Constraint-Based Refactoring with Foresight

Friedrich Steimann and Jens von Pilgrim

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org, Jens.vonPilgrim@feu.de

Abstract. Constraint-based refactoring tools as currently implemented generate their required constraint sets from the programs to be refactored, before any changes are performed. Constraint generation is thus unable to see — and regard — the changed structure of the refactored program, although this new structure may give rise to new constraints that need to be satisfied for the program to maintain its original behaviour. To address this problem, we present a framework allowing the constraint-generation process to foresee all changes a refactoring might perform, generating — at the outset of the refactoring — all constraints necessary to constrain these changes. As we are able to demonstrate, the computational overhead imposed by our framework, although threatening viability in theory, can be reduced to tractable sizes.

1 Introduction

Refactoring is the discipline of changing a program in such a way that one or more of its non-functional properties (readability, maintainability, etc.) are improved, while its behaviour is maintained [4]. When applied to real programs written in real programming languages, refactoring involves complex precondition checking and mechanics that contain deeply nested case analyses, making refactoring without tool support tedious and error-prone. A steadily growing number of fully automated refactoring tools is therefore being devised; of these, a considerable part is constraint based (e.g., [1, 3, 5, 9, 10, 16–20]).

Current approaches to constraint-based refactoring use so-called *constraint rules* to generate sets of constraints from the programs to be refactored. The generated constraints rule over how the program may be changed without affecting its well-formedness or behaviour. Constraints are said to be generated in a syntax-directed manner [1, 5, 10, 20], i.e., based on the program's abstract syntax tree (AST), which represents the program as is before the refactoring.

One problem of this approach is that constraint rule application is unable to see the structural changes of the AST imposed by a refactoring. For instance, if a refactoring moves a program element to another location (corresponding to another node in the AST), the constraints imposed on the element by this new location have not been generated, since at the time of rule application, the element was still in its old location. Note that taking the *refactoring intent* (here to move the element to a known location) into account does not generally suffice to fix the problem, since the intended

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 535–559, 2012.

refactoring may require other elements to change as well (here: to move to the same location), which precisely only being known *after* having solved the constraints describing the refactoring problem. Thus, constraint-based refactorings need some kind of recognition of how a program is going to change.

In this paper, we present a solution to a class of problems that, following our first mention of it in [17], we have dubbed the *foresight problem* of constraint-based refactoring. Our solution relies on constraint rule rewriting and quantified constraints extending the scope and expressiveness of constraint rules so that they can cover all possible changes of a program's structure that might infringe its well-formedness or affect its behaviour. To address the computational complexity introduced by this, we present an algorithm for cancelling constraints not needed for a concrete refactoring. That our approach is indeed viable is demonstrated by applying it to five different variants of the PULL UP FIELD refactoring.

The remainder of this paper is organized as follows. In Section 2, we motivate our work by presenting various examples of the foresight problem, and reflect on the related literature. In Section 3 we provide a quick introduction to constraint-based refactoring, mostly for readers not acquainted with this refactoring technique. Sections 4 and 5 introduce our notions of quantified constraints and constraint rule rewriting that will be exploited in Section 6 for addressing the foresight problem (including the complexity imposed by our solution). After a brief sketch of the implementation in Section 7, our evaluation in Section 8 shows that the added complexity can be reduced significantly for many practical cases.

2 Motivation

To give the reader an impression of how constraint-based refactoring works, we first take a look at the following simple

EXAMPLE 1: Consider the Java program

```
class A {}
class B extends A {
  int i = 1;
  int j = this.i;
}
```

and the intended refactoring "pull up field `j` from class `B` to class `A`". A quick analysis shows that this is not possible if the pulling up of `j` is the only change the refactoring is allowed to make; if it may perform additional changes, it may be made to work by pulling up field `i` as well, or by separating the declaration of `j` from its initialization. An indeed, Eclipse (whose implementation of the PULL UP FIELD refactoring is constraint-based [19]) warns the user of the fact that field `i` will be undefined in the new location of `j`. ♦

In constraint-based refactoring, what a refactoring tool must do in order to perform the refactoring correctly is described as a *constraint satisfaction problem* (CSP) generated from the program as is, and the refactoring intent. The two constraints sufficiently describing the refactoring problem of the above example are that

1. the declaring type of `j`, the location (i.e., the hosting type) of `this`, and the location of the reference to `i` must be equal (since they are part of the same statement); and

2. the location of this must be a (non-strict) subtype of the declaring type of i (so that i is defined, either directly or via inheritance, for the object represented by this).

Note that both constraints are satisfied by the program as is:

1. the declaring type of j, the location of this, and the location of the reference to i are all B and
2. the location of this, B, is a (non-strict) subtype of the declaring type of i, also B.

However, the constraints are not satisfied after the declaration of j has been pulled up to A, since then the declaring type of j becomes A which, by satisfaction of Constraint 1, implies that the location of this is also A, which violates Constraint 2, since A is not a subtype of the declaring type of i, B. Both constraints can be satisfied, however, by changing the declaring type of i to A also, which is equivalent to pulling up i as well.

In current implementations of constraint-based refactoring, constraints like the above are generated from a program to be refactored by application of so-called *constraint rules*, whose precedents are matched against the AST representation of the program as is before the refactoring. As the following examples will demonstrate, this approach (which worked fine for Example 1) is challenged by refactorings that change the structure of the AST, by changing the locations of program elements.

2.1 Examples of the Foresight Problem

We begin our exploration of the foresight problem with the following simple

EXAMPLE 2: In the sample program

```
package p;
public class A {}
package q;
class B extends p.A { protected int i; }
class C extends B { void m(B b) { i = b.i; } }
```

pulling up i from B to A seems possible at first glance (since generally, *protected* accessibility suffices for inheritance across different packages), but will cause a compile error on b.i in C.m(B), since a rule of the Java language specification (JLS; [6], §6.6.2) mandates that

if a member (here: A.i) of an object (b) is accessed (b.i) from outside (q) the package in which it is declared (p) by code (C.m(B)) that is not responsible for the implementation of that object,
then accessibility must be *public*.

However, without special measures this rule (which corresponds to the rule Acc-2 of [17]) fails to generate the constraint required for adjusting i's accessibility to *public*. ♦

The problem exposed by Example 2 is that with the program as is, one conjunct of the rule precedent, that the access occurs from outside the package, is not fulfilled (since the declaration of i and its access in C.m are in the same package when the rule is applied), so that no constraint requiring *public* accessibility will be generated by applying this rule to the program. That *public* accessibility is required for i is known only after the fact, namely after it has been pulled up. Generation of the corresponding constraint thus requires foresight of the move.

Of course, one could argue that required accessibility always depends on the location of the accessor and the accessed, and that any constraint constraining accessibility should take the variability of the locations of the two into account. This is somewhat different for the following

EXAMPLE 3: In the sample program

```
class A {}
class B extends A { static int i = 0; }
interface I { int i = 1; }
class C extends A implements I {
  static int j = i;
}
```

pulling up the field `i` from class `B` to class `A` makes the access of `i` from class `C` ambiguous, since it is unclear whether `I.i` or `A.i` is referenced. However, Eclipse's constraint-based implementation does not foresee the problem and performs the refactoring without warning. ♦

One possible remedy for the problem of Example 3 is to add a constraint requiring an accessibility of `A.i` that makes it inaccessible from `C`, but then, this constraint makes no sense for the program before the refactoring — there is no `A.i` and why should accessibility of `B.i` be lowered? Again, a solution to this problem requires foresight of the situation after the refactoring.

