual details of lifting the prototypes to the generic combinator level are omitted since we do not discuss the actual definitions of the datatypes $\mathit{TP}\ m$ and $\mathit{TU}\ a\ m$. Comparing the list of prototypes and the generic combinators from the first block, one can see that most prototypes can be instantiated for both the type-preserving and the type-unifying case with the only exceptions being *idTP* and *constTU*. This is because the identity function is necessarily type-preserving, and the constant function is unavoidably type-unifying.

The second block in Figure 2 provides combinators for generic traversal. The *all* couple applies the argument function to *all* immediate subterms (say, children). The *one* couple applies the argument function to *one* immediate subterm. The type-preserving combinators *allTP* and *oneTP* preserve the outermost term constructor. In the type-unifying case, the overall shape of the input term cannot be preserved for simple typing arguments. As for *oneTU*, one immediate subterm is processed, and this gives immediately the result of the type-unifying traversal. As for *allTU*, all children are processed and the intermediate results are reduced with the binary operator of a monoid. Hence, in this case, the unified result type has to correspond to a monoid.

Parametric polymorphic heritage

$$
\begin{array}{ll}
idTP & :: \; Monad\ m \Rightarrow TP\ m \\
constTU & :: \; Monad\ m \Rightarrow m\ a \rightarrow TU\ a\ m \\
seqTP & :: \; Monad\ m \Rightarrow TP\ m \rightarrow TP\ m \rightarrow TP\ m \\
letTP & :: \; Monad\ m \Rightarrow TU\ a\ m \rightarrow (a \rightarrow TP\ m) \rightarrow TP\ m \\
seqTU & :: \; Monad\ m \Rightarrow TP\ m \rightarrow TU\ a\ m \rightarrow TU\ a\ m \\
letTU & :: \; Monad\ m \Rightarrow TU\ a\ m \rightarrow (a \rightarrow TU\ b\ m) \rightarrow TU\ b\ m \\
failTP & :: \; MonadPlus\ m \Rightarrow TP\ m \\
failTU & :: \; MonadPlus\ m \Rightarrow TU\ a\ m \\
choiceTP & :: \; MonadPlus\ m \Rightarrow TP\ m \rightarrow TP\ m \rightarrow TP\ m \\
choiceTU & :: \; MonadPlus\ m \Rightarrow TU\ a\ m \rightarrow TU\ a\ m \rightarrow TU\ a\ m
\end{array}
$$

Generic traversal

$$
\begin{array}{ll}
allTP & :: \; Monad\ m \Rightarrow TP\ m \rightarrow TP\ m \\
oneTP & :: \; MonadPlus\ m \Rightarrow TP\ m \rightarrow TP\ m \\
allTU & :: \; (Monad\ m, Monoid\ a) \Rightarrow TU\ a\ m \rightarrow TU\ a\ m \\
oneTU & :: \; MonadPlus\ m \Rightarrow TU\ a\ m \rightarrow TU\ a\ m
\end{array}
$$

Generic function update

$$
\begin{array}{ll}
adhocTP & :: \; (Monad\ m, Term\ t) \Rightarrow TP\ m \rightarrow (t \rightarrow m\ t) \rightarrow TP\ m \\
adhocTU & :: \; (Monad\ m, Term\ t) \Rightarrow TU\ a\ m \rightarrow (t \rightarrow m\ a) \rightarrow TU\ a\ m
\end{array}
$$

Figure 2: Basic combinators for generic functions

The third and the last block in the figure deals with function update. The two combinators *adhocTP* and *adhocTU* enable one to update a generic function so that it exposes type-dependent behaviour. In other words, one can construct ad-hoc polymorphic functions by a kind type case. This is indispensable for generic traversals which are supposed to interact with the involved term types. The idea is that one starts with a parametric polymorphic function like *idTP* or *failTP*, and then establishes specific behaviour for distinguished term types via generic function update. That is, when an updated function is applied to a term, the ad-hoc case (i.e., the second argument of *adhocTP* or *adhocTU*) will be applied if applicable as for the type, that is, if the updated term type coincides with the term type at hand. Otherwise, we resort to the updated function (say, the generic default, i.e., the first argument of *adhocTP* or *adhocTU*).

2.3 Strafunski in action

To illustrate the basic combinators, let us consider some examples. Also, we should provide some reusable traversal schemes and other generic functions which are frequently needed. To start with, let us give a simple example of a combinator defined in terms of several basic ones. The following combinator *combTU* lifts a binary operation to the generic level.

$$
\begin{array}{lcl}
combTU\ o\ s\ s' & = & s\ `letTU`\ \lambda a \rightarrow \\
 & & s'\ `letTU`\ \lambda b \rightarrow \\
 & & constTU\ (o\ a\ b)
\end{array}
$$

The combinator takes a binary combinator o, and two type-unifying functions s and s'. A type-unifying function is constructed which passes through the incoming term to both s and s' and combines the intermediate results of these applications via o.

```
selectStatement :: (MonadPlus m, Term t) ⇒ t → m Statement
selectStatement = applyTU selectStatementStrategy
  where
    selectStatementStrategy :: MonadPlus m ⇒ TU Statement m
    selectStatementStrategy =
      (adhocTU failTU
               (λstat → case stat of
                          StatementFocus stat' → return stat'
                          _ → mzero))
      `choiceTU`
      oneTU selectStatementStrategy
```

