A Collection of Ideas for Haskell Transformation

Chris Brown

June 22, 2006

Contents

Chapter 1

A Haskell Program Slicer

This section explores the idea of a program slicer, which has been developed, to some extent, for the Haskell Refactorer, HaRe. In this section I aim to discuss the issues surrounding program slicing in the context of a functional language. I also aim to propose the areas that I wish to pursue in slicing Haskell programs.

Weiser introduced the concept of program slicing to the world in the late seventies [7, 5, 6]. Program slicing is essentially a family of techniques for isolating parts of a program which depend on or are depended upon a specific criterion, known as a *slicing criterion*. In most of the mainstream research on program slicing, the functional programming paradigm has been largely neglected. Apart from the Programatica project [3], and the HaSlicer [4], program slicing remains an uncharted territory for Haskell. Even in the mentioned Programatica project, the slicer remains undocumented, and it is unclear exactly what it does. The HaSlicer on the other hand, introduces a means to begin formalising slicing algorithms for functional programs using a *Functional Dependence Graph* (an FDG). The HaSlicer, however is more of a proof-of-concept analysis tool, and does not actually produce program slices, but instead, graphs that show the sliced FDG based on some criterion.

1.1 Functional Program Slicing

Weiser, in [6] introduces a program slice S as *reduced executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P.* This process is driven from a *slicing criterion*, usually a line number and a variable name. This 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. Current formalizations of slicing techniques are all based on some form of abstract, graph-based representation of the program under scrutiny, from which dependency relations between the entities it manipulates can be identified and extracted.

Mainstream research on program slicing all tend towards imperative languages and, therefore it is orientated towards variable assignment, program statements, execution order and mutable state. A program slicer for a functional program must take a different perspective. Functions, instead of program statements are the basic building blocks of a functional program. Functional composition replaces program statements, and execution order is based on the precedence of functional operations. Due to referential transparency, a functional program has no concept of variable assignment or mutable state, unless dealing with particular kinds of monadic effects.

1.2 The Program Slicer

I have produced a *backwards static* program slicer for HaRe. The slicer works in two modes. First a particular sub-expression of interest is highlighted, and a backwards slice is computed by calculating a variable dependence graph of the free variables in the expression. Secondly, a function returning a tuple can be split into multiple definitions, each encapsulating the functionality of each tuple element.

However, there is no formalization of the slicing algorithm and no justification of its correctness.

- I propose to fully formalize the slicer. I would like to see how constructing a formalization of the algorithm lets one see how the slicer fits into other methods of program slicing. Formalizing the slicer will allow potential holes in the design to be reconsidered carefully. I also believe that this is an opportunity to produce some very interesting and novel research in the area of program slicing. How does slicing functional programs relate with slicing imperative programs? Can the slicer be extended to work with monads? Can the slicer be extended to work dynamically (i.e. some of the input is known beforehand)

- I would like to construct a proof of correctness for the slicer. After a formalization it would be possible to reason that the slicer does, in fact, produce a backwards slice of a sub expression.

- Incorporating symbolic evaluation, it would be possible to add a symbolic evaluator to the slicer. Symbolically evaluating particular arguments to functions, or names within an expression could have the same effect as producing a *dynamic* program slicer. I would like to see the outcome of this.

- Finally, the slicer could be extended to work as a stand-a-lone tool for Haskell 98, using the Programatica front-end and Strafunski libraries. At the moment it only works under the scope of a selected function, and possibly to make it more research based, it would need to be extended to work under the scope of a whole program.

Chapter 2

A High-level Language for Refactorings

The number of possible refactorings to implement seems endless, no tool developer will ever be able to implement refactorings for every programmer's needs. There are also a number of complex refactorings such as memoisation and deforestation, which are not a high priority on the list of refactorings to be implemented. HaRe provides a number of *core* refactorings, but even these are not sufficient in satisfying every possibility. Providing one with the ability to compose their own refactorings together equally does not achieve the same affect as providing one with the ability to define one's own refactorings. Equally unsatisfactory is the lack of user-definable refactorings. For tool providers such as ourselves, because we are given the never ending task of implementing new refactorings, hoping each one is of interest to our market. Also for the users, becuase they are forced either to wait for some release, hoping to find their desired refactoring there, or to implement their own refactorings using the HaRe API. However, the latter option is probably not ideal to most users, as investing a lot of time and effort into developing a new refactoring is not worth it, especially if they have a tight schedule to develop a project.

2.1 Composing refactorings

A solution out of the problem is a provision to allow users to compose their own refactorings together using some kind of higher order language, that makes it easy to describe compositions. To make it a working *tool* that most programmers could use in their day-to-day activities, ideally it would need to have:

- **Declaritive composition**: users should only need to specify which refactorings are to be composed, along with their dependencies. The composition of the specification should happen automatically, so that it does not disrupt the user's activities. There should be no additional programming in composing a refactoring.

- **Program-independant composition**: the composition must be possible without knowing the program(s) to which the composition will work on. This will mean the user will have to supply a set of pre and postconditions which each refactoring in the composition must adhere to.
- **Universally applicable composition**: Is it possible to create a higher-level language that can be used to compose refactorings that work over any domain? Does it have to be specific for HaRe refactorings, or could it also be used for, say, the Erlang refactorer?

Composing refactorings together offers several advantages. Just as complex programs are built from composing functions together, complex refactorings can also be built by composing refactorings together.

What primitives would a higher-order language need that could compose refactorings together?