Example 3 differs from Example 2 in that for the program as is, the field `B.i` is completely unrelated to the rest of the program, so that there seems to be no reason at all to generate a constraint for it (in fact, `B.i` could even be deleted without changing the meaning of the program). This was different for Example 2, in which `B.i` was already considered in the constraint generation process, only not in a manner that was still sufficient after the refactoring. However, both examples have in common that with the refactoring intent known to the constraint generator, it could be tweaked to insert the necessary constraints. This will be different for

EXAMPLE 4: Extending Example 1 to

```
class Z { int i = 0; }
class A extends Z { { assert this.i == 0; } }
class B extends A {
  int i = 1;
  int j = this.i;
}
```

the pulling up of `B.j` to `A` should be rejected since the necessary accompanying pulling up of `B.i` to `A` changes the binding of references to `i` on instances of type `A`, unless all receivers of references to `Z.i` of (static) type `A` (here: `this` in the assert statement) are cast to `Z`, or one of `Z.i` and `B.i` is renamed. Again, the current constraint-based implementation of PULL UP FIELD in Eclipse fails to see this. ♦

The problem highlighted by this example is that, given that the refactoring intent is to pull up `B.j`, it is difficult to foresee, for a constraint generator applied to the original program and refactoring intent, that access to `Z.i` should involve a cast, or a field should be renamed, since it is unknown, before the generated CSP has been solved, that `B.i` must be pulled up as well. Without foresight of that such a change might happen, no constraints protecting bindings to `Z.i` will be generated. Generally, refactorings can have far-reaching ripple effects that are difficult to foresee, and a correct implementation must account for them all. This is a non-trivial problem.

2.2 Related Work

In his doctoral dissertation, Griswold used bidirectional mappings between Scheme programs and program dependence graphs that allowed him to perform restructurings on the latter (and thus on a representation in which behaviour preservation is relatively easy to assert) [8]. This approach is somewhat analogous to that taken by constraint-based refactoring, which transforms a program to a CSP, in which refactoring amounts to constraint solving (see Section 3 for details).

Constraint-based type refactorings as pioneered by Tip et al. [19, 20] make a number of simplifying assumptions avoiding most problems that we are addressing here. In particular, they assume that program elements whose moving may accidentally change behaviour-critical dependencies (such as binding or overriding relationships) are adequately renamed before the move [20]. However, even if programs are prepared in such manner, the constraint rules provided in [20] still fail to address some of the foresight problems we are solving: for instance, rule 12 (for hiding) makes sure that existing hiding relationships are preserved (preventing a renaming), but cannot avoid a change of binding due to the pulling up of a field, as in our Example 4.

The work of Schäfer et al. [13–15] avoids accidental changes of behaviour-critical program dependencies by recording them prior to refactoring and by introducing a correction phase that restores the original dependencies, if possible, after the intended refactoring has been performed (solving the problem with hindsight, so to say). However, the changes necessary to perform the correction may themselves affect well-formedness and meaning of a program, which is why Schäfer's approach of locked dependencies has recently been combined with constraints [16]. As we will show, our work presented here is more general not only in that it is capable of addressing the primary refactoring and all corrections required using a single formalism (avoiding the looping between dependency locking/unlocking and constraint solving), but also in that it can choose between different measures for maintaining the original dependencies: for instance, it may rename problematic program elements, or make them inaccessible, or move them to locations in which they do not interfere.

Dynamically changing systems gave rise to the investigation of so-called *dynamic constraint satisfaction problems* [11], in which the activeness of certain constraints depends on the satisfaction of others. More specifically, the introduction of conditional constraints allows the constraint solver to explore dynamic reconfigurations of (usually hardware) systems certain components of which may or may not be present (i.e., switched on or off). This situation is not unlike the foresight problem of refactoring, which must also let dynamically changing configurations (that is, changes to the program structure) be explored by the constraint solver. However, restructuring software must deal with the more general problem of moving program elements around, and placing a switch at every possible location would be absurdly expensive (especially since, as has been shown in [7], the computational burden of conditional constraints is heavy).

In object-oriented programming, conditional constraints have been used for type inference [12] and also for certain constraint-based refactorings [1, 3]. In particular, [3] has used conditional constraints (there called *guarded constraints*) to handle the interplay of parameterized and raw types when converting Java programs to use generic libraries: whereas parameterized types require additional constraints on variables

representing the type parameters, raw types do not give rise to such variables which, consequently, cannot be constrained. Constraints generated from assignments must therefore be made sensitive to the "parameterizedness" of the participants and, since the parameterizedness may be changed by the refactoring, this sensitivity must be dynamic. Similarly, [1] has used conditional constraints (there called *implication constraints*) to let generated type constraints depend on a binary switch indicating whether an occurrence of a constructor or method call of a legacy type has been replaced (by the constraint solver) with an equivalent call of a migration type. However, both problems are analogous to the hardware problem of switching on or off components, whereas we have to deal with moving program elements to new locations.

In earlier work of ours on constraining accessibility under refactoring, we introduced so-called foresight application of constraint rules which, knowing to which location a program element was to be moved by a refactoring (the refactoring intent), computed the required accessibilities for that location [17]. However, since this computation occurred outside the constraint solving process, we had to know in advance which program elements were to be moved where, which, as argued above, is an unrealistic assumption in the general case.

3 A Brief Recap of Constraint-Based Refactoring

In constraint-based refactoring, a program is sufficiently represented by

- a set of variables, called *constraint variables*, representing selected properties of the program and
- a set of relationships, called *constraints*, constraining the properties, representing syntactic and semantic rules of the programming language as applied to the program.

Together, the constraint variables and the constraints define a CSP whose solution space represents programs that are refactorings of each other. For instance, the CSP corresponding to the refactoring problem of Example 1 consists of the constraint set

$$\{v_1 = v_2 = v_3,\ v_2 \leq v_4\} \tag{1}$$

where v_1 represents the declaring type of `j`, v_2 represents the location of `this`, v_3 represent the location of the reference to `i`, and v_4 represents the declaring type of `i` (all having the initial value `B`; note that (1) is solved with these values). Pulling up `j` translates to assigning v_1 the new value `A`, which (via the equality constraint $v_1 = v_2 = v_3$) is propagated to v_2, which in turn (via the inequality constraint $v_2 \leq v_4$) requires v_4 to change to `A` as well, translating to pulling up `i` along with `j`.

Generally, the CSP representing a refactoring problem is solved with the initial values of the constraint variables assigned. A refactoring intent (such as pulling up field `j` from `B` to `A`) translates to changing one or more variable values, which may require other variable values to change as well for the CSP to remain solved, which ones precisely being computed by a constraint solver. Each solution of the CSP then corresponds to a refactored program that is obtained by writing back the values of the changed variables to the original program.

Table 1. Properties of program elements used in more than one occasion throughout this paper

PROPERTY	MEANING
$e.\alpha$	the *declared accessibility* of e (corresponding to $\langle e \rangle$ in [17])
$e.\lambda$	the *location* of e, the type in whose body e occurs (corresp. to $\lambda(e)$ in [17] and, for declared entities, to *Decl*(e), the *declaring type* of e, in [19])
$e.\lambda_T$	the *top level type* hosting e; same as $e.\lambda$ for elements directly occurring in the bodies of top level types
$e.\iota$	the *identifier* of e
$e.\pi$	the *package* hosting e
$e.\tau$	the *type* of e (declared or inferred; corresponding to $[e]$ in [19])

3.1 Constraint Rules

A CSP such as (1) that represents a program to be refactored is generated from this program by application of so-called *constraint rules*, which are generally of the form

$$\frac{\textit{program queries}}{\textit{constraints}}$$

Here, *program queries* is a set of predicates (implicitly conjoined) that are interpreted as queries over a program, and *constraints* represents the set of constraints to be generated (added to the CSP) for program elements selected by the queries. Both the program queries and constraints contain variables which are bound to *program elements* (*declared entities* and *references* to declared entities of a program; the nodes of its AST) by the queries; the constraint rule is implicitly universally quantified over these variables (note that these variables are *not* the constraint variables).

EXAMPLE. Application of the constraint rule

$$\frac{\textit{overrides}(M_2, M_1)}{\textit{accessible}(M_2, M_1)}$$

to a program searches the program for occurrences of pairs of methods (M_2, M_1) such that M_2 overrides M_1, and generates for each found pair a constraint requiring that M_1 is accessible from M_2 ([6], §8.4.8.1). ♦

Constraints such as the above *accessible*(M_2, M_1) generated by the application of constraint rules do not constrain program elements directly — rather, they constrain *properties of the program elements* (which are therefore the constraint variables of the CSP). The properties of program elements and their domains depend on the elements' kinds (i.e., whether an element is a declared entity or a reference, whether a declared entity is a method, a field, etc.): for instance, the declaration of a field has at least the properties *location* (where the field is declared; the hosting type), *type* (the declared type of the field), and *accessibility* (the access modifier used in the declaration, in Java one of *private*, *package*, *protected*, or *public*). We use Greek letters to denote

properties: $e.\tau$ for the type of element e, $e.\lambda$ for the location of e, $e.\alpha$ for the declared accessibility, etc.; Table 1 summarizes the properties that we will be using repeatedly throughout this paper.