Figure 3: A function that selects Java-like statements in a focus

Let us consider a slightly more involved example related to the topic of refactoring. In fact, we consider an example dealing with a primitive operation involved in some Java refactorings. In Figure 3, a function *selectStatement* is defined which, given a term, looks up the statement which the focus is placed on if any. To this end, we assume that focused statements are surrounded by the term constructor *StatementFocus*. The function *selectStatement* defines a local type-unifying function *selectStatementStrategy*. The function obviously has to be specific about statements. For that reason, we use *adhocTU* to combine a *Statement*-case and a default case. The specific case actually examines the given statement in order to unwrap the focus term constructor if present. As for the function constructed with *adhocTU*, there are two options how failure might arise. Either we are not faced with a statement altogether (cf. *failTU*), or the focus is not placed on the given statement (cf. catch-all in **case**). The top-level application of *choiceTU* makes it possible to recover from failure. The second alternative in the choice recursively descends into the given term via an *oneTU* traversal.

It is worth mentioning that the above problem of locating a term in a focus can be expressed in a more generic, and hence more reusable fashion. We will ultimately attempt such more generic definitions. In this manner, we will approach to a style of generic refactoring. The present definition of *selectStatement* is not generic because it talks explicitly about statements, about the focus term constructor *StatementFocus* for statements, and it defines a traversal scheme from scratch. By generic refactoring we mean that language-independent refactoring functionality is identified, and that reusable and completely generic traversal schemes are employed.

In Figure 4, we show a fragment of a library for generic programming. These reusable application-independent combinators are needed in the paper. Before we explain all these definitions, let us point out a convention used in the figure. We use names ending on "...*S*" for combinators which can be overloaded for the type-preserving and the type-unifying case. This includes the basic combinators *seqS*, *letS*, *failS*, *choiceS*, *allS*, *oneS*, and *adhocS* introduced separately for *TP m* and *TU a m* before. We can still use all these operators with the "..*TP*" and "..*TU*" prefixes, if we want to express that they are used in a context that specifically requires *TP m* or *TU a m*.

Let us briefly explain the definitions in Figure 4. The first combinator *monoS* lifts a function on a term type to the generic level by assuming failure as generic default. In fact, failure is a prominent kind of generic default. Ultimately, we define a few (overloaded) traversal schemes. Firstly, we define the schemes *oncetdS* and *oncebuS* which attempt to apply the given function argument once somewhere in the tree. These two schemes simply differ in the vertical direction of search. That is, *oncetdS* and *oncebuS* stand for "once top-down" or "once bottom-up", respectively. The schemes *aboveS* is concerned with paths in trees (say, terms). The combinator takes a generic function s for selection or transformation, and another generic function s' serving as a kind of generic predicate. The goal

$$
\begin{array}{lcl}
monoS\ f & = & adhocS\ failS\ f \\
oncetdS\ s & = & s\ `choiceS`\ (oneS\ (oncetdS\ s)) \\
oncebuS\ s & = & (oneS\ (oncebuS\ s))\ `choiceS`\ s \\
s\ `aboveS`\ s' & = & oncebuS\ ((oncetdTU\ (oneTU\ s'))\ `letS`\ (\lambda\_ \rightarrow s)) \\
propagateS\ e\ s'\ s & = & (s\ e)\ `choiceS`\ (s'\ e\ `letS`\ (\lambda e' \rightarrow oneS\ (propagateS\ e'\ s'\ s)))
\end{array}
$$

Figure 4: Specifications of some reusable generic functions

is to apply s to a node above another node which meets the condition s'. In order to minimise the distance between the two nodes, the overall traversal is dominated by a bottom-up traversal to find the bottom-most node admitting application of s while s' is met below this node, that is, the condition is checked in top-down manner. The last traversal scheme *propagateS* favours top-down traversal as *oncetdS* does, but in addition propagation is performed. The scheme takes an initial parameter e, a type-unifying scheme s' to update the parameter before descending into the children, and the actual scheme s for selection or transformation. As an aside, the scheme illustrates that we do not necessarily need to employ the monad parameter for effects like propagation (or accumulation as well) but the effect handling can be largely hidden in the traversal scheme.

As a simple exercise for applying the defined combinators, let us rephrase the function *selectStatement* from Figure 3. We employ *monoTU* to lift the **case** for focus identification to the generic level. We also use *oncetdS* to describe the traversal underlying focus selection. The resulting code is much more concise:

```
selectStatement = applyTU (oncetdTU (monoTU (
                    λstat → case stat of
                              StatementFocus stat' → return stat'
                              _ → mzero)))
```

3. The refactoring framework

The framework for refactoring is structured as follows. Firstly, there are several generic algorithms to perform simple analyses and transformations as needed in the course of refactoring, e.g., to operate in a focus, or determine free variables in a certain scope. Secondly, there is an interface to deal with abstractions of a language. Ultimately, we can define refactorings in terms of the abstraction interface and the generic algorithms. The specifications of both the generic algorithms and the refactorings carry formal parameters which need to be instantiated to obtain language-dependent variants. These parameters and the obligation to provide instances of the abstraction interface form the hot spots of the refactoring framework. For brevity, we only provide detailed discussions of two examples of parameterized program refactorings, namely extraction and introduction.

