Refactoring Functional Programs

Simon Thompson and Claus Reinke
Computing Laboratory, University of Kent

ABSTRACT

Refactoring is the process of redesigning existing code without changing its functionality. Refactoring has recently come to prominence in the OO community. In this paper we explore the prospects for refactoring functional programs.

Our paper centres on the case study of refactoring a 400 line Haskell program written by one of our students. The case study illustrates the type and variety of program manipulations involved in refactoring. Similarly to other program transformations, refactorings are based on program equivalences, and thus ultimately on language semantics. In the context of functional languages, refactorings can be based on existing theory and program analyses. However, the use of program transformations for program restructuring emphasises a different kind of transformation from the more traditional derivation or optimisation: characteristically, they often require wholesale changes to a collection of modules, and although they are best controlled by programmers, their application may require nontrivial semantic analyses.

The paper also explores the background to refactoring, provides a taxonomy for describing refactorings and draws some conclusions about refactoring for functional programs.

1 INTRODUCTION

Refactoring is 'improving the design of existing code' and as such, it has been practised as long as programs have been written. Its key characteristic is the focus on structural changes, strictly separated from changes in functionality.

In his 1978 (!) ACM Turing Award Lecture [3], Robert Floyd argued that serious programmers should spend part of their working day examining and refining their own methods: "After solving a challenging problem, I solve it again from scratch, retracing only the *insight* of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems, that *would* have led me to approach the given problem in the most efficient way the first time.". Identifying "paradigms of programming" in this way – and developing support for such paradigms – would improve programmer abilities, computer science teaching and learning, and language designs.

In practice, industrial software development projects will not be restarted from scratch when they have already reached their prime objective. Nevertheless, the idea of continuous design improvements finally became attractive and feasible because: (a) the increasing pressure of maintaining the design quality of long-living "legacy software" in spite of large numbers of modifications, and (b) the realisation that the necessary redesign could be achieved in an incremental fashion, by employing *program transformations*.

Adapting program transformations originating in derivational (or transformational) program development [1, 2] for languages with side-effects, Griswold introduced the idea of automated program restructuring to aid software maintenance [5, 6]. The techniques were extended to cover object-oriented language features, and have recently come to prominence in the OO and extreme programming (XP) communities [4, 9] under the name of *refactoring* (`http://www.refactoring.com`).

Given that functional program transformations were investigated very early on, it is somewhat surprising to see this particular use of program transformations almost exclusively limited to OO languages. Functional programmers might quip that the problems of inflexible program structures are more pressing in OO languages, triggering the need for complex program manipulation techniques to compensate for shortcomings in the languages. OO programmers might retort that functional programs are largely academic in nature and rarely reach the necessary complexity or longevity to expose this kind of problems. These positions are not obviously wrong or right, and we have started to investigate refactoring in functional languages, both to answer this kind of question and to make refactoring techniques and tools available to functional programmers.

The aims of the present paper are threefold: we want to alert the semantics-based program manipulation community to this use of program transformations in the OO and XP communities. Secondly, we introduce and investigate the concepts of functional refactoring in the context of a small case study, refactoring a 400 line student program. Finally, we present some preliminary conclusions and outline our approach towards improving support for refactoring in functional languages.

2 REFACTORING

Refactorings are source-to-source program transformations that change program structure and organisation, but not program functionality. They are the source-level realisation of software redesign. Typically, they either precede changes of program functionality, by adapting program structure for the intended changes, or they follow changes of program functionality, cleaning up the structure of programs after a series of modifications. In either case, changes in program structure and functionality are kept separate: the first are complex, but should preserve functionality, so they need not introduce any bugs, whereas the second do change functionality, making it more difficult to identify potential bugs introduced in the process, but they can be kept free of structural complexities (by suitable refactorings).

