Refactorings that Split and Merge Programs

Christopher Brown[1] and Simon Thompson[2]

[1] University of Kent, Canterbury, Kent, UK.
cmb21@kent.ac.uk
[2] University of Kent, Canterbury, Kent, UK.
S.J.Thompson@kent.ac.uk

Abstract. Program slicing is a well understood concept in the imperative paradigm, but so far there has been little work on program slicing in the context of functional languages. This paper describes a program slicing technique for Haskell that takes tuple-returning functions apart (called splitting); the converse of this is also described (called merging). The slicer is implemented as a transformation for the Haskell Refactorer, HaRe. Splitting functions is a useful transformation to allow the programmer to extract a particular subset of the functionality of a tuple-returning function into a new definition. Merging is a useful transformation because it allows many definitions to be merged together, thus eliminating duplicate code and encouraging code reuse. Splitting and merging can help to reduce dead code and increase program productivity and can be also used for debugging purposes.

1   Introduction

Refactoring was first introduced by Opdyke in his PhD. thesis in 1992 [10]. Refactoring is the process of changing the internal structure and organization of a program, while preserving its semantics. The key aspect of refactoring —in contrast to general program transformations, such as genetic programming [4] — is the focus on purely structural changes rather than changes in program functionality. Refactoring also contrasts with other meaning-preserving transformations which emphasize a change in efficiency or other non-functional aspects. Refactoring is aimed at improving code quality, increasing programming productivity and increasing the ability for code to be reused. This functionality-preservation is crucial so that refactorings do not introduce, or remove, any bugs.

This paper is concerned with the investigation of a number of refactorings for Haskell. We start with dead code elimination and based upon that the process is extended to introduce a notion of function splitting and merging. As an example of merging and splitting, consider the following Haskell library functions, `take` and `drop`:

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ []= []
```

```
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"

drop :: Int -> [a] -> [a]
drop 0 xs          = xs
drop _ []          = []
drop n (x:xs)
  | n>0  = drop (n-1) xs
drop _ _ = error "PreludeList.drop: negative argument"
```

As a concrete example of the usage of `take` and `drop`, consider:

```
> (take 10 "hello world", drop 10 "hello world")
> ("hello worl","d")
```

A merge refactoring allows the creation of a function which provides both results using only one list traversal rather than one traversal for each of `take` and `drop`. In the following example, `splitAt` is the result of merging `take` and `drop`:

```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:ys,zs)
    where
    (ys,zs) = splitAt (n-1) xs
splitAt _ _ = (error "PreludeList.take: negative argument",
               error "PreludeList.drop: negative argument")
```

The following is an example of the usage of `splitAt`:

```
> splitAt 10 "hello world"
> ("hello worl","d")
```

Splitting is the converse of merging, for example, the extraction of the definitions of `take` and `drop` from `splitAt`. Although `splitAt` is a predefined Haskell function, it serves as a useful example to illustrate splitting and merging. The process of these refactorings is described in section 3.1.

The refactorings are implemented in the Haskell Refactorer, HaRe. HaRe is the result of the combined effort of the *Refactoring Functional Programs* project at the University of Kent [7]. HaRe provides refactorings for the full Haskell 98 standard, and is integrated with the two most popular development environments for Haskell programs: Vim and (X)Emacs. HaRe refactorings can be applied to both single- and multi- module programs; HaRe is itself implemented in Haskell, and is built upon the Programatica [11] compiler front-end, and the Strafunski [6] library for generic tree traversal. The HaRe programmers' API provides the user with an abstract syntax tree (AST) together with utility functions (for example, tree traversal and tree transforming functions) to assist with the implementation of refactorings.

Pure functional programs are referentially transparent [14], therefore the opportunities for refactoring are much greater than for imperative programs. The classical example of this, of course, is that in a functional language it is always possible to transform `f x + g x` into `g x + f x` (assuming, of course, that `+` is a binary commutative operator). This is not possible for an imperative language because either `f` or `g` may change the value of the parameter `x` and therefore the result will depend on execution order. This is not, of course, the case in a functional language. In addition to this, in a functional language duplicated occurrences of an expression (within the same scope) will have identical values, and so could be replaced with a shred computation; this is also not the case for imperative programs.

The following sections first introduce the idea of eliminating code within a function that is not needed; this has two flavours: unused code elimination and irrelevant code elimination. A notion of program slicing in Haskell is then outlined. A backwards, static program slicing technique that takes apart tuple-returning functions is defined. Following this the converse of the splitting operation, namely merging is also defined. The paper concludes by looking at possible future developments in program slicing for Haskell.

2 Code Elimination

Dead code elimination [3] is a compiler optimization used to reduce a program size by removing the parts of the program which are not needed. In Haskell dead code contains code that is unreachable by evaluating the `main` function.

This section introduces two flavours of code elimination: dead code elimination and irrelevant code elimination. Dead code elimination is concerned with taking a particular top-level function of interest, and removing any nested declarations within that function that are not needed. Irrelevant code elimination is a generalization of the former. Irrelevant code elimination focuses upon removing nested declarations that are not needed to compute a particular sub-expression within the top-level declaration. Irrelevant code elimination is used to aid the programmer in debugging code.

2.1 Dead Code Elimination

Often programmers write a first version of a program without paying full attention to programming style or design principles [5]. Often, having written a program, the programmer will realise that a different approach would have been much better. Refactoring tools provide software support for modifying the original program into a better written program thus avoiding the expense of starting from scratch.

Programming "rapidly and badly" can cause lots of unnecessary declarations in the program; declarations that are never called and are hence "dead". For example, the code below takes a list of type variables and produces a data structure containing the list of type variables:

```
createApp [var]
 = (Typ (HsTyVar (nameToTypePNT var)))
createApp (v:vars)
 = Typ (HsTyVar (nameToTypePNT (v ++
                                    (concatMap (" -> "++) vars))))
    where
     myConcat :: [ String ] -> String
     myConcat [] = []
     myConcat (x:xs) = (x ++ " -> ") ++ (myConcat xs)
```

The function `createApp` contains one declaration that is not needed: `myConcat`
can be removed, as it is not needed to evaluate the result. Dead code can make a
function look particularly messy and depending on the particular compiler used
may actually consume memory and slow execution. Dead Code Elimination is a
refactoring implemented in HaRe that searches the AST for the definition of a
particular function (supplied as a location in the original code), removing any
declarations contained within that particular function that are not needed. The
examples provided in this section present only a `where` clause; a similar technique
is used to cope with lambda definitions and `let` clauses.

There are two stages to the refactoring. An analysis stage that collects all
the information required to do the modification, and a modification stage that
actually performs the modification on the AST with the information provided by
the analysis stage. Currently the refactoring only removes dead code from within
one function. However, it could easily be expanded to take a whole module of
functions (or, indeed, a set of modules) into consideration.

The main algorithm is as follows:

1. The particular function clause in question is extracted from the AST.
2. The right-hand-side is traversed until the result is found.
3. A list of free identifiers (identifiers that are declared at another point in the
   program) are calculated from the expression. Identifiers that are declared
   outside of the scope of the function clause are removed from the list.
4. There are three main stages to the removal of dead code:
   (a) declarations that are used on the right-hand-side are then also traversed
       to determine free variables contained therein.
   (b) steps 3 - 4 are repeated for each declaration appearing in the list of free
       variables (this checks to see whether some declarations depend on other
       declarations in scope). Nested declarations are also considered.
   (c) the `where` clause is then traversed. Steps 3 - 4 is repeated for each decla-
       ration in the `where` clause. This takes into account nested `where` clauses.
5. For mutually referential declarations the algorithm traverses the AST for
   all declarations to ensure that all free variables in those declarations are
   retained.
6. Any declarations remaining are removed from the AST.

In order to use this tool, firstly the user selects a function from the editor
window. The user then selects *dead code elimination* from the HaRe drop-down

menu. To capture the entire namespace, the whole program is parsed into an AST and token stream. The AST is then traversed using Strafunski to find the particular function clause in question. This traversal is performed by using the location information in the AST; it is possible to traverse into any functions corresponding to the selection region in the editor. Once the function is found the function's `where` clause and right-hand-side is retrieved. The function's right-hand-side is then traversed until the result is reached. For example the result of `createApp` is:

```
Typ (HsTyVar (nameToTypePNT (v ++ (concatMap (" -> "++) vars))))
```

Declarations declared in the scope of `createApp` must be analysed to check whether they are dead; these declarations can appear within `let` clauses and lamdba expressions within the right-hand-side of the function. The refactoring takes into consideration nested declarations; for example: `where`/`let` clauses. Once the returning subexpression is reached, the free and bound names within the subexpression are calculated. The list of free names is then used to remove those declarations residing on the right-hand-side that do not appear within the list of free names. For example:

```
f x =  z + res              f x =  z + res
       where                        where
       res = f (x-1)                res = f (x-1)
       res2 = f (x+1)               z = 46
       y = x + 1
       z = 46
```

The result expression is `z + res` and the only free variables are `z` and `res`; therefore, `y` can be removed from the right-hand-side of `f` as it is not used within `z + res`.

Any mutually referential declarations must be taken care of by ensuring that all free variables in those declarations are retained. For example in the code below it can clearly be observed that the declaration `z` depends on the declaration `y`.

Once the right-hand-side has been modified to remove the declarations that are not used, the `where` clause of the function in question is then analysed. This time the free variables are calculated for each sub-expression within the modified right-hand-side; each member of the `where` clause that appears in the list of free variables is then analysed for its free variables. All the declarations in the `where` clause that are not needed by the right-hand-side of the function in question, and

do not appear in the dependancy graph of any needed declaration, are removed:

```
f x = z + res                f x = z + res
    where                        where
    res = f (x-1)                res = f (x-1)
    res2 = f (x+1)               y = x + 1
    y = x + 1                    z = 46 + y
    z = 46 + y
```

After the AST has been modified, the source code is also modified to mirror the changes of the refactoring.

A popular technique in abstract interpretation [2] is strictness analysis [8]. Dead code elimination is related to strictness analysis in that strictness analysis searches for parts of a program that will always be used. Dead code elimination searches for parts of the program that will never be used. Strictness analysis works by abstracting away from the program so that some dynamic information can be inferred in a static way: inferring that the boolean conditional in an `if` expression is `False`, say, and therefore calculating that the consequence of the `if` is never evaluated. This paper takes a static approach to determining dead code.

2.2 Irrelevant Code Elimination

Irrelevant code elimination is useful for debugging purposes and to some extent is used in algorithmic debugging [13]. In algorithmic debugging the debugging tool asks the user a series of questions about whether a particular sub-expression in the code is generating the correct result or not. As the questions proceed, the particular parts of the program that the debugging tool is asking questions about becomes more clearly focused. It is often the case, however, that the programmer will have some intuition where the bug will lie within the code. Extracting a particular sub-expression that is suspected to cause a bug increases the chances of fixing the error. Parts of a function that are known (or at least assumed) to be correct are temporarily removed so the programmer can concentrate effort on fixing the incorrect sub-expression.

As described above, the Dead Code Elimination technique may be generalized to facilitate debugging. It is possible to select a particular sub-expression of interest and to have the function pruned of declarations that are not needed by that particular sub-expression. The expression on the right-hand-side is replaced with the selected sub-expression. This particular generalization of dead code elimination is not a refactoring, it is in fact a transformation since it changes the semantics. For example, consider:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )

pad :: [ [a] ] -> [ [a] ]
```

```
pad lists = map (pad' (count lists)) lists
     where
         pad' count entry = entry ++ (replicate count (head entry))
```

