# Language support for context-aware computations

Tomas Petricek[1], supervisor: Alan Mycroft[1]

[1] University of Cambridge, 15 JJ Thomson Avenue, CB3 0FD, UK
tomas.petricek@cl.cam.ac.uk, alan.mycroft@cl.cam.ac.uk

## 1. Motivation

Modern software applications behave differently depending on the environment or context in which they execute. They often run in increasingly rich environments that provide resources (e.g. database or GPS sensor) and are gradually more diverse (e.g. multiple versions of different mobile platforms). Web applications are split between client, server and mobile components; mobile applications must be aware of the context and of the platform while the "internet of things" makes the environments even more heterogeneous; applications that access rich data sources need to propagate security policies and provenance information about the data.

Writing such context-aware applications is a fundamental problem of modern software engineering. The state of the art relies on ad-hoc approaches – using hand-written conditions or pre-processors for conditional compilation. Common problems that developers face include:

- **System capabilities.** When writing code that is cross-compiled to multiple targets (e.g. SQL, CUDA or JavaScript) the compilation often occurs at runtime and developers have no guarantee that it will succeed until the program is executed[1].
- **Platform versioning.** When developing application for multiple versions of a system (e.g. Android), developers rely on lazy loading at runtime or use conditional compilation using `#if`. The former delays errors to runtime, while the latter requires building all possible configurations to discover simple compile-time errors.
- **Resources & data availability.** When programming applications that access resources or data provided by the environment (e.g. specific database table, GPS sensor), the program typically performs dynamic check for the resource availability. However, this is not checked in any way – we have no easy way to tell what happens when the resource is not available (e.g. is there a fallback strategy or not?)
- **Data provenance & security.** Different kinds of data come with different policies – sensitive data (e.g. credit card number) should never be exposed; data from certain sources may have limited validity. Such policies are difficult to guarantee without extensive (and expensive) testing.

The presented research aims to solve such problems by integrating contextual information directly into the programming languages and, in particular, into the type system. This approach can guarantee correctness properties of programs and also allows development of additional tools – such as development editors with immediate feedback. To make the resulting language practical, we adopt two basic principles: first, the system must be unified, but expressive enough to capture a wide range of applications (highlighted above) and second, the system must be extensible – developers need to be able to specify properties they need to track for their specific application.

[1] For example, LINQ compiles a subset of C# to SQL at runtime, but may fail with "Method X has no supported translation to SQL". This is an important issue –Google search for the term reports 31400 results e.g. StackOverflow (2011).

## 2. Background

The work on context-aware programming languages connects two directions in existing research on the theory of programming languages. On one side, effect systems (Gifford and Lucasen (1986)) and monadic computations (Moggi (1991), Wadler and Thiemann (2013)) provide a detailed and established method for tracking what effects programs have – that is, how they *affect* the environment where they execute. On the other side, the work on comonadic notions of computations (Uustalu and Vene (2008)) shows how to use the mathematical dual of monads – comonads – to give categorical semantics of context-dependent computations.

Effect systems introduced track actions such as memory operations or communication. They are described by typing judgments of a form $\Gamma \vdash e: \tau, \sigma$ where $\Gamma$ is the context of a program (typically available variables), $e$ is the expression (program) itself, $\tau$ is the type of values returned by the program (e.g. integer or boolean) and $\sigma$ is a set of possible effects. The judgment states that, given the context $\Gamma$, an expression has a type $\tau$ and can only perform effects specified by the set $\sigma$. Wadler and Thiemann (2003) explain how this shapes effect analysis of a lambda abstraction – that is, how effect systems analyze the effects associated with a definition of a function:

> *In the rule for abstraction, the effect is empty because evaluation immediately returns the function, with no side effects. The effect on the function arrow is the same as the effect for the function body, because applying the function will have the same side effects as evaluating the body.*

This means that, when a programmer defines a function, the system records that executing the function will perform the effects of the body of the function. However, context-dependent computations do not match this pattern. A function may place context requirements on both the call-site and the declaration-site. This means that context-dependent computations have different syntactic properties.

## 3. Approach

Programming languages are fundamental tools used by software developers on daily basis and so the correctness of programming languages is absolutely crucial. An error in the design of a language that becomes popular can have massive influence on software quality. For this reason, we place strong emphasis on the theoretical foundations of the work. We also believe that engaging developers from the industry early is the best way to evaluate such project. This section briefly discusses both aspects of the approach.

### Theoretical foundations

We follow the approach pioneered by Gifford and Lucasen for effect systems. We extend the type system of (functional) programming languages with a notion of *context-dependence*. As outlined earlier, such systems have different syntactic properties than effect systems and they also differ philosophically – by tracking what programs *require* from the environment rather than tracking how they *affect* the environment.

For this reason, we associate the contextual information with the left-hand side of the entailment in the typing judgment. Our rules have a form $\Gamma, \sigma \vdash e: \tau$. The interpretation is that a program $e$ can only executed when provided with variables $\Gamma$ and an additional custom context $\sigma$ (and then it yields a value of type $\tau$).