EXAMPLE. A spelled out and extended variant of the previous constraint rule is

$$\frac{\mathit{overrides}(M_2, M_1)}{M_2.\lambda \leq_\tau M_1.\lambda \quad M_1.\alpha \geq_\alpha \alpha(M_2, M_1)}$$

in which $\leq_\tau$ represents the subtype relationship defined by the program, $>_\alpha$ represents the (total) ordering of access modifiers in Java (with $\geq_\alpha$ being defined as usual), and α is a helper function computing the minimum required accessibility for the declared entity of the second argument when accessed from the location of the first ([17]; see Figure 2 for how α is defined in terms of constraints). Taken alone, these constraints allow it that M_1 or M_2 are moved up or down the class hierarchy as long as $M_2.\lambda$ remains a subtype of $M_1.\lambda$, and that the declared accessibility $M_1.\alpha$ may be increased or lowered, as long as it remains above what is required by the locations of M_1 and M_2 relative to each other. ♦

The constraint rules governing the PULL UP FIELD refactoring of Example 1 are

$$\frac{\mathit{same\text{-}statement}(e_1, \ldots, e_n)}{e_1.\lambda = \ldots = e_n.\lambda}$$

where the variable argument query *same-statement*(e_1, …, e_n) finds all tuples of program elements occurring in the same statement, and

$$\frac{\mathit{binds}(f, F) \quad \mathit{receiver}(f, r)}{r.\tau \leq_\tau F.\lambda}$$

where *binds* finds all pairs of field accesses f and field declarations F such that f binds to F, and *receiver* finds all pairs of field accesses f and references r such that r is the receiver of f.[1] Applied to the program of Example 1, these two rules generate the constraint set

$$\{\ \mathsf{j}^{\mathsf{B}}.\lambda = \mathsf{this}_{\mathsf{B}}.\lambda = \mathsf{i}_{\mathsf{B}}.\lambda,\ \mathsf{this}_{\mathsf{B}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda\ \}$$

(with j^{B} representing `B.j`, $\mathsf{this}_{\mathsf{B}}$ representing the reference to `this` in `B`, etc.).

It is instructive to note that to a certain extent, program queries and constraints can be exchanged for each other. For instance, the expression $e_1.\lambda = e_2.\lambda$ can be interpreted as a query, in which case it means "select all pairs of program elements (e_1, e_2) such that e_1 and e_2 are located in the same type", or interpreted as a constraint, meaning "whatever the location of e_1 or e_2, it must be the same as the other". The main differences are operational: whereas the query finds all instances of e_1 and e_2 in the program that satisfy the stated condition, the constraint makes sure that the properties of the found instances (representing the constraint variables) always remain aligned. Also, while program queries are evaluated at rule application (i.e., constraint generation) time, when all properties have their initial values, constraints are evaluated at constraint solution time, during which the values of the properties may be changed. This latter difference will play an important role below.

[1] Following the convention of [19], we use upper case letters for variables representing declared entities, and lower case letters for variables representing references.

3.2 Conditional Constraints

A *conditional constraint* [7, 11] of the form $P \rightarrow C$ is a constraint over two (reified) constraints, the *premise constraint*, P, and the *consequent constraint*, C. Satisfaction of $P \rightarrow C$ requires satisfaction of C only if P is satisfied; if not, C can be ignored. P can therefore be considered a guard switching C on or off. Conditional constraints are readily handled by contemporary constraint solvers (e.g., [2]).

Conditional constraints have many uses in constraint-based refactoring. For instance, the JLS mandates that if two fields are declared in the same statement, their declared type, τ, must be the same ([6], §8.3). Expressed as a constraint rule:

$$\frac{\textit{same-declaration}(F_1, F_2)}{F_1.\tau = F_2.\tau} \tag{2}$$

If, for some reason, a refactoring required that the declared type of one, but not both, of f_1 and f_2 is changed, the refactoring would have to be refused. However, this rejection may be overly strict, namely if the declaration can be split as part of the refactoring (in which case the constraint need no longer hold). The constraint rule

$$\frac{\textit{same-declaration}(F_1, F_2)}{F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau} \tag{3}$$

in which the property σ represents the statement in which a field is declared, generates a conditional constraint that solves this problem: only if F_1 and F_2 are declared in the same statement need the types of F_1 and F_2 be the same. If the constraint solver can assign $F_1.\sigma$ or $F_2.\sigma$ a new value so that $F_1.\sigma \neq F_2.\sigma$, the declared types of F_1 and F_2 may differ. Thus, the constraint solver can compute that splitting the declaration solves the refactoring problem.

3.3 Specification of Refactorings

Constraint rules are generally independent of refactorings. However, not all constraint rules are applicable or relevant for all refactorings. This is so because not all properties may be changed by all refactorings: for instance, if the intended refactoring is to pull up a field, renaming that field or others that stand in the way of the pulling up may not be compatible with the refactoring intent, so that identifiers are fixed for this refactoring. Thus, the full specification of an intended refactoring (a refactoring problem) involves

- the program to be refactored,
- the set of constraint rules constraining the properties whose changes are associated with the refactoring,
- the concrete refactoring to be performed, as expressed by a selection of properties (usually one) and their new, mandatory values, and
- a specification of the other properties the constraint solver is allowed to change in order to perform the refactoring. [18]

The last item divides the properties extracted from a program into two kinds, those whose values are *fixed* and those whose values are *non-fixed*. This distinction will also play an important role in our treatment of the foresight problem.

4 Constraint Rule Rewriting

Ignoring the operational differences (noted at the end of Section 3.1) between the query *same-declaration*(F_1, F_2) and the constraint $F_1.\sigma = F_2.\sigma$, the two appear to express the same thing in different terms. In fact, considering that both a constraint rule and a conditional constraint are implications of some kind, (3) contains a tautology: either the query of the constraint rule or the premise of the conditional constraint could be dropped without affecting the contribution of the constraint rule to a refactoring. The only caveat is that the choice which one to drop is not free.

To see why this is the case, we have to look at the variability of program properties and the different evaluation times of queries and constraints. If all properties involved in the premise of a conditional constraint are known to be always fixed (i.e., their only allowed values are their initial values), satisfaction of the premise can be computed at rule application time (when the constraints are generated), after the variables in the queries have been instantiated with program elements. Thus, the premise can be pulled up ("promoted") to the rule precedent, transforming (3) to

$$\frac{\textit{same-declaration}(F_1, F_2) \quad F_1.\sigma = F_2.\sigma}{F_1.\tau = F_2.\tau}$$

Since $F_1.\sigma = F_2.\sigma$ as a query has the same meaning as *same-declaration*(F_1, F_2), the former can be dropped, giving us the simplified rule (2). This kind of rule rewriting is worthwhile since it saves the generation of conditional constraints.

If however the properties involved in the premise of a conditional constraint are non-fixed (so that their values may be changed by the solver), satisfaction of the premise cannot be computed at rule application time. In fact, in this case it is even questionable whether an equivalent query should be evaluated at this time, since this restricts the generation of the conditional constraint to program elements fulfilling the premise for the program as is. For instance, the query of (3) requires that F_1 and F_2 are declared in the same statement, so that no (conditional) constraint will be generated for pairs of fields that are not, preventing the solver from merging two field declarations separate at the time of constraint generation into one should their types (become) equal. In that case, the query should be pushed down ("demoted") to the premise of a conditional constraint, transforming (3) to

$$\frac{F_1 \quad F_2}{\textit{same-declaration}(F_1, F_2) \rightarrow F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau}$$

or, since $F_1.\sigma = F_2.\sigma$ and *same-declaration*(F_1, F_2) as constraints are equivalent, to

$$\frac{F_1 \quad F_2}{F_1.\sigma = F_2.\sigma \rightarrow F_1.\tau = F_2.\tau}$$

Generally, if the precedent of a constraint rule contains queries that relate to properties of the program that may change during the refactoring, those queries (rephrased as constraints) should be pushed down from the rule precedent to the premise of a conditional constraint in the consequent of the rule. This *demotion of queries* to premises of conditional constraints serves the generalization of constraint rules (so that more constraints that cover more refactoring problems are generated). Conversely, if the premise of a conditional constraint in a rule consequent can always be evaluated at rule application time (since for every application it constrains only fixed properties), the premise can be pulled up to the rule precedent. This *promotion of premises* of conditional constraints to queries of constraint rules serves the tuning of constraint-based refactoring, by making constraint generation more specific (so that, if the promoted queries are not redundant to existing queries, fewer constraints are generated), and by simplifying the constraints that must be solved. Both promotion and demotion will be made use of in our solution of the foresight problem as presented in Section 6.