When does refactoring arise? To take an example, we might first program a system using an algebraic data type, and then decide to change the way that the data are represented. How should we proceed with this? One option is to modify the `data` type directly, that is to achieve the modification in a single step: this will require us to make substantial modifications to a program's functionality *and* structure simultaneously.

On the other hand, we might do the same in two stages. First we could transform the algebraic data type into an abstract data type (ADT), and only after this *refactoring* is done, would we modify the definition of the ADT. This two-stage transformation aims to separate the structural changes (from algebraic to abstract data type) from the changes in functionality. It also makes the program more amenable to further change, as ADT representations can be modified with no cost to the data type user.

We refactor in other situations too. We might program in an *exploratory* way: first establishing the functionality we seek, and then refactoring into a more elegant form. Our experience leads us to suspect that functional languages are particularly suited to this form of programming, because their clean semantic basis makes wholesale transformations more feasible than for a language in the C family, say.

Finally, we might refactor to *understand* code written by someone else, and this is the focus of the case study later in the paper, where we try to understand a non-trivial student assessment written in Haskell.

3 THE NATURE OF REFACTORING

One of the simplest refactorings renames a function to reflect its use. We have already discussed the rather more complex refactoring from an algebraic to an abstract data type. These examples share two important characteristics of refactorings.

Diffuse Their effect is diffuse, in that they require changes throughout a module and indeed throughout a system of modules. A change of function name needs to be effected at each function call; a change from a `data` type to an ADT will require changes to all functions that directly manipulate the data by pattern matching, for instance.

Bidirectional A change from a general name (e.g. `f`) to a more specific one (e.g. `findMaxVolume`) might later be followed by a change to a more general name (e.g. `findMax`).

We have discussed the change from an algebraic data type to an ADT, but in other situations it is perfectly reasonable to change an ADT into an algebraic type. One motivation might be to use pattern matching, which leads to very concise programs in an equational style.

4 SUPPORTING REFACTORING

We have seen that refactorings have a bureaucratic aspect: changes have to be made at all sites that a function is called, for example. With current technology we would use a text editor to assist in making the changes, and rely on a type checker to catch any errors introduced in the refactoring. OO refactorers underline the importance of continual (re)testing of code to ensure correctness [**?**].

It is important to document refactorings. Fowler's catalogue of OO refactorings [4] explains the general ideas, opportunities and traps. More importantly, it provides OO refactorers with guidelines on how practical refactoring tasks can be achieved by series of smaller refactorings, and it documents the mechanics of individual refactorings, helping refactorers to ensure that they have taken all relevant aspects and potential problems into account.

Moreover, it is entirely feasible to support these refactorings in a variety of *tools* of increasing levels of sophistication; the experience of the OO community [11] in this respect is broadly positive. A tool could allow users to do and undo refactorings of various sorts; it could also check the applicability of certain transformations, such as renaming or lifting. To enable complete rollback of partially executed refactorings if any of their component steps fails, support for transactions is needed; to ensure that refactorings are completed and that refactorers do not lose sight of their original goals, automatically maintained todo-lists would be helpful. More detailed considerations of tool design are to be found in [**?**].

For a functional programming language one could use reasoning to establish the correctness of many classes of refactorings. One class of refactorings corresponds to the rules in an operational semantics and thus these refactorings will be correct by definition. In the absence of a formal semantics for Haskell one can gain substantial assurance from the intuitive semantics of the language. In practice, refactorings will be larger, but one way of validating these is to see them as composed of many smaller, validated refactorings.

Many refactorings have non-trivial syntactic, typing or static semantic preconditions; one can expect tool support to play a role here. On the other hand, some larger-scale refactorings rely upon the maintenance of complex whole-program invariants, and in the absence of full theorem-proving support one must rely upon a combination of programmer intuition and extensive testing to validate instances of these refactorings.

5 A CASE STUDY

The ideas in this paper stand or fall by how useful they prove to be in practice: to explore this we now look at an illustrative example of refactoring. The particular example refactors a Haskell program designed to implement semantic tableaux for propositional logic, written by a second year undergraduate student at the University of Kent. The aim of the refactoring here is twofold.