3.1 Generic algorithms

A specification of a refactoring should preferably be composed from simple reusable transformations. In addition, a refactoring employs analyses to determine parameters required for some transformation step, or to ensure some pre- or postcondition. Corresponding generic algorithms are presented in the sequel. Firstly, we discuss functionality to operate on a focus or a scope. Secondly, we provide algorithms to determine free or bound names in different manners.

Select the focus

```
selectFocus :: (MonadPlus m, Term f, Term t)
            ⇒ (f → m f)          -- Get focus
            → t                  -- Input term
            → m f                -- Focused term
selectFocus getFocus = applyTU (oncetdTU (monoTU getFocus))
```

Replace the focus

```
replaceFocus :: (MonadPlus m, Term t, Term t')
             ⇒ (t → m t)         -- Transform focus
             → t'                -- Input term
             → m t'              -- Output term
replaceFocus trafoFocus = applyTP (oncetdTP (monoTP trafoFocus))
```

Mark the host of the focus

```
markHost :: (MonadPlus m, Term f, Term h, Term t)
         ⇒ (f → m f)             -- Get focus
         → (h → h)               -- Set host
         → t                     -- Input term
         → m t                   -- Output term
markHost getFocus setHost
  = applyTP ((monoTP (return ∘ setHost))
             `aboveTP`
             (monoTU getFocus))
```

Figure 5: Functions to deal with focus and scope

Focus and scope In Figure 5, we specify functions to select a focused term, to replace a term in the focus by another term, and to mark the host of a focused term in a certain way. Note the type of these functions. These are ordinary polymorphic functions but they internally employ generic functions in order to perform traversal for the relevant selection, replacement, or marking. Let us explain the three functions in some detail:

- The function *selectFocus* takes a parameter *getFocus* the type of which also regulates the type of the focused entity. The monomorphic function *getFocus* is lifted to the generic level via *monoTU*. Applying the resulting generic function to a term, it will succeed (and return the input term) if the focus is placed on the given term. Otherwise, the application fails. In order to apply *getFocus* all over the tree until it succeeds, we simply employ the traversal scheme *oncetdS* from Figure 4.

- Replacement of the focused entity, as defined by the function *replaceFocus*, is very similar to selection, that is, we basically perform a top-down traversal with one intended application of a monomorphic function. This time, it is a type-preserving traversal. One might argue that the function for replacement of the focus is unnecessarily liberal in that it further descends into terms even if the focus was found but the replacement failed because of an applicability condition which did not hold. Indeed, one can specify a variant which does not descend any further once the focus has been located. We omit this optimization.

- The function *markHost* attempts to find a term which passes the *setHost* parameter, and which

is above the focused entity identified by the *getFocus* parameter. It marks then the found host so that subsequent transformations can observe a focus on the host. In this sense, the function is concerned with both the notion of scope (to determine a host) and focus (as for the focused entity and the marked host). By host we mean entities like abstractions. To identify the host of a focused term, we employ the scheme *aboveS* from Figure 4. Here we assume that the host of a focused term is the deepest term which meets the following two conditions. Firstly, it is a host-like term, that is, it can be transformed via *setHost*. Secondly, it contains the focused term.

Name analyses In addition to generic algorithms dealing with focus and scope, we also need further algorithms to analyse the names used in certain ways in program fragments. Essentially, refactorings need to be able to determine free and bound variables in a given scope. Here we make several assumptions. Firstly, names arise from all kinds of abstractions available in the language at hand. Secondly, the programming language is free to regulate name space issues, that is, the abstractions might live in one name space, or in separated name spaces. Thirdly, as for typed languages, abstractions can be associated with types which are either prescribed in the input program, or inferred by a corresponding algorithm. Fourthly, we basically distinguish two kinds of occurrences of names, namely declared or referring occurrences. Based on these assumptions, generic name analyses relevant for refactoring are specified in Figure 6. As an aside, the aforementioned assumptions are also taken into account in the interface for abstraction that will be presented shortly. Note the types of the functions for name analyses. These functions receive generic function parameters in order to generically identify names and possibly their types in given terms in a language-specific manner. Otherwise, these functions are ordinary polymorphic functions from terms to lists (say, sets) of names (maybe paired with types). Of course, the functions employ internally generic functions to perform the deep collections required for name analyses.

Let us explain the three functions in Figure 6 in detail:

- The function *freeNames* determines the set of free names in a given term. To this end, the function is parameterized by two type-unifying functions. The function *declared* is meant to identify declaration forms, and to return the corresponding declared names if any. In the same sense, *referenced* is expected to identify referenced names. The algorithm for free name analysis is based on a type-unifying bottom-up traversal of the following kind. The free names correspond to the union of the names referenced at the root node, and all the free names found for the subtrees (cf. $combTU\ union$ and $allTU$), except the names declared at the present node (cf. $combTU\ (\backslash\backslash)$).

- The function *boundTypedNames* accumulates all bound names and their types by descending into the given term until the focus is found. The accumulated name-type pairs are returned together with the focused term. In this manner, we determine what declarations are visible in the focused piece of code. It is interesting to notice that, in the context of refactoring, name analyses interact with the focus concept. The accumulation of bound names is based on the additional assumption that a declaring occurrence of a name will usually provide a type for the name.