5 Quantified Constraints

As pointed out in Section 3.1, constraint rules are implicitly universally quantified over the elements of a program. However, every single application of a constraint rule generates only one instance of the constraints in its consequent. There are situations in which this is insufficient, as demonstrated by the following

EXAMPLE: Beginning with Java 5, a method may be annotated with the `@Override` annotation, in which case the compiler checks that the method overrides a method defined by a superclass. This is captured by the constraint rule

$$\frac{\mathit{overrides}(M)}{\exists\, M' \neq M : M.\lambda <_\tau M'.\lambda \wedge M'.\alpha \geq_\alpha \alpha(M, M') \wedge \mathit{override\text{-}equivalent}(M', M)}$$

which requires that there is at least one method defined in a superclass that is accessible from M and has an override-equivalent signature ([6], §8.4.2). ♦

Unlike conditional constraints, quantified constraints are not readily handled by available constraint solvers. However, since the domains that are being quantified over, namely sets of program elements, are always finite, a quantified constraint can be unrolled to a finite disjunction or conjunction of constraints. For instance, if a program has three methods, M_1, M_2, and M_3, of which M_1 is annotated with `@Override`, application of the above constraint rule unrolls to

$$\begin{aligned} &M_1.\lambda \leq_\tau M_2.\lambda \wedge M_2.\alpha \geq_\alpha \alpha(M_1, M_2) \wedge \mathit{override\text{-}equivalent}(M_2, M_1) \\ \vee\; &M_1.\lambda \leq_\tau M_3.\lambda \wedge M_3.\alpha \geq_\alpha \alpha(M_1, M_3) \wedge \mathit{override\text{-}equivalent}(M_3, M_1) \end{aligned}$$

Contrasting this simple one, below we will encounter examples of quantified constraints whose unrolling is exceedingly expensive.

Quantified constraints also offer opportunities for rule rewriting. Because constraint rules are implicitly universally quantified over their variables, a universally quantified constraint occurring in a rule consequent can be stripped of the quantifier, by moving the quantified variable (representing program elements) to the rule precedent. Effectively, this makes unrolling a universally quantified constraint an immanent part of constraint generation (rule application). This "promotion" of universal quantification will be exploited by our capture of foresight, as detailed in Section 6.

EXAMPLE. The JLS mandates that of all top-level classes contained in a compilation unit, only one may be declared *public*. This translates to the constraint rule

$$\frac{\mathit{top\text{-}level\text{-}class}(C) \quad C.\alpha = \mathit{public}}{\forall C' \neq C,\ \mathit{top\text{-}level\text{-}class}(C'):\ C'.\upsilon = C.\upsilon \rightarrow C'.\alpha <_\alpha \mathit{public}}$$

(in which $C.\upsilon$ represents the compilation unit of C), which is equivalent to

$$\frac{\mathit{top\text{-}level\text{-}class}(C) \quad C.\alpha = \mathit{public} \quad C' \neq C \quad \mathit{top\text{-}level\text{-}class}(C')}{C'.\upsilon = C.\upsilon \rightarrow C'.\alpha <_\alpha \mathit{public}}$$

in which the explicit universal quantification in the consequent has been replaced by the introduction of C' as a variable in the rule precedent whose quantification (and unrolling) is implicit in rule application. If the intended refactoring does not allow moving classes between compilation units, the rule can be further rewritten to

$$\frac{\mathit{top\text{-}level\text{-}class}(C) \quad C.\alpha = \mathit{public} \quad C' \neq C \quad \mathit{top\text{-}level\text{-}class}(C') \quad C'.\upsilon = C.\upsilon}{C'.\alpha <_\alpha \mathit{public}}$$

(by promoting the premise of the conditional constraint to the rule precedent), saving the generation of conditionals for classes of the same compilation unit, and the generation of constraints for classes from different compilation units altogether. ♦

6 A Constraint-Based Solution of the Foresight Problem

The foresight problem exposed by Examples 2–4 of Section 2.1 is that the constraints generated from the constraint rules as applied to the program as is are insufficient: certain constraints are missing. With constraint rule rewriting and quantified constraints at hand, we are sufficiently equipped to systematically generate them.

6.1 Foresight with Constraint Rule Rewriting

Example 2 of Section 2.1 suggests that a constraint should have been generated that constrains the declared accessibility of field i to public *after* its pulling up to a class of another package, a constraint that would however have constrained the program as is incorrectly. Generation of a conditional constraint escapes this dilemma, by guarding the constraint with the condition that the access occurs from another package via a reference whose (static) type is not a (non-strict) supertype of the declaring type of i. This is obtained by rewriting the constraint rule of Example 2, here formalized as

$$\frac{\mathit{binds}(m, M) \quad \mathit{receiver}(m, r) \quad m.\pi \neq M.\pi \quad \neg\, r.\tau \leq_\tau r.\lambda}{M.\alpha = \mathit{public}}$$

to

$$\frac{\mathit{binds}(m, M) \quad \mathit{receiver}(m, r)}{m.\pi \neq M.\pi \wedge \neg\, r.\tau \leq_\tau r.\lambda \rightarrow M.\alpha = \mathit{public}} \tag{4}$$

in which the queries $m.\pi \neq M.\pi$ and $\neg\ r.\tau \leq_\tau r.\lambda$ have been demoted to the guard of a conditional constraint. Applied to the program of Example 2, this rule generates a consequent constraint ($M.\alpha = public$) that is inactive (switched off) for the program as is; if however the program is changed in such a way that the guard holds, the consequent is activated, and contributes to the refactoring. Thus, the rewritten rule codes foresight of the possible change.

As can be seen the use of constraint rules with demoted queries is expensive in that it leads to the generation of more constraints, and conditional ones at that. Therefore, if rule (4) is used in a specific refactoring, say GENERALIZE TYPE [19], fixed constraints should be promoted to queries, in the case of GENERALIZE TYPE leading to

$$\frac{binds(m, M) \quad receiver(m, r) \quad m.\pi \neq M.\pi \quad M.\alpha \neq public}{r.\tau \leq_\tau r.\lambda}$$

if only the declared types of program elements may be changed by the refactoring.

6.2 Foresight with Quantified Constraints

As detailed in Section 2.1, the problem highlighted by Example 3 is of a different nature than that of Example 2 in that a declared entity must be constrained that, for the program as is, is unrelated to the program elements to be refactored. This lack of relatedness suggests that such program elements escape ordinary constraint rules.

This is where quantified constraints step in. For the case of Example 3, that no field must exist to which a reference could bind alternatively (so that the reference would be ambiguous) is conveniently expressed using a non-existence constraint in the consequent of a constraint rule, as in

$$\frac{binds(f, F) \quad receiver(f, r)}{\neg\exists\ F' \neq F : F'.\iota = f.\iota \wedge F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\ F.\lambda <_\tau F'.\lambda} \tag{5}$$

which reads "there must not exist a field F' distinct from F that has the same name (identifier) as f (the reference that must not be ambiguous), that is accessible for f, that is declared in a supertype of the type of receiver r, and that is not declared in a supertype of the declaring type of F". Similarly, the problem exposed by Example 4, namely that no field must exist that hides the field a reference currently binds to, is countered by generating the constraint

$$\neg\exists\ F' \neq F : F'.\iota = f.\iota \wedge r.\tau \leq_\tau F'.\lambda \wedge F'.\lambda <_\tau F.\lambda$$

Note that in both cases, all conjuncts of the quantified constraint but the last equally apply as conditions required for f to bind to F — to avoid ambiguity or rebinding, conditions sufficient for f binding to F must not hold for other fields F' as well.

Unlike in the example of Section 5, unrolling quantified constraints such as the above can be very expensive. In the worst case, if a refactoring may change the name of a field and its location freely, a constraint must be generated for every other field in the program, and with it for every reference to a field (which may have to be renamed as well). However, most refactorings are not granted this freedom, so that constraint rules such as the above (which directly mirror the rules of the programming language) can be rewritten to suit specific refactorings.