- Refactoring helps the refactorer to *understand* the program by transforming it incrementally into a program which conforms to their idiomatic style rather than the original programmer's.
- Refactoring is used to *simplify* the program and to make it more amenable to change: in this case by extending it to deal with predicate logic, say.

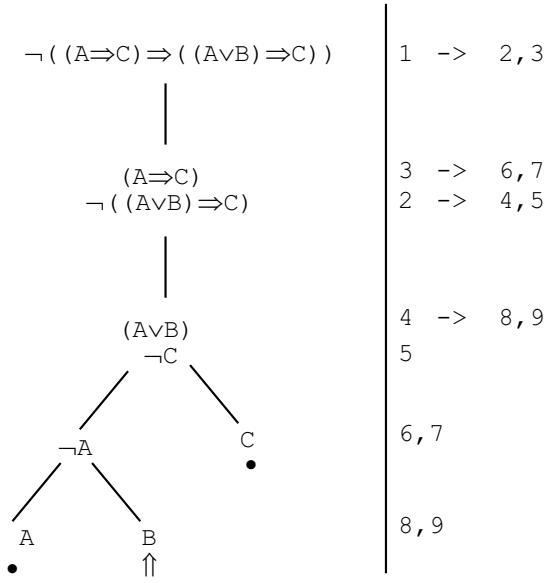

Figure 1: Semantic Tableau Example

5.1 Semantic Tableaux

A semantic tableau for a formula of propositional logic gives a systematic mechanism for finding a valuation which makes the formula true. The tableau systematically decomposes formulae according to their syntax.

In the case of implication, for instance, to satisfy the formula $\neg(\phi \Rightarrow \psi)$ it is necessary for ϕ to be true and ψ to be false: that is, both ϕ and $\neg\psi$ must be satisfied. In the example tableau in Figure 1, stages 1 and 2 show this. On the other hand, to satisfy $\phi \Rightarrow \psi$ there are two alternatives: it is sufficient to satisfy either $\neg\phi$ or ψ; hence the branch in the tableau given by decomposing formula 3.

The branches of the tableau represent possible ways of satisfying the formula at the top. Not all of them need be consistent, and indeed if none is then the formula at the top is unsatisfiable (and so its negation is valid). In Figure 1 the leftmost and rightmost branches are inconsistent (as indicated by •) but the middle branch (⇑) indicates that the formula is satisfiable if `A` and `C` are false and `B` is true.

5.2 The original program in a nutshell

The initial program is written in a concrete, first-order style. Thus, propositions are represented by an algebraic type, branches are given by lists of propositions and tableaux by lists of lists. Functions are defined using explicit recursion, and few library functions are used (`and` is an exception); one function definition uses a list comprehension. The program is a literate script that is 402 lines long, including comments.

The core of any tableau algorithm is an iteration in which rules are applied successively until no expandable formulas remain. There is a wide choice of implementations: the program in question applies a single rule to each branch at each step. Moreover, the rules are applied in a specified order, which is described in the program by an `Int`; for each branch under consideration this value is calculated and then passed to the iteration function to indicate the rule to be applied.

The full sequence of refactorings is available at `http://www.cs.ukc.ac.uk/people/staff/sjt/Refactor`; it is instructive to compare different versions using the vdiff tool: further details of this can be found at `http://www.cs.ukc.ac.uk/development/kst/descriptions/vdiff.html`.

5.3 The refactoring sequence

The example was refactored in a series of steps, which we list now in exactly the form that they were performed. Upon reflection they might have been performed differently, or in a different order. Discussion of this and other observations follow in the next section.

Stage 1. The program uses lists of Propositions – `Prop`, an algebraic data type – to represent branches, and the tableau itself is represented by a list of lists. At this stage the two types are named:

```
type Branch  = [Prop]
type Tableau = [Branch]
```

The purpose of this is twofold: it makes the script easier to read, avoiding potential confusion between `[Prop]` and `[[Prop]]`; more significantly it is a first step in making it possible to change the representation of branches and tableaux.

