PROPOSAL FOR THESIS CORRECTIONS

The starting point for addressing the reviewers concerns is observing that in many practical programming languages, the value of an expression depends on the type derivation (Section 1). Often, the choice is hidden behind a mechanism that selects one type as primary[1].

The same is the case for the coeffect calculi presented in the thesis, but our approach makes the dependence on the typing explicit and we resolve it by defining the semantics as a function of a typing derivation[2]. this approach does not diverge from, but rather extends Moggi's work [3] (Section 2).

In the following two sections, we argue that specifying program semantics based on a type derivation is not unusual in practice (Section 1) and theory (Section 2)[3]. Finally, Section 4 concludes with a concrete correction proposal.

1 TYPE-DIRECTED SEMANTICS IN PRACTICE

The aim of the thesis is to provide foundations for extending practical functional-first programming languages such as F# [5] and Scala [4] with support for context-aware programming, so we consider two F# examples. In both, the meaning of a term depends on the typing:

```
let nextTwo rnd = (rnd.Next(), rnd.Next())
let add a b = a + b
```

The two examples demonstrate two ways of resolving ambiguities in typing derivations and corresponding semantics:

- The first example illustrates an ordinary .NET method invocation. In this case, the meaning of rnd.Next() depends on the type of rnd. This could be any class with the Next method. The F# compiler can emit .NET bytecode only if it knows the concrete type and so the developer has to resolve the ambiguity through a *type annotation*.
- Operators are treated differently. The meaning of + depends on the types of the operands[4], but F# can compile it without additional information thanks to a *defaulting mechanism*, which chooses int if there are no other constraints. If the program contains add "hi" "there", F# will use string concatenation instead of the default (integer addition).

In both examples, the meaning of the term depends on a typing derivation, but the compiler includes a mechanism for finding a unique typing.

2 TYPE-DIRECTED SEMANTICS IN THEORY

In the thesis, we give the semantics of terms based on a typing derivation (both for concrete languages and for coeffect calculi). Using the notation used in the thesis, the semantics of a (plain) let-binding is written as:

$$\begin{aligned}&[\![x:\tau \vdash \mathtt{let}\ x_1 = e_2\ \mathtt{in}\ e_2 : \tau_2]\!] = \\ &\quad [\![x_1:\tau_1 \vdash e_2 : \tau_2]\!] \circ [\![x:\tau \vdash e_1 : \tau_1]\!]\end{aligned}$$

1 An F# example is discussed in Section 1 and Haskell example is discussed in Section 3.
2 The thesis does not explicitly state this and it will be clarified in the final version.
3 This background will be added in the final version of the thesis.
4 F# differs from OCaml, which uses + for integers, +. for floats and ˆ for string concatenation.

The reader might incorrectly see the equation as defining a function from typing *judgements*, as opposed to typing *derivations*. The correct reading is the latter. In fact, the same approach has been used by Moggi in his seminal paper on monads, who writes [3, p7] (emphasis mine):

> An interpretation $[\![-]\!]$ of the language in a category C is parametric in an interpretation of the symbols in the signature and is defined by *induction on the derivation* of well-formedness for (types,) terms and equations.

The notation used by Moggi is unambiguous (and we intend to adopt the same clear notation in the revised version of the thesis):

$$\frac{[\![x:\tau \vdash e_1 : \tau_1]\!] = g_1 \qquad [\![x_1:\tau_1 \vdash e_2 : \tau_2]\!] = g_2}{[\![x:\tau \vdash \texttt{let}\ x_1 = e_2\ \texttt{in}\ e_2 : \tau_2]\!] = g_2 \circ g_1}$$

In systems that have a unique typing (including Moggi's), the fact that the semantics is defined by induction *over a typing derivation* means merely that we only define semantics for well-typed programs. Languages with non-syntax-driven rules (and no unique typing derivation) have two options:

- Coherent semantics. For certain systems [1], the semantics of a term is the same regardless of a typing derivation. In that case, we can define the semantics over a typing derivation, but show that all derviations yield the same semantics.
- Type-directed semantics. Systems where the meaning of a term *does depend* on a typing derivation (cf. Section 1) do not have the coherence property and thus require *type-directed* semantics. This is the case for example, for the flat coeffect system for implicit parameters (Section 3).

Contextual dependencies can often be satisfied in multiple ways[5]. By capturing the contextual properties in the type-and-coeffect system, we are making the existing run-time ambiguities *more explicit*. Thus the proposal is to focus on systems with *coeffect-directed semantics* rather than modifying the calculi that inspired the work in the thesis in order to force coherence.

3 RESOLVING AMBIGUITY

The case where the semantics of a context-aware program depends on the typing derivation is best demonstrated using the system for implicit parameters (Section 3.2.1 of the thesis). Consider the following example:

```
let f = ( let ?x = 10 in λv → ?x) in
let ?x = 15 in f 0
```

Here, a function value $\lambda v \to ?x$ is created in a context where the implicit parameter ?x has a value 10. The function is then called in a context where ?x has a value 15. Is the result of the expression 10 or 15? The choice of the run-time semantics needs to be reflected in the type system.

RESOLUTION IN HASKELL. In Haskell, the implicit parameters from the *declaration-site* are preferred and so the result is 10. Correspondingly, the type system only permits one typing – with the parameter ?x required on the declaration-side (allowing no rebinding at the call-site). However, if the

5 For example, see the discussion about distributed programming in Section 3.2.2 of the thesis.

function value was defined in a context not containing ?x, the value from the call-site would be used. Thus, Haskell uses a *defaulting mechanism* for implicit parameters, similar to that of F# operators discussed in Section 1.