Generally, constraint rules expressing that no program element must exist with properties that would infringe the program's well-formedness or change its behaviour have the form

$$\frac{query(e, \ldots)}{C(e, \ldots) \quad \neg\exists\, e' \neq e : C'(e', \ldots)} \tag{6}$$

in which $C(e, \ldots)$ and $C'(e', \ldots)$ represent arbitrary constraints expressing the relationships between the properties of e and others that must hold, and relationships between the properties of e' and others that must not hold. In a first rewriting step, we split the constraint rule (6) into two

$$\frac{query(e, \ldots)}{C(e, \ldots)} \qquad \frac{query(e, \ldots)}{\neg\exists\, e' \neq e : C'(e', \ldots)}$$

and leave aside the first, since it is standard. Next, we split the quantified constraint $C'(e', \ldots)$ into two conjuncts $F(e', \ldots)$ and $N(e', \ldots)$, the former containing only constraints whose constrained properties (constraint variables) are fixed for the refactoring, the latter containing the constraints of which at least one constrained property is non-fixed (note that either conjunct may be empty). This lets us rewrite the rule to

$$\frac{query(e, \ldots)}{\neg\exists\, e' \neq e : F(e', \ldots) \wedge N(e', \ldots)}$$

which is equivalent to

$$\frac{query(e, \ldots)}{\forall\, e' \neq e : \neg(\, F(e', \ldots) \wedge N(e', \ldots)\,)}$$

which is in turn equivalent to

$$\frac{query(e, \ldots)}{\forall\, e' \neq e : F(e', \ldots) \rightarrow \neg N(e', \ldots)}$$

Since the rule consequent is now a universally quantified conditional constraint whose premise depends on fixed properties only, we can rewrite the rule to

$$\frac{query(e, \ldots) \quad F(e', \ldots)}{\neg N(e', \ldots)}$$

whose additional query $F(e', \ldots)$ acts as a filter leading to the generation of fewer constraints than would have been introduced by the unrolling of $\neg\exists\, e' \neq e$: $C'(e', \ldots)$. For instance, for a refactoring that is not allowed to change identifiers or declared accessibilities, rule (5) can be rewritten to

$$\frac{binds(f, F) \quad receiver(f, r) \quad F' \neq F \quad F'.\iota \neq f.\iota}{F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda < F'.\lambda} \tag{7}$$

Note that the conjunct constraining accessibility cannot be promoted to a query, since it depends on location (cf. the definition of α in Figure 2), which may be changed by the refactoring (but see below for further savings possible).

Generally, what seems like a rather discouraging threat to the tractability of constraint-based refactoring with foresight may be tamed by rule rewriting, allowing the evaluation of constraints — as queries — at rule application time. How effective this is in practice will be explored in the evaluation of Section 8.

6.3 Further Savings

The above introduced possible rewritings of constraint rules depend on the (lack of) variability of constrained properties in the rule consequent, more specifically on whether the satisfaction of a constraint can be decided — for every possible application of the rule — at rule application time: if it can, the constraint can be promoted to a query where it acts as a filter causing fewer generated constraints. This lets us tailor general (i.e., refactoring-independent) constraint rules to a specific refactoring characterized by which properties are non-fixed and which are fixed (cf. Section 3.3). This tailoring can be carried out before the refactoring is actually applied, as it holds across all possible applications. Practically, this means that it can be performed for the tuning of a set of constraint rules to a specific refactoring tool.

However, further savings are possible when a (specifically tailored) refactoring is actually performed. When a constraint rule is applied, i.e., when all variables of the rule have been instantiated with concrete program elements, it may be the case that individual constraints to be generated can already be evaluated. For instance, continuing the example of rewriting rule (5) from the previous subsection to rule (7), for all fields F' whose declared accessibility is *public*, the constraint $F'.\alpha \geq_\alpha \alpha(f, F')$ in the consequent of (7) need not be generated, since it is always satisfied (recall that the refactoring was not allowed to change accessibility). Furthermore, reified constraints (cf. Section 3.2) that can be evaluated at rule application time may lead to shortcut evaluations of the Boolean constraints (including conditional constraints) constraining them, which may lead to further savings (including immediate abortion of a refactoring if a top-level constraint is unsatisfiable). These optimizations do not correspond to rewritings of constraint rules, since they are performed individually, for single applications of rules. We will evaluate their impact in Section 8.

6.4 Basic Algorithm of Constraint-Based Refactoring with Foresight

An algorithm for constraint-based refactoring with foresight that performs the possible tailoring described in Sections 6.1 and 6.2 and the additional optimizations of Section 6.3 is shown in Figure 1. It takes as input the parameters necessary to specify a constraint-based refactoring (as detailed in Section 3.3) and produces a refactored program, if the refactoring can be performed.

The algorithm is split into four stages: the rewriting (tailoring) of the constraint rules to the specific refactoring, the application of the rules to the program to be refactored, the performing of the individual optimizations, and the generation and solution of a CSP (including writing back the solution of the CSP to the original program). Some explanations follow:

Algorithm *RefactoringWithForesight*(*P*, *R*, *I*, *F*)
Input:
P, the program to be refactored
R, a set of constraint rules
I, the refactoring intent (a set of properties and their target values)
F(*p*), a filter selecting the non-fixed properties *p*
Output:
C, a CSP
P, the refactored program
Steps:
rule rewriting
1. **for each** constraint rule *r* **in** *R*
2. convert the rule consequent of *r* to disjunctive normal form (DNF)
3. extract the disjuncts of the DNF that are filtered as invariant
4. promote the extracted disjuncts to negated conjuncts of the rule precedent
5. **if** the remainder (variable disjuncts) is not empty, make it the new consequent
6. **else** drop the rule for this refactoring

rule application (constraint generation)
7. **for each** constraint rule *r* transformed as above
8. apply *r* to *P*, by evaluating the program queries and promoted constraints
9. **for each** match, instantiate the constraints of *r*'s consequent

early evaluation
10. **for each** instantiated constraint *c*
11. **if** any of its disjuncts evaluates to *true*, drop the entire constraint
12. **else if** all disjuncts evaluate to *false*, **fail**
13. **else** delete the disjuncts evaluating to *false* from the constraint and add it to *C*

initializing and solving the CSP, and writing back
14. **for each** property *p* **in** *P* constrained by a constraint in *C*
15. initialize *p* with its value from *P*
16. **if** $p \in I$, replace its initial value with that in *I*
17. **if** $F(p) \wedge p \notin I$, set *p*'s domain according to the type of *p*
18. **else** make *p* constant
19. **if** *C* is solvable
20. solve *C*
21. **for each** changed property *p* **in** *P* write back its new value to *P* to reflect the change
22. **else fail**

Fig. 1. Basic algorithm of refactoring with foresight

Step 2: Conversion to DNF is performed after replacing $\neg\exists\, x\colon \varphi(x)$ with $\forall\, x\colon \neg\varphi(x)$ and dropping the explicit universal quantification as shown in Section 4.

Step 3: A disjunct is filtered as invariant if none of its constrained properties may be changed (meaning that its satisfiability depends only on its instantiation during rule application in Step 9). Note that since, at this stage, all properties are properties of unbound variables (the rules have not yet been applied to actual program elements), the filtering condition must hold for all program elements that can be substituted for the variables. More specifically, only filters of the kind "all access modifiers may be changed" can be evaluated at this stage.

Step 4: A set of invariant disjuncts $A_1, \ldots, A_n$ with (variant) remainder B (which may itself be a disjunction) is interpreted as the precedent $A := \neg(A_1 \vee \ldots \vee A_n)$ of

a conditional constraint $A \rightarrow B$, which, as explained in Section 4, can be promoted to the rule precedent (a conjunction of queries), where it is added as $\neg A_1 \wedge \ldots \wedge \neg A_n$.

Step 6: If none of the constrained properties are changeable, the constraint adds nothing to the solution (recall that all constraints are always satisfied initially).

Step 8: Evaluation of the program queries and promoted constraints substitutes the variables of the rules with the program elements matching the queries.