- The function *freeTypedNames* is an elaboration of the function *freeNames* making use of accumulated name-type pairs *env*. In fact, a prime candidate for accumulation is the function *boundTypedNames*. That is, the function *freeTypedNames* qualifies the free names obtained by *freeNames* according to the name-type pairs received via the argument *env*. Here, we do not assume that a referring occurrence of a name necessarily exhibits a type for the relevant name. The types are rather obtained from the additional *env* parameter.

Generic analysis for free untyped names

```
freeNames :: (MonadPlus m, Eq name, Term t)
          ⇒ TU [(name, tpe)] m        -- Identify declarations
          → TU [name] m               -- Identify references
          → t                         -- Input term
          → m [name]                  -- Free names
freeNames declared referenced = applyTU freeNamesStrategy
  where
    freeNamesStrategy = combTU (\\)
      (combTU union (referenced `choiceTU` const []) (allTU freeNamesStrategy))
      ((declared `letTU` (λds → constTU (map fst ds))) `choiceTU` const [])
```

Generic analysis for bound typed names

```
boundTypedNames :: (MonadPlus m, Term f, Term t, Eq name)
                ⇒ TU [(name, tpe)] m        -- Identify declarations
                → (f → m f)                 -- Get focus
                → t                         -- Input term
                → m ([(name, tpe)], f)      -- Focus in context
boundTypedNames declared unwrap
  = applyTU (propagateTU [] updateEnv stopAtFocus)
  where
    stopAtFocus env = monoTU (λf → unwrap f ≫= λf′ → return (env, f′))
    updateEnv env = combTU (unionBy (λa → λa′ → fst a ≡ fst a′))
                           (declared `choiceTU` const [])
                           (constTU env)
```

Generic analysis for free typed names

```
freeTypedNames :: (MonadPlus m, Eq name, Term t)
               ⇒ TU [(name, tpe)] m        -- Identify declarations
               → TU [name] m               -- Identify references
               → [(name, tpe)]             -- Accumulated declarations
               → t                         -- Input term
               → m [(name, tpe)]           -- Free names with types
freeTypedNames declared referenced env t
  = freeNames declared referenced t ≫= λfrees →
    return (filter (λe → elem (fst e) frees) env)
```

Figure 6: Name analyses

3.2 Abstractions

In addition to the hot spots provided by the above generic algorithms, we also need an interface for language abstractions to detach ourselves from language-specific abstractions. Abstractions are so important because most refactorings deal with declaration forms and applications forms of abstractions. The interface is shown in Figure 7. In fact, the interface is defined as a highly-parameterized but otherwise completely systematic (if not trivial) Haskell class. The class members model observers and constructors. The class parameters are essentially place holders for syntactical domains. There is a parameter *abstr* for the domain of abstractions itself. There are parameters *name*, *formal*, and *body*

```
class (
        -- Abstractions and their components
    Term abstr,         -- Term type for abstraction
    Term [abstr],       -- Anticipation of lists of abstractions
    Term formal,        -- Formal parameters
    Term body,          -- Body of abstraction
        -- The corresponding applications
    Term apply,         -- Application
    Term actual,        -- Actual parameters
        -- Equality on names
    Eq name
  )
    ⇒ Abstraction abstr name tpe formal body apply actual

  |     -- Dependencies between syntactical domains
  abstr                → name,
  abstr                → tpe,
  abstr                → formal,
  abstr                → body,
  abstr                → apply,
  name formal body     → abstr,
  name actual          → apply,
  formal name tpe      → abstr,
  actual name tpe      → apply,
  apply                → name,
  apply                → actual,
  apply                → abstr,
  apply                → body,
  body                 → apply

where
    -- Observers
  getAbstrName    :: MonadPlus m ⇒ abstr → m name
  getAbstrType    :: MonadPlus m ⇒ abstr → m tpe
  getAbstrParas   :: MonadPlus m ⇒ abstr → m formal
  getAbstrBody    :: MonadPlus m ⇒ abstr → m body
    -- Constructors
  constrAbstr     :: MonadPlus m ⇒ name → formal → body → m abstr
  constrApply     :: MonadPlus m ⇒ name → actual → m apply
  constrBody      :: MonadPlus m ⇒ apply → m body
  constrFormal    :: MonadPlus m ⇒ [(name, tpe)] → m formal
  constrActual    :: MonadPlus m ⇒ [(name, tpe)] → m actual
```

Figure 7: A class of abstractions

for the constituents of abstractions. To be precise, the domain *name* does not just model names of the particular form of abstraction at hand but it corresponds potentially to a sum domain of all possible

forms of names for a language. Similarly, the parameter *tpe* is a place holder for all possible types (be it an attribute type, a method profile, or others). We assume that abstractions always admit the concept of application. Hence, there is a corresponding parameter *apply* and another parameter *actual* for the arguments of an application. The functional dependencies [9] state all the relations between the various syntactical and other domains. The members of *Abstraction* are intended to observe all the ingredients of both abstractions and applications. It also provides corresponding members for construction. Note that formal and actual parameter lists are constructed from lists of name-type pairs.

One can say that the definition of the class *Abstraction* corresponds to the Haskell-way of defining a signature morphism. All the class constraints on the parameters and the functional dependencies between the parameters effectively restrict possible instantiations. The use of the Haskell class mechanism provides us with two features. Firstly, when compared to explicit parameters, we can reduce the number of parameters in the various generic algorithms and refactorings since the abstraction interface is global. Secondly, note that we can easily deal with several forms of abstractions due to overloading.