Stage 2. A number of functions are renamed to reflect their purpose better. For instance, `removeBranch` is renamed `removeDuplicateBranches` and `remove` becomes `removeDuplicatesInBranches`; the former function removes duplicate branches and the latter removes duplicate propositions *within* branches.

Stage 3. The file is transformed from a literate script – in which program text is explicitly flagged – into a standard script where comments are indicated explicitly. This is done because trivial errors were being introduced on the erroneous assumption that the file was a standard script.

Stage 4. In the original system almost all functions over lists are defined using explicit recursion. At this stage a number of these are replaced by calls to appropriate higher-order functions from the prelude or libraries. For instance,

```
displayBranch :: Branch -> String
displayBranch [] = []
displayBranch (x:xs) = (show x) ++ "\n"
                     ++ displayBranch xs
```

is replaced by

```
displayBranch = concat . map (++"\n") . map show
```

This refactoring makes the program more abstract: pattern matching over a concrete type is replaced by calls to functions like `map` whose analogues might appear in the interface of any collection type.

Stage 5. Not all functions can immediately be replaced by calls to library functions. An example is `removeDuplicateProps` of type `Branch -> Branch` which removes duplicate propositions from a branch. It uses the auxiliary function

```
findProp :: Prop -> Branch -> Prop
findProp z [] = FALSE
findProp z (x:xs)
    | z == x    = x
    | otherwise = findProp z xs
```

to check whether a proposition (`x`, say) is contained in a branch or not: if it is, then `x` is returned; if not, the proposition `FALSE` is the result.

How is this function used within the calling function `removeDuplicateProps`? Its result is tested against `x` in a guard: `x == findProp x xs`, so `findProp` is effectively only used to return a Boolean value indicating whether or not `x` is an element of `xs`. Thus it is possible to redefine `findProp` to return a `Bool`:

```
findProp :: Prop -> Branch -> Bool
findProp z [] = False
findProp z (x:xs)
    | z == x    = True
    | otherwise = findProp z xs
```

and to replace the guard by the expression `findProp x xs`.

This redefinition is correct only in certain circumstances.

- We make the *closed world* assumption that we know all the calling sites of the function, and can modify them. If `findProp` were in the interface of a module then a type change like this would be problematic.
- It is assumed that `x` will never be the proposition `FALSE`, as otherwise the behaviour of the old and new definitions will be different. Such an invariant might, for example, be justified by reference to the original specification of the problem, or could perhaps be inferred from the behaviour of the remainder of the program.

Stage 5.1. It is now clear that `findProp` is simply a redefinition of `elem`; similarly, the function `removeDuplicateProps` should be an instance of `nub`. These refactorings are also made to the function which removes duplicate branches, `removeDuplicateBranches`, since it is defined in an analogous way.

Stage 5.2. However, testing the result of this second replacement reveals a problem. With the standard definition of `nub` duplicate branches (with their elements ordered differently) appear in the test result. The reason for this is that the program uses an alternative definition of `nub` that selects the *last* occurrences of duplicated elements rather than first occurrences as does `nub`. The variant definition, `nubVar`, is added to the file.

At this stage, it becomes apparent that the program is sensitive to the data representation and control flow in a way that could not be expected from the problem specification.

Stage 6. The definition of `nubVar` is moved into a separate module, which is then imported into the program itself.

Stage 7. This is a housekeeping stage.

- More functions are renamed, including `foo` and `bar` (really!).
- The function `looseEmptyLists`, which removes empty lists from a `Tableau`, is identified as an instance of `filter` and then its single call is inlined.
- An auxiliary function is moved into a `where` clause.

This tidies up the script for the next major stage.