Step 11: Since the constraints c are in DNF, one disjunct evaluating to *true* renders all others irrelevant. For a disjunct to evaluate to *true* at this stage (i.e., before the actual constraint solving), the values of the constrained properties must be invariant; their value is then the initial value (obtained as in Step 15).

Step 12: Since all constraints of a CSP are implicitly conjoined, there is no chance of a solution if a single constraint always (under all assignments) fails.

Step 13: There is a third case since not all disjuncts can always be evaluated at this stage: those whose constrained properties are at least partly variable (as decided by F) depend on values assigned by the solver (Step 20).

The effectiveness of the savings introduced by algorithm *RefactoringWithForesight* depends on the number of disjuncts in the rule consequents (as introduced by the conditional constraints expressing foresight) and on the selectivity of the filter F (Step 3). In particular, if the filter F selects many properties as non-fixed (meaning that many different kinds of changes are allowed), opportunities for rule rewriting are rare. However, in these cases early evaluation may still be effective, especially if only some properties of a specific kind (such as locations of fields) are non-fixed (as is typically the case for filters such as "allow only locations of fields of the same class to change", a filter used by the PULL UP FIELD refactoring). Our evaluation in Section 8 will shed light on the effectiveness of rule rewriting and early evaluation.

7 Implementation

We have implemented refactoring with foresight as described here as an extension to our refactoring constraint language REFACOLA [18]. REFACOLA allows the developer of a refactoring tool to define different *kinds* of program elements (beyond the declared entity and reference distinction made in this paper, e.g., `Field`, `Method`, `Variable`, etc.), and to associate with each kind a fixed set of *properties* (such as the ones listed in Table 1). Each property comes with a *domain*, which may be predefined (such as `Identifier`), enumerated (such as `Accessibility`), or program-dependent (such as `Location`). The REFACOLA language is complemented by a REFACOLA framework which provides a predefined set of program queries, a generic algorithm for applying the constraint rules to a program, an interface to constraint solvers such as Choco [2], and routines for writing back the solved constraints to the program source. The REFACOLA compiler and framework have been implemented as plugins to Eclipse, with adapters for C# and Eiffel compilers. Refactoring specifications in REFACOLA are completely declarative: refactoring tools can be generated from these specifications at the push of a button. The generated tool used for the evaluation in Section 8 (enhanced with a basic user interface) can be downloaded from `www.feu.de/ps/prjs/refacola`.

One of the main contributions of the REFACOLA framework is its *GenerateConstraints* algorithm [18], which keeps constraint-based refactoring tractable by generating only the constraints constraining (properties of) program elements that are, directly or indirectly, related to the code change intended by the refactoring (so that the change can propagate to them). In our current work, the savings achieved by this algorithm appear to be traded for addressing the foresight problem, since quantified constraints providently involve all program elements of a given kind, including ones seemingly unrelated to the refactoring intent (cf. Example 3). However, as we will show next, the promotion of constraints to queries and the early evaluation of constraints allow us to retain much of the original savings in many cases.

8 Evaluation

To be able to judge the impact our solution to the foresight problem has on the viability of constraint-based refactoring in practice, including the effectiveness of the rule rewritings and early evaluation suggested, we have performed a systematic evaluation on the basis of several variants of the PULL UP FIELD refactoring [4] used as an example throughout this paper. We chose PULL UP FIELD because it strikes a good balance between simplicity of the refactoring (so that our focus is not diffused by other problems of refactoring) and occurrence of foresight problems (as suggested by the motivating examples). To be able to assess the impact rule rewriting and early evaluation have on constraint generation, and also the dependence on the permissiveness of the filter F (cf. Figure 1), we evaluated several variants of PULL UP FIELD that differ in the degrees of freedom granted to the refactoring, i.e., whether it is allowed to pull up other fields as well, rename fields, or change their accessibility.

8.1 Specification of PULL UP FIELD with Foresight

The constraint rules immediately relevant for PULL UP FIELD with foresight are shown in Figure 2. The program queries are given expressive names that serve to name the rules also (note how FIELDACCESS(r, f, F) combines *binds*(f, F) and *receiver*(r, f)); their implementation is of no interest here. We have omitted some general rules for enforcing well-formedness of locations (every nested type residing in a top-level type resides in the package the top-level type resides in; every program element residing in a type resides in the top-level type and package the type resides in; etc.), for restricting the accessibility of top-level types (only *package* and *public* are allowed) and members of interfaces (all *public*), etc.

The rules of Figure 2 are explained as follows: FIELDDECLARATION(F) requires that no two fields exist in the same class that have the same name (note that both identifier and location are considered non-fixed by this rule, and all others for that matter). INITIALIZINGFIELDDECLARATION(F,r) adds to it that each field F and the reference r that is assigned to it reside in the same location (*co-location*) and that the (inferred) type of the reference is a non-strict subtype of the declared type of the field (*typing*). THISACCESS(t) is the standard type inference rule for *this*, expressing that the type of *this* (as a reference) is the type it is located in.

FIELDDECLARATION(F)

$$\neg\exists\ F' \neq F : F'.\iota = F.\iota \wedge F'.\lambda = F.\lambda \quad \text{(no name collision)}$$

INITIALIZINGFIELDDECLARATION(F, r)

$$F.\lambda = r.\lambda \quad \text{(co-location)}$$
$$r.\tau \leq_\tau F.\tau \quad \text{(typing)}$$

THISACCESS(t)

$$t.\tau = t.\lambda \quad \text{(typing)}$$

FIELDACCESS(r, f, F)

$$f.\iota = F.\iota \quad \text{(name equality)}$$
$$f.\tau = F.\tau \quad \text{(typing)}$$
$$r.\lambda = f.\lambda \quad \text{(co-location)}$$
$$r.\tau \leq_\tau F.\lambda \quad \text{(member)}$$
$$F.\alpha \geq_\alpha \alpha(f, F) \quad \text{(accessible member)}$$
$$f.\pi \neq r.\tau.\pi \rightarrow r.\tau.\alpha = \textit{public} \quad \text{(implicit type access)}$$
$$r.\tau < F.\lambda \rightarrow F.\alpha > \textit{private} \quad \text{(inherited member access 1)}$$
$$\forall\ T : r.\tau <_\tau T \leq_\tau F.\lambda \rightarrow (F.\alpha <_\alpha \textit{protected} \rightarrow T.\pi = F.\pi) \quad \text{(inherited member access 2)}$$
$$f.\pi \neq F.\pi \wedge \neg\, r.\tau \leq_\tau r.\lambda \rightarrow F.\alpha = \textit{public} \quad \text{(protected accessibility)}$$
$$\neg\exists\ F' \neq F : F'.\iota = F.\iota \wedge F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda <_\tau F'.\lambda \quad \text{(no ambiguity)}$$
$$\neg\exists\ F' \neq F : F'.\iota = F.\iota \wedge r.\tau \leq_\tau F'.\lambda \wedge F'.\lambda <_\tau F.\lambda \quad \text{(no hiding)}$$

$e_2.\alpha \geq_\alpha \alpha(e_1, e_2) \equiv$
if $e_1.\lambda_T = e_2.\lambda_T$ then $e_2.\alpha \geq_\alpha \textit{private}$
else if $e_1.\pi = e_2.\pi$ then $e_2.\alpha \geq_\alpha \textit{package}$
else if $e_1.\lambda \leq_\tau e_2.\lambda$ then $e_2.\alpha \geq_\alpha \textit{protected}$
else $e_2.\alpha \geq_\alpha \textit{public}$

Fig. 2. Constraint rules for the PULL UP FIELD refactoring (excerpt)

The rule FIELDACCESS(r, f, F) requires that: each field F and all references f to it have the same name (*name equality*); that the inferred type of f is the declared type of F (*typing*); that the type of the receiver r is a subtype of the declaring type of F (so that r has f as a member; *member*); that F is accessible from the location of f (*accessible member*); that the receiver type is accessible for f (*implicit type access*); that an inherited member cannot have *private* accessibility (*inherited member access 1*) and must have *public* accessibility if any intervening class on the inheritance path resides in a different package than F; and includes rule (4) of Section 6.1 (*protected accessibility*), as well as the rules from Section 6.2 (*no ambiguity* and *no hiding*).

Finally, Figure 2 shows how the function α is expressed as a nested conditional constraint, specifying how required accessibility of e_2 adapts to changes of location of e_1 and e_2 relative to each other.