RESOLUTION FOR EXCEPTIONS. In case of implicit parameters, the choice is between *lexical* and *dynamic* scoping. The same choice appears for other language constructs, such as exception handling. Exception handlers in most modern languages are dynamically scoped, but Beta and original version of Smalltalk support lexical scoping [2]. A language that supports both dynamic and lexical exception handlers would expose the same problem.

RESOLUTION IN COEFFECT SYSTEMS. Coeffect systems presented in the thesis explore the most general approach. For example, the calculi for implicit parameters permits both lexical and dynamic scoping without a built-in ad-hoc defaulting mechanism.

Our design choice means that semantics of a term depends on the typing. In other words, the inherent choice in the semantics is reflected in the fact that a term can have multiple types. We argue that a *coeffect-directed semantics* (as used in the thesis) provides more insight into the problem of context-aware programming as it lets us explore a range of design possibilities.

4 CORRECTION PROPOSAL

In the preceding sections, we argued that *coeffect-directed semantics* is the right approach for discussing context-aware programming languages. This clarifies the confusion arising from the fact that coeffect systems (like implicit parameters and others) do not satisfy categorical *coherence*. This section outlines four specific improvements to address the reviewers concerns.

4.1 TYPE ERASURE. The coeffect-directed semantics in the thesis is presented in the style of a categorical semantics (as it was, indeed, inspired by categorical structures). However, it does not follow the full rigour of category theory and so it is better seen as *translational semantics* or as *type erasure*. Given a typing derivation we produce a new term that unambiguously defines the meaning of the original term (Chapter 3 uses direct denotational style; Chapters 4 and 5 use a common abstract structure).

The main value of the semantics is guiding the definition of the type system, because the structure of the coeffect annotations in the typing rules follows from the structure of this coeffect-directed translation.

In the revised thesis, we intend to make this more explicit and treat the formalism as a “coeffect-directed translation”, as well as explicitly define the target term language for the translation.

4.2 UNIQUE TYPING DERIVATION. As argued above, the fact the that semantics depends on the typing derivation is an important aspect of our work. This lets us better explore the design space, but it does not guide concrete language implementations that need a unique typing (cf. Haskell’s implicit parameters).

In the final version of the thesis, we intend to follow the example of Haskell and include an algorithmic formulation of the type system that produces a single default typing derivation[6] and resolves the ambiguity.

6 There are three sources of typing ambiguity (and hence semantic ambiguity) – subtyping, lambda abstraction and structural rules. We intend to make subtyping explicit in the language, add externally provided function for resolving ambiguities in lambda abstraction and have algorithmic typing rules instead of contraction, weakening and exchange rules.

4.3. EQUATIONAL STRUCTURE. One specific concern raised in the referees report is the lack of equational structure for the categorically-inspired structures. This arises from the fact that our usage of categorical structures (indexed comonads) is mainly to inspire the abstractions that can be used to structure context-aware computations (as opposed to focusing on using category theory to prove properties of context-aware programs).

Our work should be seen as a coeffect-directed translation to a target language with structures inspired by indexed comonads. Preserving this link is important in order to relate coeffect calculi with previous work [6], however our structure need not be seen as categorical semantics.

4.4. GENERALIZATION. One of the key claims of the thesis is that the comonad-inspired structure of the target language appropriately captures the structure of concrete examples of context-aware computations (presented in Chapter 3) and as a result, it provides an appropriate type system.

This claim is left implicit and we intend to address this in the final version (by proving that the three concrete examples spelled out in Chapter 3 do, indeed, fit the common patterns identified later in Chapters 4 and 5).

4.5. UNIVERSAL TYPE SYSTEMS. The thesis does not clearly explain the benefit of introducing the universal type systems based on the coeffect calculi, as we do not prove the usual type-safety property for the concrete systems. This becomes clearer when we consider the system as *type erasure*.

The type systems guarantee that *coeffect indices* will match in the resulting translation. The relationship from indices to values is injective in the general case, but we can also show a number of important properties (which will be clarified in the revised version) for the concrete systems, e. g. an upper bound on the number of required past values in causal data-flow.

5 SUMMARY

An interesting aspect of many context-aware computations is that they require access to some context, but do not explicitly specify where the context comes from. In case of implicit parameters, the values can be provided either by the call-site or by the declaration-site.

In some systems like Haskell's implicit parameters, this aspect is hidden through a mechanism that chooses a *default typing* (and a *default semantics*) and thus hides the ambiguity. Our approach is to make the ambiguity explicit in the type system. We follow F# and accept the fact that this makes the semantics of a term dependent on the typing and we extend Moggi's *type-directed semantics* into a *coeffect-directed semantics*.

Our use of categorical structures (namely *indexed comonads*) is similar to practical Haskell use of monads in that we use them to *capture a common structure*. This is in contrast with use of category theory for *proving program properties*. Correspondingly, our focus is on finding the right structure and showing that it captures our motivating examples and gives a type system with desirable properties.

This document explains an additional background and motivation that is missing in the submitted version of the thesis (Section 1 and 2) as well as additional clarification showing how the semantics of implicit parameters depends on the typing (Section 3). Finally, we list four specific improvements that will be made in the final version of the thesis to better reflect our approach and the methodology clarified in this document (Section 4).

BIBLIOGRAPHY

[1] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In *Programming Languages and Systems*, pages 331–350. Springer, 2014.

[2] C. D. Knudsen, Jørgen Lindskov. Advanced topics in exception handling techniques. 2006.

[3] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.

[4] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification. Available at `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2014.

[5] The F# Software Foundation. F# 3.0 language specification. Available at `http://fsharp.org/specs/language-spec`, 2015.

[6] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203:263–284, June 2008.