Stage 8. There are nine different tableau rules for classical propositional logic: two rules for each of the binary connectives, one each for the un-negated and negated case; there is also a rule to eliminate double negations. The implementation defines three functions for each rule. For instance:

```
splitNotNot :: Branch -> Tableau
splitNotNot ps = combine (removeNotNot ps)
                         (solveNotNot ps)

removeNotNot :: Branch -> Branch
removeNotNot [] = []
removeNotNot ((NOT (NOT _)):ps) = ps
removeNotNot (p:ps) = p : removeNotNot ps

solveNotNot :: Branch -> Tableau
solveNotNot [] = [[]]
solveNotNot ((NOT (NOT p)):_) = [[p]]
solveNotNot (_:ps) = solveNotNot ps
```

This design is undesirable for various reasons: a substantial amount of code is repeated (e.g. the definition of the `split` functions), and it also makes it difficult to modify the code, by adding, for instance, the rules for a new connective.

The goal of this stage of refactoring is to produce three functions `split`, `remove` and `solve` which work for all connectives. The refactoring performed here somewhat modifies the algorithm: other less radical refactorings would avoid that. In the original system, the rules are applied in a fixed order of priority; in the new version within each branch the first decomposable proposition is identified and the corresponding rule is applied to it. The effect of this is to compute the same results, but in a different order. It is a matter of debate whether this should be seen as a refactoring, or goes beyond what is legitimate. We discuss this further in Section 6.

How does the redefinition proceed? Examining the code above, it is clear that `split` should be defined to `combine` the results of `solve` and `remove`. A general `solve` is defined by selecting from each of the `solveXXX` functions the line which embodies the rule (as emphasised in *`solveNotNot`* above) and making this one of the cases in the general function. The general `remove` function deletes the first decomposable element from the branch.

Stage 8.1. The redefinition of the control flow of the algorithm changes the behaviour of the tableau mechanism. The algorithm produces duplicate branches (with different orderings) and to avoid this, `map sort` is added to the top-level composition of functions to sort each branch (using the derived order on `Prop`), prior to duplicate removal.

Stage 9. Many of the difficulties in the earlier refactorings have come from the use of lists (and lists of lists) to represent branches and tableaux. At this stage the type representations were modified to

```
type Branch  = Set Prop
type Tableau = Set Branch
```

Function definitions have then to be modified. This is helped by earlier stages (especially 4,5,7,8) which have replaced many explicit pattern matches over lists by calls to combinations of library functions. These calls can be replaced by the corresponding functions over an implementation of sets.

The function controlling rule application uses a primitive recursion, and a corresponding function

```
primRecSet :: (a -> Set a -> b -> b)
            -> b -> Set a -> b
```

needs to be added to the `Set` library. Of course, one needs to apply such fold-like functions with care: to ensure a sensible result they must only be applied to functions that are commutative, associative and idempotent.

Aspects of the algorithms work element-by-element, and at some points in the code, the `flatten` function is used to transform a set into a list and `pick` is used to select an element from the set.

Stage 9.1. The overall effect of this refactoring is a drastic simplification. There is no need to include functions which reorder lists, or remove duplicates from them, since the equality of `Set` is order-insensitive and sets do not contain repetitions. It is therefore much easier to see the essence of the algorithm rather than to have it entangled with functionality designed to maintain implicit invariants on data representations.

Further stages. The refactoring into `Set` suggests further steps. It would be possible to remove references to `flatten` and `pick` if at each stage *all* possible rules are applied to a branch, rather than applying them one at a time.

6 LESSONS LEARNED

In this section, we reflect on our experience of refactoring the tableau program. We begin by making some specific observations and then draw some more general conclusions.

- The granularity of the different stages is quite different: some are simple, others complex. In particular, Stage 5 comprises a number of separate steps. Whilst these steps could have been presented as separate refactorings, they represent – from point of view of the user – one logical step in the transformation of the program.

- The order of refactorings is somewhat arbitrary: in retrospect it would have made more sense to move from literate `.lhs` to standard `.hs` scripts at the start of the refactoring process.