As can be seen from Figure 2, the constraint rules FIELDDECLARATION and FIELDACCESS introduce three negated existential quantifications which, given that they are applied to each field declaration and field access of a program and that each one needs to be unrolled to all fields declared in the program, must be expected to lead to substantial numbers of additional constraints. This is countered by the

Table 2. Filters used for specifying the different variants of the PULL UP FIELD refactoring used in the evaluation

FILTER	DEFINITION (SPECIFYING NON-FIXED PROPERTIES)
NOC	$F_l.\lambda$, $F_l.\lambda_T$, and $F_l.\pi$, where F_l is the field to be pulled up
CL	$F.\lambda$, $F.\lambda_T$, and $F.\pi$, where F is all fields of the class of the field to be pulled up
CA	$F_l.\lambda$, $F_l.\lambda_T$, $F_l.\pi$, $F.\alpha$, and $T.\alpha$, where F is any field and T is a (its) type
CI	$F_l.\lambda$, $F_l.\lambda_T$, $F_l.\pi$, $F.\iota$, and $f.\iota$, where F is any field and f is any reference to a field
CLAI	CL ∪ CA ∪ CI

RefactoringWithForesight algorithm of Figure 1, whose rewriting stage, applied for instance with the filter "allow only locations of fields of class *C* to change" (the filter CL of Table 2), transforms the two constraint rules to

$$\frac{\text{FIELDDECLARATION}(F) \quad F' \neq F \quad F'.\iota = F.\iota}{F'.\lambda \neq F.\lambda} \quad \text{(no name collision)}$$

$$\frac{\text{FIELDACCESS}(r, f, F) \quad F' \neq F \quad F'.\iota = F.\iota \quad \ldots}{\begin{array}{ll} F'.\alpha \geq_\alpha \alpha(f, F') \wedge r.\tau \leq_\tau F'.\lambda \wedge \neg\, F.\lambda <_\tau F'.\lambda & \text{(no ambiguity)} \\ \neg\, r.\tau \leq_\tau F'.\lambda \vee \neg\, F'.\lambda <_\tau F.\lambda & \text{(no hiding)} \end{array}}$$

Note that, had the filter been different (for instance, "allow only changes of identifiers"), the transformation would have been different. Also note that, although it is clear from the filter that not all properties λ of a program may be changed by the refactoring (only those of the same class), no further constraints from the rule consequents can be promoted to the precedent, since the rule applies to all properties of all elements of a program. This is different for the early evaluation stage of the algorithm which, when applied to the program of Example 4, generates the constraint set

$$\{\ \mathsf{i}^{\mathsf{Z}}.\lambda \neq_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \mathsf{i}^{\mathsf{B}}.\lambda \neq_\tau \mathsf{i}^{\mathsf{Z}}.\lambda,\ \mathsf{this}_{\mathsf{B}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \neg\, \mathsf{i}^{\mathsf{Z}}.\lambda <_\tau \mathsf{i}^{\mathsf{B}}.\lambda,\ \neg\, \mathsf{this}_{\mathsf{A}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda \vee \neg\, \mathsf{i}^{\mathsf{B}}.\lambda <_\tau \mathsf{i}^{\mathsf{Z}}.\lambda\ \}$$

in which $\mathsf{this}_{\mathsf{A}}$ and $\mathsf{this}_{\mathsf{B}}$ refer to the references to this in classes A and B, respectively, and i^{Z} and i^{B} to the different declarations of i. As can be seen, these constraints (correctly) prevent the pulling up of i from B to A, since this would make the first disjunct of the last constraint ($\neg\, \mathsf{this}_{\mathsf{A}}.\tau \leq_\tau \mathsf{i}^{\mathsf{B}}.\lambda$) *false* without making the second disjunct (which is *false* with the program as is) *true*.

8.2 Variants of PULL UP FIELD

For our evaluation, we defined the PULL UP FIELD refactoring with five different degrees of freedom, based on the definition of the filters described in Table 2:

- Filter NOC (for *n*o *o*ther *c*hanges) specifies the basic variant of PULL UP FIELD: it excludes the automatic pulling up of other fields (as required by Examples 1 and 4) and the automatic adaptation of accessibilities (required by Examples 2 and 3).
- Filter CL (for *c*hange *l*ocation) enables the automatic pulling up of all other fields of the same class to the same target class.

Table 3. Sample projects used as the basis of the evaluation

PROJECT	NO. OF CLASSES	NO. OF FIELDS	PULL-UP OPPORTUNITIES†
ANTLR V3.2	71	118	221
Apac. commons.codec V1.3	19	56	210
Apache commons.io V1.4	74	47	80
Apache math V2.1	178	532	1164
Cream V1.06	32	63	71
Fit V1.1	95	122	237
HTML Parser V1.6	148	275	636
Jaxen V1.1.1	167	142	359
Jester V1.2.2	30	39	41
Junit V3.8.1	105	104	234
PicoContainer V1.3	73	116	348
total	992	1614	3601

† one per field and non-library superclass of the class declaring that field

- CA (for *c*hange *a*ccessibility) adjusts the accessibility of the pulled up field and of its type, if necessary. Any access modifier can be used.
- CI (*c*hange *i*dentifier) allows the refactoring to rename the pulled up field, or the field with the same name, to a fresh one in case of name collision, ambiguity, or hiding.
- CLAI (*c*hange *l*ocation, *a*ccessibility, or *i*dentifier) allows all additional changes.

Note that all of the above variants of PULL UP FIELD are completely defined by specifying the corresponding filter F that is supplied to the *RefactoringWithForesight* algorithm of Figure 1, and by supplying the domains of the properties selected by F (cf. [18]). Of the filters, NOC is the least permissive (allowing the fewest applications of the refactoring, because no other changes can be computed that would make the refactoring possible), and CLAI is the most permissive filter (allowing the most refactorings).

8.3 Experimental Setup and Results

To systematically evaluate our approach, we have applied our implementation of the algorithm of Figure 1 using the ruleset of Figure 2 to the sample programs of Table 3, using the following procedure:

```
let R be the constraint rules of Figure 2
for each sample program P of Table 3
  for each field f and class C of P such that f.λ<τC (ie, f is defined in a subclass of C)
    let I = { f.λ = C } (ie, pull up f to C)
    for each filter F of Table 2
      for each mode m of Table 4
        measure performance of RefactoringWithForesight(P, R, I, F) in mode m
```

The results of this procedure are shown in Table 5. Note that solvability and number of solutions are the same for modes f^{+}, f^{++}, and f^{+++}: this is so because rule rewriting and early evaluation are optimizations that have no effect on the solution space.

Table 4. Modes of application

MODE	FUNCTION
f^-	all foresight problem related constraints disabled
f^+	all foresight constraints, but no rewriting or constant evaluation, enabled
f^{++}	rule rewriting enabled
f^{+++}	early evaluation of constraints enabled

Ignoring foresight problems (mode f^-), PULL UP FIELD is almost always applicable (meaning that the corresponding CSP is solvable), even with no other changes allowed (filter NOC). Its applicability can only slightly be increased by allowing the refactoring to pull up other fields as well (filters CL and CLAI). The picture is entirely different when the constraints covering foresight problems are added (modes f^+, f^{++}, f^{+++}): with no other changes allowed (NOC), applicability is reduced by more than one half (64%). However, this reduced applicability (which prevents the refactoring from producing ill-formed or behaviourally changed programs) can be fully compensated by allowing PULL UP FIELD to make additional changes: with filter CLAI, all foresight problems can be solved by adapting locations, accessibilities, or identifiers, resulting in the exact same applicability (97.3%).

Of the additional changes permitted by CLAI, changing accessibility (filter CA) makes the biggest single contribution: taken alone, it improves applicability by 55.5 percent points. Note that by comparison, changing only identifiers (CI, which is equivalent in effect to Schäfer's name unlocking via insertion of qualifiers [13]) enables only a small fraction of refactorings affected by foresight problems. That allowing PULL UP FIELD to change locations of other fields (CL) increases its applicability in the foresight modes more than it does in f^- is due to the fact that in the former, pulling up a private field that is used to initialize another field requires pulling up the other field as well, if accessibility cannot be changed. This is ignored in mode f^-.