3.3 Refactorings

The refactorings for extraction and introduction are defined in full detail. In fact, introduction, that is, insertion of a so-far unused abstraction, is also one of the major steps of extraction. It would be straightforward to present the dual refactorings, namely inlining and elimination. In the conclusion of the paper we comment on further refactorings.

Generic extraction The parameterized transformation function that models generic extraction is given in Figure 8. The first six parameters are framework parameters, that is, these parameters need to be fixed if a concrete, language-specific refactoring for extraction is derived. The first two parameters *declared* and *referenced* correspond to the ingredients of the name analyses. The parameter *find* specifies how to find the focused fragment which is subject to extraction. The two parameters *mark* and *find′* deal with marking and selecting a focus in lists of abstractions. This second kind of focus is relevant for the introduction step of extraction, that is, when the newly constructed abstraction is added to the appropriate list of abstractions. Finally, the parameter *check* anticipates that language-specific conditions need to be checked for the focused entity. Otherwise, the final two parameters *name* and *prog* just correspond to the desired name for the new abstraction, and the input program.

The actual specification of the *extract* refactoring is merely a list of small analysis, destruction, construction, and transformation steps. Let us just read all the 11 steps in Figure 8. First, we navigate to the focus while accumulating the bound names (cf. *boundTypedNames*). Then the language-specific requirements are tested for the focused entity (cf. *check*). Then, the abstraction is constructed in several steps corresponding to the smaller ingredients of the abstraction. In this course, the free names and their types are determined for the focused piece of code. The resulting name-type pairs serve as input for the construction of formal (and actual) parameter lists. The actual insertion of the constructed abstraction is defined via the separate refactoring *introduce* the application of which is preceded by a step to mark the relevant list of abstractions (cf. *markHost*). Afterwards, an application is constructed in two steps. Ultimately, the focused fragment is replaced by the application of the new abstraction (cf. *replaceFocus*).

Generic introduction In Figure 9, the generic refactoring *introduce* is specified. In order for an inserted abstraction not to interfere with the preexisting abstractions in a program (i.e., for the sake of semantics preservation), the name of the new abstraction should neither be bound nor free in the scope of the target list of abstractions. The parameters of *introduce* are the ingredients of the variable analyses, and the recognition function for the focused list of abstractions. These are the steps which

```
extract :: (MonadPlus m,
            Abstraction abstr name tpe formal body focus actual,
            Term prog
           )
         ⇒ TU [(name, tpe)] m                    -- Identify declarations
         → TU [name] m                           -- Identify references
         → (focus → m focus)                     -- Find focus
         → ([abstr] → [abstr])                   -- Mark host
         → ([abstr] → m [abstr])                 -- Find host
         → ([(name, tpe)] → focus → m ())        -- Check focus
         → name                                  -- Name for abstraction
         → prog                                  -- Input program
         → m prog                                -- Output program

extract declared referenced find mark find′ check name prog
  = do
        -- Operate on focus
      (env, focus)   ← boundTypedNames declared find prog
      ()             ← check env focus
        -- Construct abstraction
      frees          ← freeTypedNames declared referenced env focus
      formal         ← constrFormal frees
      body           ← constrBody focus
      abstr          ← constrAbstr name formal body
        -- Insert abstraction
      prog′          ← markHost find mark prog
      prog′′         ← introduce declared referenced find′ abstr prog′
        -- Construct application
      actual         ← constrActual frees
      apply          ← constrApply name actual
        -- Replace focus by application
      replaceFocus (λf → find f ≫= (const (return apply))) prog′′
```

Figure 8: Definition of generic extraction

are performed by a generic introduction. Firstly, the relevant list of abstractions is selected from the focus. Secondly, the name of the abstraction subject to insertion is determined. Thirdly, the free names *frees* in the relevant list of abstractions are determined. Then, also the names *defs* of all the abstractions in the local list are collected. Afterwards, it is tested that the name of the new abstraction is neither contained in *frees* nor *defs*. Ultimately, the list of abstractions is extended with the new abstraction (cf. *replaceFocus*).

It is important to notice that a generic refactoring is not concerned with all the details of the static and dynamic semantics of the involved syntactical fragments. The result of an introduction refactoring, for example, is not necessarily well-typed because the abstraction might fail to fit into the focused location (as for types of free names). The only purpose of checks in refactoring are to ensure that semantics preservation holds. This is the reason that we check that the introduced abstraction does not override some other visible abstraction. Simple checks for semantics preservation and static

```
introduce :: (MonadPlus m,
              Abstraction abstr name tpe formal body apply actual,
              Term prog
             )
          ⇒ TU [(name, tpe)] m              -- Identify declarations
          → TU [name] m                     -- Identify references
          → ([abstr] → m [abstr])           -- Find scope with abstractions
          → abstr                           -- Abstraction to be inserted
          → prog                            -- Input program
          → m prog                          -- Output program

introduce declared referenced find abstr
  = replaceFocus
      (λabstrlist →
          do
             abstrlist' ← find abstrlist
             name       ← getAbstrName abstr
             frees      ← freeNames declared referenced abstrlist'
             defs       ← mapM getAbstrName abstrlist'
             guard (and [¬ (elem name frees), ¬ (elem name defs)])
             return (abstr : abstrlist'))
```

Figure 9: Definition of generic introduction

semantics checking are two separate concerns. Of course, we should usually perform a subsequent check to ensure that the result of refactoring is correct regarding the static semantics. Here, we assume that this kind of executable language semantics is available. There are also refactorings which are completely self-checking not just for semantics-preservation but even for static correctness. A good example is extraction. If the instantiation of generic extraction is properly performed, the result of a language-specific extraction will always be statically correct.