- **Conditions**: a method for specifying pre conditions and post conditions. A pre condition to a refactoring in a chain of refactorings in a composition, could be that the previous refactoring has transformed the program into a suitable state. Post conditions could possibly include proving the program correct, afterall, a refactoring should be changing the behaviour of the program. Is it possible to automatically prove refactorings correct?
- **Disjunction**: ORing refactorings together, if one fails an alternative can be chosen in a chain of refactorings.
- **Conjunction**: ANDing refactorings would mean that if one refactoring would fail then the whole process would stop. We also want to make sure that *each* of the refactorings in a sequence are applied - the first *and* the second *and* the others.
- **Enabling refactorings**: The composition can infer that earlier refactorings can set up the preconditions for later ones. This is particularly useful when certain preconditions cannot be evaluated by static program analysis.
- **Chaining Undo**: In a conjunction of refactorings, we want to include the possibility that something might go wrong. An undo function to roll back to before the composition was applied is necessary, as is an undo to roll back through each step in the composition.

There has been some previous work done on composing refactorings together. Kniesel [1] describes in his paper, a method for a static composition of refactorings. In his paper he introduces a *refactoring editor* that essentially allows one to create, compose and edit refactorings. Composing refactorings is also proposed in [2], but it is only flagged for future work.

Chapter 3

Data oriented Refactorings

This chapter presents a plethora of data oriented refactorings that could be implemented for HaRe. Data refactorings affect the representation of data within the program in some way. Data refactorings will indirectly affect all functions which involve the type(s) that are modified by the refactoring. All data refactorings should be module aware. A number of new data oriented refactorings follow, each refactoring includes a description of what that refactoring does.

3.1 Add or remove field names in a `data` type

A constructor definition in a `data` definition labels to the fields of the constructor. The fields are added in the record syntax (`C { ... }`). Constructors using field names may be freely mixed in a data type with constructors that do not use them. Labels are simply a shorthand for operations, or functions, that use an underlying positional constructor. For example, the declaration:

```
data Event = Key {char :: Char, isDown :: Bool}
           | Button {point :: Point, isLeft :: Bool, isDown :: Bool}
```

Defines a type and a constructor identical to the following:

```
data Event = Key Char Bool
           | Button Point Bool Bool
```

The purpose of this refactoring is to allow fields names to be added or removed from a data type easily. If the field names are being removed, then selector functions need to be created to take the place of the field labels.

3.2 Change implementation of an ADT

In the most general sense, this will require that a semantically equivalent to be introduced. One might implement sets by lists, or other means. A more specific example would be to perform some *memoisation* over the values within a data type. Perhaps introducing a field, say, for a tree structure that would be used to store the depth of the tree. This process could be done automatically.

3.3 Name a type using `type`

Identify uses of a type in a particular way by making them instances of a type synonym. Applying this refactoring has no semantic effect on the program. Type synonyms cannot be used for `instance` declarations.

The problem with this refactoring is finding the types that could be replaced by a type synonym. Consider the user defined type synonyms:

```
type Name = String
type Surname = String
```

The refactoring cannot infer which type to use when replacing the `String` type. However, if a more unique structure is used, such as:

```
type DB = [((String, Int, String), Bool)]
```

Then type signatures of the following structure:

```
f1 ::  [((String, Int, String), Bool)] -> Bool
```

Can be replaced by:

```
f1 :: DB -> Bool
```

It would also be possible to do the converse where a type synonym in a type signature is unfolded, leaving the original type structure in its place.

3.4 Add or remove discriminator functions for a `data` type

A discriminator function decides which part of a sum a value belongs to. In particular, a discriminator function also decides which part of a data type a particular value belongs to. A traditional naming scheme for discriminators follows the form `isBlah` where `Blah` is the name of the constructor under discrimination.

Taking a data type:

```
data Event = Key Char Bool
           | Button Point Bool Bool
```

Typical discriminator functions would be:

```
isKey :: Event -> Bool
isKey (Key _ _) = True
isKey _ = False
```

Adding discrimators to a program would essentially be trivial, simply creating functions like the above for each constructor in a selected data type. Removing discriminators, on the other hand, would require replacing all calls to the discriminator to a dummy function, perhaps returning an error message. Removing discriminators could be a useful prerequisite to removing a data type from a module, say.

3.5 Type change: convert definitions that work over a particular type to work over a different type

This refactoring would require all definitions over the type in question to be modified. This refactoring would fall under the category of type-aware. A more general case would be to allow one to specify one's own type changes. For example a program may need to modified to work over sets (represented by a data type) as opposed to lists. Some examples could be:

- Convert `Maybe` to `List` or to `Either`;
- Convert `Bool` to `Maybe`;
- Convert between tuples and (one constructor) algebraic types;
- Convert between tuples and (homogenous) lists.

Bibliography

[1] G. Kniesel and H. Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[2] N. Moha, S. Bouden, and Y.-G. Guéhéneuc. Correction of high-level design defects with refactorings. In S. Demeyer, S. Ducasse, Y.-G. Guéhéneuc, K. Mens, and R. Wuyts, editors, *Proceedings of the* 7^{th} *ECOOP Workshop on Object-Oriented Reengineering*, July 2006.

[3] PacSoft. Programatica: Integrating programming, properties and validation. www.cse.ogi.edu/PacSoft/projects//, 2005.

[4] N. Rodrigues and L. S. Barbosa. Component identification through program slicing. *Proc. Formal Aspects of component software (FACS'05).*, 2005.

[5] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[6] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[7] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, 1979.