Reading, Writing and Relations
Towards Extensional Semantics for Effect Analyses

Nick Benton[1], Andrew Kennedy[1], Martin Hofmann[2], and Lennart Beringer[2]

[1] Microsoft Research, Cambridge
[2] Ludwig-Maximilians-Universität, München

Abstract. We give an elementary semantics to an effect system, tracking read and write effects by using relations over a standard extensional semantics for the original language. The semantics establishes the soundness of both the analysis and its use in effect-based program transformations.

1 Introduction

Many analyses and logics for imperative programs are concerned with establishing whether particular mutable variables (or references or heap cells or regions) may be read or written by a phrase. For example, the equivalence of while-programs

```
C ; if B then C' else C''  =  if B then (C;C') else (C;C'')
```

is valid when `B` does not read any variable which `C` might write. Hoare-style programming logics often have rules with side-conditions on possibly-read and possibly-written variable sets, and reasoning about concurrent processes is dramatically simplified if one can establish that none of them may write a variable which another may read.[1]

Effect systems, first introduced by Gifford and Lucassen [8,11], are static analyses that compute upper bounds on the possible side-effects of computations. The literature contains many effect systems that analyse which storage cells may be read and which storage cells may be written (as well as many other properties), but no truly satisfactory account of the semantics of this information, or of the uses to which it may be put. Note that because effect systems *over*estimate the possible side-effects of expressions, the information they capture is of the form that particular variables will definitely *not* be read or will definitely *not* be written. But what does that mean?

Thinking operationally, it may seem entirely obvious what is meant by saying that a variable X will not be read (written) by a command C, viz. no execution trace of C contains a read (resp. write) operation to X. But, as we have argued before [3,6,4], such intensional interpretations of program properties are over-restrictive, cannot be interpreted in a standard semantics, do not behave well with respect to program equivalence or contextual reasoning and are hard to

[1] Though here we restrict attention, in an essential manner, to sequential programs.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 114–130, 2006.

maintain during transformations. Thus we seek extensional properties that are more liberal than the intensional ones yet still validate the transformations or reasoning principles we wish to apply.

In the case of not writing a variable, a naive extensional interpretation seems clear: a command C *does not observably write the variable* X if it leaves the value of X unchanged:

$$\forall S, S'.\ C, S \Downarrow S' \implies S'(X) = S(X)$$

Note that this definition places no constraint on diverging executions or the value of X at intermediate states. Operationally, C may read and write X many times, so long as it always restores the original value before terminating. Furthermore, the definition is clearly closed under behavioural equivalence. If we have no non-termination and just two integer variables, X and Y, and the denotation of C is $[\![C]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ then our simple-minded definition of what it means for C not to write X can be expressed denotationally as

$$\exists f_2 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (X, f_2(X, Y))$$

which is the same as saying $[\![C]\!] = \langle \pi_1, f_2 \rangle$.

The property of *neither reading nor writing* X, i.e. of being *observationally pure in* X is also not hard to formalize extensionally:

$$\forall S, S', n.\ C, S \Downarrow S' \iff C, S[X \mapsto n] \Downarrow S'[X \mapsto n]$$

Alternatively $\exists f_2 : \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (X, f_2(Y))$, which is the same as saying $[\![C]\!] = 1 \times f_2$.

The property of *not observably reading* X is rather more subtle, since X may, or may not, be written. We want to say that the final values of all the other variables are independent of the initial value of X, but the final value of X itself is either a function of the other variables or is the initial value of X:

$$\exists f_1 : \mathbb{Z} \to \mathbb{B}, f_2, f_3 : \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (f_1(Y) \supset X \mid f_2(Y), f_3(Y))$$

This is clearly a more complex property than the others. Another way to think of it is that the final values of the variables other than X are functions of the initial values of those variables and that for each value of those other variables, the (curried) function mapping the initial value of X to its final value is either constant or the identity. The tricky nature of the 'does not read' property also shows up if one tries to define a family of monads in a synthetic, rather than an analytic fashion (as in Tolmach's work [20]): neither reading nor writing corresponds to the identity monad; not writing corresponds to the reader (environment) monad; but there is no *simple* definition of a 'writer' monad.

Our basic approach to the soundness of static analyses and optimizing transformations is to interpret the program properties (which may be expressed as points in an abstract domain, or as non-standard types) as binary relations over a standard, non-instrumented (operational or denotational) semantics of