4. Instantiation for JOOS

We have instantiated the framework for several languages, among them a Haskell subset, definite clause programs, XML schemata, syntax definitions, Pascal, and the Java subset JOOS.[1] In the sequel, we will discuss the JOOS instance in some detail. As an aside, in [17], we describe an extract method refactoring for Java (say, JOOS) in the Strafunski style but in a Java-specific manner, that is, without an attempt to employ a generic and reusable specification of extraction. It is fair to say that the non-framework approach is much less concise when compared to the framework approach from the present paper. The JOOS instance parallels the framework. Firstly, we refine the generic algorithms of the framework for JOOS. Secondly, we provide an instance of the abstraction interface of the framework for JOOS method declarations. Ultimately, the framework refactorings are specialised to JOOS refactorings dealing with method declarations.

[1] JOOS was originally designed by Laurie Hendren. The language has been used in various courses and research projects in the last few years in various locations.

Syntax extensions

data *Statement* = ⋯ | *StatementFocus Statement*
data *MethodDeclaration* = ⋯ | *MethodDeclarationFocus MethodDeclaration*

Focus on statements

wrapStatement = *StatementFocus*
unwrapStatement (*StatementFocus x*) = *return x*
unwrapStatement _ = *mzero*

Focus on a list of method declarations

wrapMethodDeclarations xs = [*MethodDeclarationFocus xs*]
unwrapMethodDeclarations [*MethodDeclarationFocus xs*] = *return xs*
unwrapMethodDeclarations _ = *mzero*

Figure 10: Kinds of focus for JOOS

4.1 Algorithm refinement

Firstly, we define the kinds of focus relevant for the planned JOOS refactorings. Secondly, we refine the name analyses for JOOS.

Focus We need two kinds of focus for the upcoming JOOS refactorings. Firstly, the focus for extraction of JOOS method declarations is concerned with statements. Secondly, the focus for insertion of JOOS method declarations is of the type of lists of method declarations. These kinds of focus are specified in Figure 10. We assume that the syntactical domains *Statement* and *MethodDeclaration* admit corresponding constructors *StatementFocus* and *MethodDeclarationFocus*. The functions for wrapping and unwrapping a focus term constructor are then trivially defined. These function will be useful as parameters of the generic algorithms and the refactorings (cf. *getFocus*, *setHost*, etc.).

Name analysis In Figure 11, the domains of JOOS names and types are identified. In fact, we restrict ourselves to forms of names and types which are relevant for the upcoming refactorings. Furthermore, type-unifying functions for the identification of certain kinds of names are identified. In JOOS, we have that variables, methods, and method parameters all live in the same name space. Hence, the type *NameJoos* for JOOS names coincides with the syntactical domain *Identifier* of the JOOS language (as opposed to a disjoint union of some types of names). As for the types relevant for the upcoming refactoring, we separate expression types and method types. This leads to the two alternatives in the definition of *TypeJoos*.

The framework does only separate declared and referenced variables. By contrast, in a language like JOOS where one can have side effects, we should also separate defined (or assigned) references and using references. We will later see that this distinction is actually mandatory for the correct instantiation of the *extract* refactoring. Hence, we have three generic functions for name identification. The function *declaredJoos* identifies names together with their types. As one can see from the patterns covered by the function, we care about variable declarations and method parameters. The function *definedJoos* identifies left-hand side references in JOOS assignments. The function *usedJoos* identifies identifiers in expressions. Again, the patterns were selected based on a simple analysis which JOOS usage patterns of names would be relevant for the upcoming refactorings. Finally, we take the "union" of *definedJoos* and *usedJoos* via *choiceTU* to also be able just to identify references of any kind (cf. *referencedJoos*).

JOOS names and types

```
type NameJoos = Identifier
data TypeJoos = ExprType Type
              | MethodType (Maybe Type) FormalParameters
```

Declared names (with type)

```
declaredJoos :: MonadPlus m ⇒ TU [(NameJoos, TypeJoos)] m
declaredJoos = adhocTU (adhocTU failTU
                         declaredBlock)
                         declaredMeth
  where
    declaredBlock (BlockStatements vds _)
      = return (map (λ(VariableDecl t i) → (i, ExprType t)) vds)
    declaredMeth (MethodDecl _ _ (FormalParams fps) _)
      = return (map (λ(FormalParam t i) → (i, ExprType t)) fps)
```

Defined names (without type)

```
definedJoos :: MonadPlus m ⇒ TU [NameJoos] m
definedJoos = adhocTU failTU definedAssignment
  where
   definedAssignment (Assignment i _) = return [i]
```

Used names (without type)

```
usedJoos :: MonadPlus m ⇒ TU [NameJoos] m
usedJoos = adhocTU (adhocTU failTU
                     usedExpression)
                     usedInvocation
  where
    usedExpression (Identifier i) = return [i]
    usedExpression _ = mzero
    usedInvocation (ExpressionInvocation _ i _) = return [i]
    usedInvocation (SuperInvocation i _) = return [i]
```