- In the actual refactoring sequence, an error was introduced at stage 4, and only discovered at stage 5. This raises a number of issues.

  – Just because a change is type correct, it doesn't mean that it is completely correct. We shouldn't therefore rely only on type checking to validate steps, but also perform tests on completion of each stage.

  – On the other hand, type checking is really very useful: it is completely automatic and has almost no cost to the programmer in comparison with testing.

Beyond these points, which are specific to the example, one can draw some more general conclusions about the refactoring process.

- Refactoring is an exploratory process. As you refactor you discover more about the program. At the first stage you have a general idea about what is going on in the program, by forming an abstract model of the system. As you refactor you discover that features of the program that you had initially overlooked – subtle details of control flow or data representation, say – are in fact central to the program's behaviour. This allows you to build a more accurate model of the system as refactoring proceeds.

- One can take the modelling insight further: the abstract model can become a goal for the refactoring sequence. In the case study it became clear that the list representation of tableaux was problematic, but a more abstract view of the system again became valid after the final refactoring which replaced lists by sets throughout, and thus removed the program's explicit dependence on lower-level representation aspects.

- One might strengthen the point about exploration to say that it is almost impossible to understand a program passively, and that refactoring – like walkthroughs and code reviews – provides a deeper insight into the workings of a program than can a mere reading or execution.

- The case study also brings to our attention the question of what properties of the system are to be preserved. A hard-line view of refactoring would suggest that one should preserve every property, but in fact a weaker equivalence might be permissible. The refactoring sequence presented here takes this higher-level view, but we have also done an alternative sequence that does not assume knowledge about invariants and has to keep closer to the original algorithm.

  The problem is well known to those working with legacy systems. In integrating a legacy system with other systems it is important to preserve only those properties that are essential to its behaviour, and to neglect those which are merely accidental: those which are an artifact of a previous implementation discipline. Unfortunately, these differences are often undocumented, and some users of the system may have started to rely on accidental properties.

  More formally, a weaker equivalence might preserve observational behaviour only over restricted subsets of inputs. In turn, such inputs might themselves be characterised as those which have a certain invariant property, such as being balanced or search trees, rather than abritrary binary trees. Crucially any such weaker equivalence is dependent upon the *context* in which the program or program fragment is used.

- In any non-trivial refactoring sequence, we need to have version control: providing undo, revert, redo and so on. Indeed, one model would build a list of transformations by analogy with a list of tactics in an LCF-style theorem prover like Isabelle [**?**].

  The advantage of tactics is that they are more abstract than specific syntactic transformations, and so stand more of a

chance of being reusable. Such tactics can also provide a basis for user-defined refactorings.

- Machine support for refactoring is highly desirable. In the case study we were able to use the search/replace and undo/redo functions of an editor as well as the type checking provided by a Haskell system. However, the editor's textual search and replace had to be used with care. It is easy to envisage support for some of the refactorings, such as renamings, and indeed function redefinitions, by more semantics- and refactoring-aware tools and it is also possible to see this being integrated with editors and type checkers. Other refactorings provide more of a challenge: for example when replacing lists with sets the program goes through a long period of being in an inconsistent state, where some functions operate over converted data and others not.

A final observation from the experience of the case study, and indeed from prior OO experience: it is crucial to document refactorings in detail, so that they can be (re-)used with confidence, both by their author and others. This we turn to in the next section.

7 DOCUMENTING REFACTORINGS

Our first draft catalogue of functional refactorings employed a simple format, giving for each refactoring a number, a descriptive name, code examples for both sides of the refactoring, with comments on the pros and cons of the two variants. Any other potentially useful information went into a free-text comment for the refactoring as a whole. The sole purpose of the catalogue was to serve as printed documentation.