With respect to the cost introduced by addressing the foresight problem, Table 5 shows that without further measures (mode f^+), the rise in the number of constraints generated is dramatic: it 1200-folds on average. As was to be feared, the added constraints linking seemingly unrelated program elements to the refactoring intent reduce the effectiveness of the *GenerateConstraints* algorithm presented in [18] (cf. Section 6.4). However, as can also be seen from Table 5, tailoring constraint rules to a specific refactoring (here: to a specific variant of PULL UP FIELD as represented by a corresponding filter) via rule rewriting and applying early evaluation can reduce the number of constraints substantially: for NOC, CL, and CA it cuts the rise to less than 10-fold. For CI, rule rewriting is not as effective, however: this is so because for the expensive quantified constraints *no name collision*, *no hiding*, and *no ambiguity*, (cf. Figure 2) the constraint $F'.\iota = F.\iota$ cannot be promoted to a query (cf. Section 6.2), as which it would by highly selective (i.e., preclude the generation of many constraints; the number of fields with the same name in a program is usually small when

Table 5. Results of application to the pullable fields of Table 3 (all numbers averaged)

METRIC	MODE		FILTER				
			NOC	CL	CA	CI	CLAI
Solvable	f^-		97.0%	97.3%	97.0%	97.0%	97.3%
	f^+, f^{++}, f^{+++}		35.3%	36.6%	92.1%	38.7%	97.3%
No. of Constraints	f^-		10	36	10	10	36
	f^+		16694	16884	16740	17201	17456
	f^{++}		52	75	75	14227	17452
	f^{+++}		28	69	73	8816	17452
No. of Solutions	f^-		1	1.19	1	1	1.19
	f^+, f^{++}, f^{+++}		1	1.19	2.33	1.09	3.22
Times Required [ms][†]	f^-	gen	3	330	6	10	339
		solv	1	38	1	2	37
	f^+	gen	1251	2572	1807	1325	3191
		solv	69	98	70	94	160
	f^{++}	gen	907	1522	1587	5080	3085
		solv	1	26	4	75	164
	f^{+++}	gen	343	1429	1542	2912	3058
		solv	0	24	3	0	75

[†] all times obtained on contemporary laptops with 2 GHz clock speed running the Windows XP operating system with JVM heap space set to 1 GB, using the Choco [2] constraint solver

compared to the total number of fields). The additional constraints are approximately halved by early evaluation, which can exploit the fixedness of most other properties. However (and as was to be feared), average numbers suggest that both measures remain ineffective for the most permissive filter, CLAI: in this case, almost no rule rewritings and early evaluations seem to be possible.

On average, the number of solutions for all investigated variants of PULL UP FIELD remains within a range that would allow the user of the refactoring to inspect all alternatives and pick the one that is closest to his intent. Somewhat surprisingly, this is also the case for the foresight modes: one might have expected that covering more program elements' properties would lead to considerably more solutions. However, almost all constraints generated for filters NOC, CL, and CI are equality constraints (equality of locations or names) or disequality constraints with binary domains (old and new location or name), and the number of choices for the inequality constraints introduced by CA is limited to four (the number of different access modifiers; cf. Section 8.2).

In terms of the times required for generating (*gen*) and solving (*solv*) the CSPs representing the refactoring problems, Table 5 shows that across all filters and modes, *gen* (which includes querying a database representation of the program), is much larger than *solv*. In fact, while *solv* is negligible on average, *gen* can take up to 5 seconds, which is however still quite fast (but see below). This result reflects the fact that much of the effort previously burdened on the constraint solver has been shifted to the constraint generation phase, where it takes its toll.

Secondly, the time required for generating the constraints with the filters allowing a change of location (CL and CLAI) is significantly larger than for the rest. This is due to the fact that almost every constraint generated contains λ, λ_T, or π; if these are non-fixed,

Refacola's *GenerateConstraints* algorithm [18] will also look at all other properties contained in the constraint and, if variable, how they are constrained further.

The overall favourable times are relativized by three facts:

- The time for constraint generation does not include the time needed for filling the database against which the program queries are evaluated. We have excluded it since it depends largely on the infrastructure provided by the IDE in which *RefactoringWithForesight* is integrated (Eclipse in our case). The brute force approach that we used and that always analyses the whole program, regardless of the intended refactoring, can take up to 2 minutes for the larger projects of Table 3; although this can surely be optimized, one should bear in mind that many refactorings require a whole-program analysis.
- In all four modes, we did not submit generated constraints to the solver whose constrained properties all had fixed values: if such a constraint is satisfied, it does not contribute to the solution; if not, the whole CSP is not solvable. Note that "fixed values" here includes the properties of the refactoring intent, whose forced change to a new value must not be revised by the solver (even though in order to propagate the change, the properties count as non-fixed for *RefactoringWithForesight*; cf. the filter definitions in Table 2). This could of course have been handled by the constraint solver, but since all solvers we have experimented with had problems with large numbers of constraints, we added this optimization (whose integration in the algorithm of Figure 1 would have complicated its presentation).
- Times for generating constraints peaked at almost 2 minutes (for modes f^{++} and f^{+++} and filter CI) and for solving at slightly more than 30 seconds (for modes f^{+}, f^{++}, and f^{+++} and filter CLAI). This suggests that the threat to viability of constraint-based refactoring introduced by the foresight problem is very real; however, as evidenced by the relatively short average times that we observed, it can be counteracted in most cases.

9 Conclusion

Refactorings that change the structure of a program are subject to many restrictions, including ones that become apparent only after the structure has been changed. This is particularly a problem if not all structural changes are known in advance of the refactoring, for instance because some changes are dependent on others, or may or may not be needed to make a refactoring possible. To address this problem, we have identified two measures that complement each other. One turns parts of the precedents of constraint-generating rules to premises of conditional constraints, making the generated constraints more flexible in that they can adapt — during the constraint solution process — to structural changes of the program. The other is the introduction of quantified constraints representing an unlimited number of ordinary (non-quantified) constraints, constraining all conceivable changes that could be performed by a refactoring, including those that will actually be performed (which, therefore, need not be known in advance). Both measures have in common that they may generate significantly more constraints than actually needed for a specific refactoring; we have therefore devised an algorithm that keeps the number of additional constraints low. Experiments that we have conducted suggest that our algorithm can be highly effective, and that refactoring with foresight as proposed in this paper can indeed be feasible.

Acknowledgments. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1. The authors thank Andreas Thies for his contributions to the evaluation.

References

1. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: Proc. of OOPSLA, pp. 265–279 (2005)
2. CHOCO Team choco: an Open Source Java Constraint Programming Library, Research Report 10-02-INFO, Ecole des Mines de Nantes (2010)
3. Donovan, A., Kiezun, A., Tschantz, M.S., Ernst, M.D.: Converting Java programs to use generic libraries. In: Proc. of OOPSLA, pp. 15–34 (2004)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
5. Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., Keller, M.: Efficiently Refactoring Java Applications to Use Generic Libraries. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 71–96. Springer, Heidelberg (2005)
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, `http://java.sun.com/docs/books/jls/`
7. Gottlob, G., Greco, G., Mancini, T.: Conditional constraint satisfaction: logical foundations and complexity. In: Proc. of IJCAI, pp. 88–93 (2007)
8. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. PhD Dissertation, University of Washington (1992)
9. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in Java. In: Proc. of ICSE, pp. 431–440 (2008)
10. Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing Java classes. In: Proc. of ICSE, pp. 437–446 (2007)
11. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proc. of AAAI, pp. 25–32 (1990)
12. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Systems. Wiley (1994)
13. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for Java. In: Proc. of OOPSLA, pp. 277–294 (2008)
14. Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.: Correct Refactoring of Concurrent Java Code. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 225–249. Springer, Heidelberg (2010)
15. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Proc. of OOPSLA, pp. 286–301 (2010)
16. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Trans. Soft. Eng. (2012)
17. Steimann, F., Thies, A.: From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)
18. Steimann, F., Kollee, C., von Pilgrim, J.: A Refactoring Constraint Language and Its Application to Eiffel. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 255–280. Springer, Heidelberg (2011)
19. Tip, F., Kiezun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proc. of OOPSLA, pp. 13–26 (2003)
20. Tip, F., Fuhrer, R.M., Kiezun, A., Ernst, M.D., Balaban, I., De Sutter, B.: Refactoring using type constraints. ACM Trans. Program. Lang. Syst. 33(3), 9 (2011)