Referenced names (without type)

```
referencedJoos :: MonadPlus m ⇒ TU [NameJoos] m
referencedJoos = definedJoos `choiceTU` usedJoos
```

Figure 11: Ingredients for name analyses for JOOS

4.2 Method declarations

In the present paper, we restrict ourselves to refactoring for JOOS method declarations. The JOOS language also offers other forms of abstractions. In particular, JOOS class declarations would be involved in many interesting refactorings. In Figure 12, the framework class *Abstraction* is instantiated for JOOS method declarations. The actual specification is straightforward. Observers are more or less encoded by pattern matching to return the corresponding fragments of a JOOS method declaration; dually for the constructors. Note how the abstraction interface and the model for the generic algorithms for name analyses interact. Instead of plain method identifiers and types, we use the domains

```
instance Abstraction
                MethodDeclaration        -- abstr
                NameJoos                 -- name
                TypeJoos                 -- tpe
                FormalParameters         -- formal
                BlockStatements          -- body
                Statement                -- apply
                Arguments                -- actual
  where
    getAbstrName (MethodDecl _ i _ _) = return i
    getAbstrType (MethodDecl m _ f _) = return (MethodType m f)
    getAbstrParas (MethodDecl _ _ ps _) = return ps
    getAbstrBody (MethodDecl _ _ _ b) = return b
    constrAbstr n f b = return (MethodDecl Nothing n f b)
    constrApply n a = return (MethodInvocationStat
                                  (ExpressionInvocation This n a))
    constrFormal vars = mapM f vars >>= (return ∘ FormalParams)
      where f (i, tpe) = case tpe of
                           ExprType t → return (FormalParam t i)
                           _ → mzero
    constrActual vars = mapM f vars >>= (return ∘ Arguments)
      where f (i, tpe) = case tpe of
                           ExprType _ → return (Identifier i)
                           _ → mzero
    constrBody s = return (BlockStatements [] [s])
```

Figure 12: JOOS method declarations

NameJoos and *TypeJoos* as parameters for the *Abstraction* instance.

As an aside, in general, observers and constructors can be partial functions (hence, the *MonadPlus* constraints in Figure 7). This is useful if we want to enforce certain side conditions on the relevant syntactical fragments. These side conditions can deal with normal-form issues or with other restrictions of the framework. To give an example, consider forms of abstractions defined by multiple equations or clauses (e.g., predicates in logic programming, or functions in functional programming). If we want to determine the body of such an abstraction, then this is only feasible (without prior normalization) if there is precisely one equation or clause. In fact, one can think of a refactoring to prepare abstractions accordingly, e.g., to turn a function defined by pattern matching into a function defined in terms of a case expression. In Figure 12, we use partiality in a trivial manner, namely we require that the list of name-type pairs only deals with expression types.

4.3 Refactorings refinement

In Figure 13, the refactorings for extraction and introduction of JOOS method declarations are derived from the generic ones by straight parameter passing. This is the point where the framework approach pays off. The hot spots get closed. All the needed ingredients for the name analyses, and for focus processing were defined before. As for extraction, we need to define the JOOS-specific requirements for a valid extraction of a statement. Two conditions need to hold (cf. the auxiliary function *check*).

Type of transformations on JOOS programs

```
type TrafoJoos m = Program → m Program
```

Extraction of a statement to constitute a new method declaration

```
extractJoosMethod :: MonadPlus m ⇒ NameJoos → TrafoJoos m
extractJoosMethod
  = extract
      declaredJoos
      referencedJoos
      unwrapStatement
      wrapMethodDeclarations
      unwrapMethodDeclarations
      check
    where
        -- Ensure absence of returns and non-local assigns
      check env f = guard noReturn>>=
                    const (freeNames declaredJoos definedJoos f)>>=
                    guard ∘ (≡) [ ]
        -- Test for absence of returns in focused fragment
      noReturn = case (applyTU (oncetdTU (monoTU (λs → case s of
                                 ReturnStat _ → Just ()
                                 _ → Nothing)))) of
                   Nothing → True;
                   Just () → False
```

Introduction of a method declaration

```
introduceJoosMethod :: MonadPlus m ⇒ MethodDeclaration → TrafoJoos m
introduceJoosMethod
  = introduce
      declaredJoos
      referencedJoos
      unwrapMethodDeclarations
```

Figure 13: JOOS extraction and introduction by specialisation

There are no return statements contained in the focused fragment (cf. *noReturn*). There are no free variables defined in the focused fragment (cf. *freeNames*).

5. Concluding remarks

Contribution We have shown how program transformations for refactoring can be represented in a largely language-independent manner using generic functional programming as a sufficiently expressive and concise specification medium. From the examples given, it is clear that several refactorings for different forms of abstractions are accessible for such a generic approach. Among these refactorings there are extraction, introduction, inlining, and elimination. The ability to specify program transformations at this level of abstraction allows us to capture commonalities of different programming

languages in a way which provides new insight into language design and language semantics. One is used to the idea that frameworks for static and dynamic semantics are meant to cover common building blocks of languages [18, 20]. In the context of executable language definition or language implementation [5, 10], the idea of reusable components is also quite common. The contribution of the present paper is that we instantiated the idea of common building blocks for program transformation on the basis of program refactorings. We do not argue that a generic refactoring framework like the one we have proposed is particularly strong because it would enable reuse of program transformations. This would be like saying that modular semantics has significantly simplified compiler implementation. It is more important that one is able to talk about commonalities in mathematical and transformational semantics to witness the structure underlying different programming languages.

