Refactoring In Scheme Using Static Analysis

Jens Nicolay

Scheme is a small but elegant and powerful programming language with clean syntax. It allows for both imperative and functional programming and is able to express several different programming paradigms. As a consequence, Scheme has influenced the design of many other more widely-used and industrially-relevant languages, while also making the language especially suited for experimentation. Even with all this power, expressivity and influence, no refactoring catalog or well-known refactoring tools exist for Scheme. There are a couple of reasons why this is might be the case. Firstly, most refactorings are based on static analysis of code, and static analysis is far from trivial to perform in dynamic languages such as Scheme. Also, Scheme isn't widely used outside of academia, although this does not change the fact that it is very well suited as a research language, especially in the context of the growing interest in dynamic languages that we see today. We argue that refactoring in Scheme should enjoy the same status as the language itself: it should deepen our general understanding of techniques for, and implementations of, program analysis and transformation, with influences far beyond the Scheme language. Our aim is twofold. First of all we want to compile a refactoring catalog for Scheme, containing the exact specifications of general refactorings like RENAME, ADD PARAMETER, and so on, expressed as program transformations guarded by pre- and postconditions. We also want to discover refactorings that are not readily identified as general refactorings and see how these might carry over into other languages. Our second goal is to design a specification language that allows us to express Scheme refactorings. For this specification language we again choose Scheme, but with built-in backtracking for convenience and with a library of primitives that allow reasoning over Scheme programs. This reasoning is based on the results of a sufficiently precise, powerful and fast static analysis, on which several layers of primitives are layered so that the right level of abstraction can be selected by the designer of refactorings. Th analysis approximates value flow, control flow and interprocedural dependencies. Note that our two aims go hand in hand. In order to write down refactoring specifications we need a language. At the same time this specification language will determine what the refactoring specifications will look like. To validate our work, we select existing refactoring scenarios and see how our approach deals with them, assessing strengths and investigating weaknesses. In order to facilitate experimentation and as a way to make our work publicly available, we have also build an Eclipse plugin aimed at programming

in Scheme and containing several refactorings. The provided Scheme editor also detect certain patterns in the source code in order to provide feedback during development. The plugin also allows a developer to perform program analysis and transformation using a meta-programming approach, which is useful for prototyping refactorings.