Suppose the programmer suspects there is a bug in `pad`, specifically, the programmer believes that the bug is in the call `count lists`. Isolating out only the call to `count lists` into a new function would allow the programmer to test that call explicitly, eliminating the parts of `pad` that the programmer believes to be correct:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )

pad2 :: [ [a] ] -> Int
pad2 lists = count lists
```

The programmer can then place a call to `pad2` in the code, test the program, discover that the formal parameter to `count` is in fact incorrect and undo the previous transformation and correct the error:

```
count :: [ [a] ] -> Int
count list = maximum ( map length list )
```

The definition of `pad` remains unchanged.

2.3 Summary

This section has described two flavours of code elimination for Haskell. The first: *dead code elimination* was concerned with looking at the entire result of a function and removing the parts of the function that were not needed to compute the result. This was then generalized further by allowing the user to highlight a particular sub-expression in the result and then remove the parts of the function that are not needed to compute the sub-expression; this process was called *irrelevant code elimination*. Dead code elimination and irrelevant code elimination, in particular, form the basis of a backwards static program slicing tool. The next section expands on the work presented here to introduce the notion of *splitting* and *merging*.

3   Slicing Based Refactorings

This section defines what program slicing means for a functional language. In section 3.1 program slicing is introduced and is then implemented as a *splitting* refactoring in section 3.2. The converse of this is then described in section 3.3.

3.1 Program Slicing

Weiser, in [16], introduces a program slice S as a *reduced executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P*. This process is driven from a *slicing criterion*, usually a variable representing the line number and expression of interest, which is used to represent the point in the code whose impact is to be observed with respect to the entire program. Weiser introduced the concept that is now known as a *backwards, static* slicing method. A backwards slice consists of all parts of a program that have an effect on the criterion in question. Another form of program slicing is a forwards slice [15]. Starting with the slicing criterion, or the program point of interest, a forwards slice is all parts of the program that the criterion will affect.

Orthogonal to the choice between forward and backward slicing is the distinction between *static* and *dynamic* slicing: static program slicing means that all possible computations of a program are considered and dynamic program slicing means inferring names with particular values, giving a very specific computation of a program. Often, the slicing criterion is a sub-set of program variables —the program slice becoming the parts of the program related to the variables.

The obvious analogue to this within the functional paradigm is a subset of the components of a structured result, for example, the fields of a tuple in Haskell. Selecting the components of a result and looking at the computation corresponds to backwards slicing, whereas selecting the subset of the arguments of a function, and seeing what can be computed on the basis of them alone, corresponds to a forwards slice.

Hitherto, there has been little work on program slicing for functional languages. Ochoa et al.[9] introduced a dynamic slicing technique for a lazy logic language. The Haskell debugger, Hat [1], also includes a form of program slicer. However, at this time there is no standalone program slicing tool available for Haskell and therefore we attempt to define a backwards static slicer for Haskell.

This section introduces the notion of program splitting and merging. A number of issues with performing splitting and merging are also given.

3.2 Splitting

A function may return a structured value, for example, a tuple. The particular examples presented in this section use only pairs, however the technique can easily work over tuples of any order.

Splitting works by selecting an element of the tuple and then working out everything needed to calculate that element. The calculated dependencies are then simply extracted and isolated from the rest of the function.

Splitting is mostly used for debugging purposes. However splitting may also be used to extract functionality out of the function so that it can be extended or re-used. The user passes a parameter to the splitter the elements of the result of the function in question which are to be extracted.

Consider the function `parseMessage` below. `parseMessage` takes a `String` of messages each separated by the `&` character. `parseMessage` removes the initial

message and returns the next message as the first element of the result and the remainder of the message as the second element:

```
type MessageList = String
type Message = String
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
       where
         ys = dropWhile (/= '&') xs
```

As an example of the usage of `parseMessage` consider:

```
> parseMessage "goodbye&hello&world"
> ("hello","&world")
```

The splitter works through each function clause in turn, extracting the elements of the function clauses' result into separate definitions. It is worth noting that the results of these refactorings are themselves ready for a further refactoring, namely replacing `ys` with its definition (removing the `where` clause altogether). It is often the case that further refactorings such as this are often performed. The first pattern clause of `parseMessage` is essentially trivial. Therefore the value`[]` is extracted for both elements of the result and two new functions are then created:

```
parseMessage1 :: MessageList -> Message
parseMessage1 [] = []

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
```

The splitter appends an index to the end of the names of the new functions, if the new names conflict with any other identifier in scope then the splitter chooses a new distinct name. This is simply done by incrementing the index until the name no longer conflicts with another identifier in scope.

The next function clause's result is then analysed. Irrelevant code elimination is then performed for each element in the resulting tuple and the result of the code elimination is placed into new function clauses for `parseMessage1` and `parseMessage2`. `ys` is required by both elements of the result of `parseMessage` so it is retained; the new function clauses are then added to the definitions of `parseMessage1` and `parseMessage2` within the AST.

```
parseMessage1 xs = takeWhile (/= '&') (tail ys)
       where
             ys = dropWhile (/= '&') xs

parseMessage2 xs = dropWhile (/= '&') (tail ys)
```

```
        where
              ys = dropWhile (/= '&') xs
```

This gives the new definitions of `parseMessage1` and `parseMessage2`. The source code is then modified to reflect the changes within the AST.

3.3 Merging

Merging is the process of taking a number of functions and unifying them together into one tuple-returning function. The process described here only merges two functions; obviously the functionality can be easily extended however, to merge together any number of functions.

Merging works by unifying all the code from the selected functions into a new tuple-returning function. Duplicate parts of the function are also removed. Unlike when doing splitting, where names are generated automatically, the user must specify a name for the merged function.

Merging is mostly used to reuse code and improve code efficiency. For example, merging `take` and `drop` into `splitAt` results in only one recursive call instead of two. Merging functions together has the possibility to introduce further code sharing.

The remainder of this section will focus on merging `parseMessage1` and `parseMessage2`.

```
parseMessage1 :: MessageList -> Message
parseMessage1 [] = []
parseMessage1 xs = takeWhile (/= '&') (tail ys)
        where
              ys = dropWhile (/= '&') xs

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
parseMessage2 xs = dropWhile (/= '&') (tail ys)
        where
              ys = dropWhile (/= '&') xs
```

Firstly, the merge refactoring checks to see whether the pattern sets of `parseMessage1` and `parseMessage2` are the same. If they are not then the merge refactoring terminates with an error to the user. The merge refactoring also checks to see whether the types of the arguments to `parseMessage1` and `parseMessage2` are the same. If the pattern sets are not the same then the merge refactoring cannot correctly unify the patterns; if they cannot be unified the merge refactoring terminates with an error message to the user.

Each function clause in question is merged together. The first clause of `parseMessage1` and `parseMessage2` both have the same result value. Merging those clauses together is trivial:

```
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
```

The second clauses of `parseMessage1` and `parseMessage2` are now considered. The results are unified together and placed into a new function clause:

```
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
        where
          ys = dropWhile (/= '&') xs
          ys = dropWhile (/= '&') xs
```

The duplicate declaration of `ys` within the `where` clause is removed.

3.4   Design and Implementation Issues

Sometimes when splitting is used patterns that are the result of recursive calls become redundant. All patterns that are not needed in the result are replaced with wildcards so that the namespace is kept clean.

One element of a tuple may depend upon another element. For instance, it is possible for one element to come from the first part of a recursive call, but for it to be returned in the second element in the main body. In very unusual circumstances the following can occur:

```
f :: Int -> (Char, (Char, Int))
f 10 = ('a', (g 'f' 1, 2))
f n = (x, (g x 1, 2))
  where
  (_, (x, 2)) = f (n+1)

g :: Char -> Int -> Char
g x y = chr (ord x + 1)
```

The first element of the result of `f` comes from the second element in its recursive call. If one element depends on another, both of these elements must be extracted from the function, otherwise an error occurs.

Before merging, it is necessary that both functions have the same sets of patterns. If the first function has more pattern clauses than the second for example, then the merge refactoring cannot determine what to place on the right-hand-side when patterns from the first function are not matched by patterns in the second. An error message is presented to the user if the pattern sets are not the same. An additional refactoring was introduced to allow the user to build particular pattern clauses by instantiating general patterns with values of the same type. Consider, for example:

```
f1 0 l = take 42 l
f1 n l = take n l

f2 n l = drop n l
```

Both `f1` and `f2` have general cases defined, however, `f1` has an additional base case defined. This is problematic because the merge refactoring cannot infer what the intention is of the user during the merge:

```
merged 0 l = (take 42 l, undefined)
merged n l = (take n l, drop n l)
```

When merging functions, it is also necessary that the arguments to the functions have the same type so that they can be successfully merged. If the functions have different argument types then the merge refactoring returns an error. It is possible that the functions have different return types as this will be captured in a tuple.

Merging and splitting monadic functions is a difficult area, especially if the monad in question is a state monad, for example, the `IO` monad. Merging two `IO` monadic values is problematic because the merge refactoring cannot infer the correct order of sequencing. Side effects also effect splitting in a similar way, for example, an element of the tuple return value may depend upon some data written to a file. It is very difficult for the splitter to determine which parts of the monad can have a potential effect as all expressions have the potential to alter the state. Merging and splitting refactorings on monads will be the subject of a future paper.

4 Conclusions and Future Work

This paper presented a number of refactorings for HaRe. Firstly, a technique was defined to eliminate dead code from Haskell functions and then generalized further to remove irrelevant code. A backwards, static program slicer for Haskell was then described as splitting tuple-returning functions; its converse, namely merging, was also described.

In the future it is planned that dead code elimination will be expanded to take a whole program into account and not just a selected function. Dead code elimination could remove the parts of the program that are not directly related to the `main` function, which would allow the process to be extended to the whole of a large Haskell project.

It is also planned to modify the splitter to analyze the whole scope of a program rather than, simply, tuple-returning functions in the scope of a particular function of interest. It is also planned to modify the splitter so that tuple-returning functions that are called from outside a function's scope to be extracted.

A further paper will detail work on refactoring with monads. It is intended for the work on slicing to be extended further to investigate backwards, dynamic slicing and forwards slicing (both static and dynamic). There are many more exciting possibilities to analyze for the refactoring of Haskell code; and it is hoped to continue work in these directions.

5 Acknowledgments

Thanks to Dave Harrison of Northumbria University for his editorial advice.

A Source Code

All the code that has been described in this paper can be downloaded using *darcs* [12] with the following command:

```
darcs get http://www.cs.kent.ac.uk/projects/refactor-fp/HaRe_Project/
```

References

1. Olaf Chitil. Source-based trace exploration. In *Draft Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL 2004*, pages 239–244. Technical Report 0408, University of Kiel, September 2004.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
3. Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.
4. John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.
5. Fred P. Brooks Jr. The Mythical Man-Month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM Press.
6. Ralf Lammel and Joost Visser. A strafunski application letter. In *Proc. of PADL'03*, January 2003.
7. Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer: HaRe, and its API. In John Boyland and Grel Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science, http://www.sciencedirect.com/science/journal/15710661.
8. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*, pages 269–281, London, UK, 1980. Springer-Verlag.
9. Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134, New York, NY, USA, 2004. ACM Press.
10. William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.
11. PacSoft. Programatica: Integrating programming, properties and validation. www.cse.ogi.edu/PacSoft/projects//, 2005.
12. David Roundy. Darcs: distributed version management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM Press.

13. Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 157–166, New York, NY, USA, 2006. ACM Press.
14. Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, 1990.
15. F. Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.
16. Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, 1979.