Related work The idea of operator suites for refactoring is, of course, not new. In his seminal thesis [21] and accompanying conference papers, Opdyke develops a set of operators for refactoring object-oriented frameworks. His results are somewhat independent of the actual object-oriented programming language. Based on such operator suites, corresponding refactoring tools have been designed [19, 26]. As for object-oriented programming, tool-supported refactoring is well established. What is new in our work is that we collect refactorings in a truly language-independent, declarative, prescriptive and executable framework.

Research on program transformation usually aims at some degree of language independence. In [24], for example, rules and strategies for transforming both logic and functional programs are examined in depth making only few assumptions about the covered languages. Our transformation operators collected in the framework are original in that the technicalities of refactoring such as focus, name analyses, construction and destruction, or scope are all treated in a generic manner.

An important initial contribution to the idea of generic transformations originated from the Stratego project [28] where traversal schemes for analysis and transformation have been identified as reusable building blocks of program transformations. In [27], basic traversal schemes but also algorithms for variable analysis, unification, and substitution were specified in Stratego—a language with prime support for term traversal, but without strong typing, and support for general higher-order functions. Our work clearly illustrates that higher-orderness and types are desirable if not indispensable for transformation frameworks. Higher-orderness is implied by the nature of the involved parameters, and by the employment of higher-order functional programming techniques for a reasonably concise style. Types are not just convenient for documentation purposes, but they are actually instrumented to guide traversals (recall generic function update via $adhocTP$ and $adhocTU$). Types are also essential to constrain valid instantiations of the framework. Without strong type checking, very generic, highly-parameterized frameworks are easily configured in an inconsistent manner.

Our framework provides a model for manual, local transformations (i.e., refactorings) aiming at some kind of improvement of the refactored program in structural terms. Refactoring is different from other forms of transformational programming where one is rather interested in the *calculation* of a usually efficient program from a specification [22, 2].

Perspective Besides extraction, introduction, inlining, and elimination, further generic refactorings are conceivable, e.g., refactorings for lifting and dropping abstractions, say to move around abstractions in nested levels of abstraction [3, 4]. Language-specific refactoring catalogs as in [21, 7] should also be investigated to systematically extract all refactorings which make sense at an language-independent, abstract level. One might also want to go beyond refactorings in the sense that more powerful adaptations are enabled, e.g., the adaptations from [25, 12] to add computational behaviour. Furthermore, the specification of compound (generic) transformation schemes (say, strategies) in the sense of [24] is a subject for future work. Moreover, the integration of generic refactorings and truly language-specific refactorings deserves some effort, e.g., the very object-oriented refactorings in [7], or the specifically functional refactorings in [13], e.g., monad introduction.

References

1. F. Bellegarde. Notes for Pipelines of Transformations for ML. Technical Report 95-021, Oregon Graduate Institute, 1995.
2. R. S. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.
3. O. Danvy and U. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 90–106, New York, June 12–13 1997. ACM Press.
4. O. Danvy and U. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1–2):243–287, Oct. 2000.
5. R. Farrow, T. Marlowe, and D. Yellin. Composable Attribute Grammars. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages (Albuquerque, NM)*, pages 223–234, Jan. 1992.
6. J. Favre. Preprocessors from an abstract point of view. In *Proceedings of the International Conference on Software Maintenance*, pages 329–339, Washington, Nov. 4–8 1996. IEEE Computer Society Press.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
8. *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. `http://www.haskell.org/onlinereport/`.
9. M. P. Jones. Type Classes with Functional Dependencies. In G. Smolka, editor, *Proceedings of ESOP'00*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlag, Mar. 2001.
10. U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.
11. B. Kuhlbach and V. Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings Eighth Working Conference on Reverse Engineering*, pages 3–12, 2001.
12. R. Lämmel. Declarative aspect-oriented programming. In O. Danvy, editor, *Proc. 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, Jan. 1999.

13. R. Lämmel. Reuse by Program Transformation. In G. Michaelson and P. Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.
14. R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*. Springer-Verlag, 2001.
15. R. Lämmel. First-class Polymorphism With Type Case and Folds over Constructor Applications, Jan. 2002. Draft paper; Submitted; Available at `http://www.cwi.nl/~ralf/rank2/`.
16. R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*. OCG, 2001.
17. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.
18. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
19. I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250. ACM Press, 1996.
20. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
21. W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
22. H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
23. P. Pepper. LR Parsing = Grammar Transformation + LL Parsing. Technical Report CS-99-05, TU Berlin, Apr. 1999.
24. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.
25. A. Power and L. Sterling. A Notion of Map Between Logic Programs. In D. Warren and P. Szeredi, editors, *Proceedings 7th International Conference on Logic Programming (ICLP)*, pages 390–404. The MIT Press, 1990.
26. D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
27. E. Visser. Language Independent Traversals for Program Transformation. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 86–104, July 2000.
28. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proc. of ICFP'98*, pages 13–26, Sept. 1998.
29. P. Wadler. The essence of functional programming. In *Conference record of POPL'92*, pages 1–14. ACM Press, 1992.

Table of Contents