This format turned out to be insuffient, both from a maintenance point of view, as we expect the catalogue to evolve continuously, and as a basis for theory and tool development, forcing us to reconsider the format before the development of the catalogue could continue. The new format is more fine-grained and aims to separate information according to expected uses. For instance, conditions for the applicability of refactorings will later need to be formalised to prove that refactorings are functionality preserving (and in what sense); these condition will also serve as the basis for program analyses in refactoring tools. Keeping such information separate from the optional free-form comments forces them to be documented.

Figure 2 gives an example refactoring in the new format. Names of refactorings have evolved into *descriptive phrases*, which are also the main form of references to refactorings. Numbers have been dropped completely, where necessary, a *short name* or label is used for internal references. Examples of *left-hand code* and *right-hand code* together with *comments* on their pros and cons are still used, catering for the role of a refactoring catalogue in the documentation of program design patterns.

New sections include a section on *cross-references*, both internal, to related refactorings in our catalogue, and external, e.g., to related refactorings in Fowler's catalogue. Next, the refactoring is classified as either *primitive* or as *composed* of other refactorings, and further *classifiers* in the form of keywords allow for *flexible categorisations* of refactorings (e.g., type-level or language-level refactorings). Language dependencies will be documented either by explicit references to applicable languages (Haskell, ML, etc.) or by listing of the language features involved (lazy/strict, type classes, functors, ...). Note that refactorings may introduce or eliminate specific language features, so that language dependencies may differ for both sides of a refactoring.

Crucial for the development of theory and tool support is the documentation of *pre-, post- and side-conditions*, which determine the applicability of refactorings and the *kind of program equivalence* preserved. Currently, these sections guide human refactorers around pitfalls and alert them of potential problems. Later, they will serve as the starting point both for proofs of such equivalences and for program analyses and program transformations in refactoring tools. The description will also need to include clerical details such as *version information*, date of addition to the catalogue, etc. which are probably best left to a version management tool.

The purpose of the catalogue has thus begun to shift from simple textual documentation towards a knowledge base from which, e.g., the primitive, Haskell-specific refactorings can be extracted more easily, and from which several forms of documentation can be derived, including hyperlinked PDF documents. The main focus will still be on human readability, but the refined taxonomy prepares for the more formal specification of refactorings that will be necessary for building functional refactoring tools.

8 RELATED WORK

There is a long history of program transformation for functional programs, with early work in the field being described in [10]. Representative of this is the program derivation work of Burstall and Darlington [1], whose transformations work over source-level programs to build more efficient versions of algorithms for sequential or parallel machines. In the extreme case, a non-executable specification is transformed into an executable program. Later work in this vein is exemplified by the relational approach of Bird and de Moor [**?**].

Program transformations are also used automatically within compilers, acting either on source level programs or their intermediate language representations: an example of this approach is discussed in [8].

Refactoring is different from both program derivation and optimisation. These 'vertical' transformations tend to be localised in addressing a program's control or data flow. The kind of program structure considered for refactoring is often non-localised and related to the overall program design and knowledge representation, i.e., to large-scale declarative aspects rather than smaller-scale operational ones.

OO refactoring was first addressed by Griswold [5, 6] and Opdyke [9]. An approachable exposition of refactoring in OO together with a catalogue of refactorings is given by Fowler in [4]. The catalogue is kept up to date at `www.refactoring.com`, which also has links to tools and other resources. The most widely-known tool for OO refactoring is the Refactoring Browser for Smalltalk [11], but tools for other OO languages are under development.

9 CONCLUSIONS AND FUTURE WORK

In this paper we have shown how the ideas of refactoring, which have in the last few years become prominent in the OO community, are equally important in functional programming. Indeed, the 'code then revise' style of programming is perhaps the approach most naturally adopted by many functional programmers.

