Flemming Nielson:
Annotated Type and Effect Systems

Program analysis offers static techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically during computation; this may be used to validate program transformations or to generate more efficient code. The flow based approach includes the traditional data flow analysis techniques for mainly imperative languages, but also the control flow analysis techniques developed for functional and object oriented languages. The model based approach includes the parameterised denotational semantics techniques developed for functional and imperative languages, but also more generally the use of mathematical modelling in the abstract interpretation of imperative, functional, concurrent and logic languages. The inference based approach includes general logical techniques touching upon program verification techniques, but also annotated type and effect systems developed for functional, imperative and concurrent languages; it is this latter and rather recent approach that we now consider.

Annotated type and effect systems require that a typed programming language be given [6, 1]. The type system is then modified with additional annotations (called effects when they take the form of sets of properties) that expose further intensional or extensional properties of the semantics of the programs. The literature has seen a great variation in the annotations and effects used. Example effects are: collecting the set of procedures or functions called, collecting the set of storage cells written or read during execution [5], collecting the regions in which evaluation takes place [7]. Other classes of annotations are more ambitious in trying to identify the causality among various operations: that input takes place before output, that communication satisfies protocols as expressed by terms of a process algebra [4].

The methodology of annotated type and effect systems consists of: (i) devising a semantics for the typed programming language, (ii) expressing a program analysis by means of an annotated type or effect system, (iii) showing the semantic correctness of the analysis, (iv) developing an inference algorithm and proving it syntactically sound and complete, and (v) utilising the information for applications like program transformations or improved code generation. Each of these phases have their own challenges and open problems.

(i) Semantics is a rather well understood area but to serve as the foundation for the validation of program analyses one frequently needs an instrumented semantics [2] which is more precise about low-level machine detail (say concerning pipe-lining or the number and nature of registers) than is usual. Hence one must be cautious about claims that an analysis has been proved correct with respect to “the” semantics of a programming language.

(ii) Much research concerns how to incorporate the flow based considerations of
context dependent analysis: polyvariance (as opposed to monovariance), (poly-
)...k-cfa, “polymorphic” splitting etc.; similarly for the model based considerations
of relational (as opposed to independent attribute) analyses and more advanced
notions of designing combined property spaces. Although the simple instances
of monovariant \( \theta \)-cfa analyses in independent attribute form can be expressed
as annotated type and effect systems, it is still unclear how to achieve the more
advanced possibilities. Current research suggests that the use of polymorphic
recursion (for the annotations, not the types) may be essential and still decid-
able [7], but the interplay between the use of polymorphism and sub-typing and
sub-effecting is still not fully understood [5] and is an important area of further
research.

(iii) The techniques needed for establishing semantic soundness are mostly
standard. For operational semantics the statement of correctness may be a
subject reduction result and the method of proof may benefit from the use of
co-induction; when the use of denotational semantics is possible one may benefit
from the use of Kripke-logical relations.

(iv) Algorithmic techniques often involve the generation of constraint systems
in a program independent representation. Sometimes efficient techniques devel-
oped for flow based analyses can be used to solve the constraint problems; in
other cases the problems take the form of semi-unification problems and then
even decidability becomes an issue [7]. An important area of further research
is how to identify those features of the annotated type and effect systems that
lead to algorithmic intractability.

(v) Exploitation is a rather open ended area although it would seem that the
integration of program analyses and program transformations into an inference
based formulation is quite promising [8]. This may be one way of overcoming
a shortcoming of the denotational approach where program points are usually
not part of the semantics and hence some machinery is needed in order to make
them available as an interface to the exploitation phases [3].

In summary, program analysis by means of annotated type and effect systems
seems a promising area. The main strength lies in the ability to interact with
the user: clarifying what the analysis is about (and when it may fail to be of
any help) and in propagating the results back to the user in an understandable
way (which is not always possible for the flow-based approaches working on
intermediate representations). The main areas of further research concern the
expressiveness of the inference based specifications and the complexity and de-
cidability of the algorithmic realisations.

References

Springer Lecture Notes in Computer Science 582, 1992.