The paper used the case study in Section 5 to illustrate the idea of refactoring in a functional context, and Section 6 drew a number of

Lifting definition/ Demoting definition

```
f x y = ... (h x y z) ...

h x y z = ...
```

```
f x y = ... (h x y z) ...
        where
        h x y z = ...
```

Lifting definition ⟸

Lifting `h` to the top-level makes it accessible to the other functions in the module containing `f`, and prepares for export of `h` from that module. Enables reuse of auxiliary definitions.

Pre-Conditions

`h` is not already defined in outer scope (if necessary, use *Renaming* first); `h` does not depend on definitions or parameters local to `f` (if necessary, use *Close Definition* first).

Cleanup/Follow-Ons

Examples

An example is when `f` acts as a wrapper for `h`, computing the parameters for `h` from its own parameters, but `h` may be sensibly called with other wrapper functions, or on its own.

⟹ Demoting definition

Demoting `h` to a local definition block clears up the namespace (of the module containing `f` and `h`, and of the whole system). Useful for auxiliary definitions that are only used in a single context.

Pre-Conditions

`h` is not used elsewhere in outer scope.

Cleanup/Follow-Ons

In a more local scope, it is often possible to choose a more concise name for the auxiliary definition (*Renaming*), or to gain implicit access to local definitions and parameters (*Open Definition*).

Examples

An example is where `h` takes extra parameters, and `f` is simply a wrapper for `h` that supplies the initial values of these parameters. A concrete example is given by a search function which takes a list of already-visited nodes as arguments; at the top level this will be called with an empty list. Another justificication is in creating *circular* data structures: a `where`-defined `h` such as `h = x:h` will create a circular representation of the infinite list of `x`, for each call of `f`.

Potential Problems

In the special case of `h` not having any parameters, the demoted form leads to re-evaluation, whereas the lifted form leads to earlier evaluation in strict languages and to a potential space leak in lazy languages. In languages with implicit export (e.g., empty `export` list in Haskell), the potential for scoping conflicts affecting lifting or demoting is aggravated.

Comments

Lifting and demoting of definitions is also possible between adjacent levels of nested definition blocks, but too deeply nested definition blocks tend to reduce program readability.

Primitive/Composed

This is a primitive refactoring.

Classification

Classifiers: language-level, scope-related.
Language features involved: local definitions (`let` or `where`).

References

Cf. Fowler's *Move Field/Method*, *Push Down Field/Method*, *Pull Up Field/Method*. Together with *Close Definition*, *Lifting definition* is often applied as part of λ-lifting [7], and has subtle interactions with the treatment of polymorphic recursion [12].

Figure 2: An example entry from the catalogue of refactorings

both specific and general conclusions from this which we do not reiterate here. Section 7 discussed the documentation of refactorings, and it is instructive to observe that the example of lifting/demotion given there shows how even the simplest of refactorings can have subtle constraints on its application.

There is ample scope for further work in this area. We aim to build a comprehensive and detailed catalogue of functional refactorings, starting from our first draft attempt [**?**]. To do this we will consult practising functional programmers and program transformers; we will also pursue a number of other case studies.

Finally, we intend to build tool support for refactoring which can be integrated into standard functional program development tools.

REFERENCES

[1] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[2] J. Darlington. Program Transformations. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[3] Robert W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8), August 1979. Also appears in ACM Turing Award Lectures: The First Twenty Years 1965-1985.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. see also: `http://www.refactoring.com/`.

[5] William G. Griswold. *Program restructuring to aid software maintenance*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1991. Tech. Rep. No. 91-08-04.

[6] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[7] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy, France, LNCS 201*. Springer Verlag, 1985. (also as part B of author's thesis).

[8] S.L. Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *European Symposium on Programming (ESOP'96)*, April 1996.

[9] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[10] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3), September 1983.

[11] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS), special issue on software reengineering*, 3(4):253–263, 1997. see also `http://st-www.cs.uiuc.edu/users/brant/Refactory/`.

[12] Peter Thiemann. ML-Style Typing, Lambda Lifting, and Partial Evaluation. In *CLAPF '99, Recife, Pernambuco, Brasil*, March 1999.