The examples of context-dependent computations presented earlier fall into two categories. One kind captures the context of a program as a whole (e.g. resources or platform) and the other captures properties associated with individual variables (e.g. security or provenance). We develop the following two calculi to model the two situations:

- Our *flat calculus* is syntactically similar to effect systems. It tracks single information about the entire context. Such information may be e.g. a set (of required resources), number (platform version).
- Our *structural calculus* generalizes the flat calculus and captures more fine-grained structure. It associates a single piece of information with every variable of the context $\Gamma$. For example, when tracking provenance, each variable is associated with a set representing the labels of data sources.

### Typing rules

To provide more details, this section introduces the key aspects of the type systems for the two calculi. The details of the *flat calculus* can be found in Petricek et al. (2013). The details are of a technical nature, but they are the key for developing sound programming languages. More practical demonstration of the two calculi is available in the next two sections.

$$\frac{\Gamma, \mathrm{r} \vdash e_1 : \tau_1 \xrightarrow{s} \tau_2 \qquad \Gamma, \mathrm{t} \vdash e_2 : \tau_1}{\Gamma, r \vee (s \oplus t) \vdash e_1 e_2 : \tau_2}$$

$$\frac{(\Gamma, x : \tau_1), r \wedge s \vdash e : \tau_2}{\Gamma, \mathrm{r} \vdash \lambda x. e : \tau_1 \xrightarrow{s} \tau_2}$$

**Figure 1a.** Application and abstraction of the *flat calculus*

$$\frac{(x_1 : \tau, x_2 : \tau), r \times s \vdash e : \tau_1}{(x : \tau), r \otimes s \vdash e[x_1 \leftarrow x,\ x_2 \leftarrow x] : \tau_1}$$

$$\frac{\Gamma_1, r \vdash e_1 : \tau_1 \xrightarrow{s} \tau_2 \qquad \Gamma_2, t \vdash e_2 : \tau_1}{(\Gamma_1, \Gamma_2), r \times (s \oplus t) \vdash e_1 e_2 : \tau_2}$$

**Figure 1b.** Application and contraction of the *structural calculus*

The *flat calculus* (Figure 1a) uses tags of a structure $(S,\vee,\oplus)$. Variable contexts and domain of functions are annotated with a tag (written $\Gamma, \mathrm{r}$ and $\tau_1 \xrightarrow{r} \tau_2$, respecttively) to denote the context requirements. Application is typeable in a context that satisfies a combination of the requirements of the two expressions and the requirements of the function. Lambda abstraction splits the requirements of the body between the declaring context and the function (i.e. resources can be provided by both declaration and use site).

To track more fine-grained calculus, the *structural calculus* (Figure 1b) mirrors the structure of the variable context $\Gamma$ in the annotation using $\times$. Information associated with individual variables can be merged using two operations. The contraction rule combines information about two individual variables $x_1$ and $x_2$ into information $r \otimes s$ associated with a single variable. The application rule combines information about the first part of the context ($r$ corresponding to $\Gamma_1$) with an information $s \oplus t$, which specifies that all variables from $\Gamma_2$ (tagged with $s$) may affect the input of the function $e_1$ (tagged with $t$).

### Example: Flat calculus

To give a concrete example of the *flat calculus*, consider the following simple program which takes a price and converts it to another currency using a *resource* called `ConversionRate`:

```
let convertPrice price =
  (access ConversionRate) * price
```

In a distributed programming language (e.g. client-server web application), the function may be defined on the server, but then passed to the web browser and executed repeatedly (as the user edits price). The key aspect of the *flat calculus* is that the resource `ConversionRate` can be provided by either of the environments. It may come both from the server, but also obtained dynamically by the web application (for example, if the web application is connected to a web service that provides currency rate information).

### Example: Structural calculus

As an example of the *structural calculus*, consider a language that allows us to get a value of a variable (representing some changing data-source) $x$ versions back using the syntax `a[x]`. To track information about individual variables, we use a product-like operation × that mirrors the product structure of variables. For example, a program that accesses 5th value of `a` and 10th value of `b` looks as follows:

```
a[5] + b[10]
```

Such program requires the context $a\colon \mathrm{stream}, b\colon \mathrm{stream}, 5 \times 10$. The context specifies that `a` and `b` are both streams. The annotation $5 \times 10$ corresponds to the variable context $a, b$. It denotes that we need at most 5 and 10 past values of `a` and `b`. If we substitute `c` for both `a` and `b`, we get the following code:

```
c[5] + c[10]
```

The substitution corresponds to the first rule in Figure 1b. The annotations $5$ and $10$ are combined into one using the $\otimes$ operation – for this specific application, the operations needs to be the $max$ function and so the required context for the second code snippet is $max(5,10) = 10$.

## 4. Results

There is an increasing need for capturing how computations depend on the context in which they execute – examples from the literature include tracking of security information, provenance, resources or data sets accessed by programs. However, all of the above have been presented as single-purpose mechanism. We unify such notions of context-dependence into a single programming model.

### Specific contributions

The work done so far consists of analyzing different notions of context-dependence, looking how to improve data-access in main-stream languages and building the theoretical foundations for *flat calculus* and *structural calculus* presented above. Two publications written (or co-authored) by the author form the key contributions:

- Syme et al. (2013) focuses on data access as one of the most important notions of context-dependence used in majority of applications today. We presented a mechanism that integrates data access directly into a (functional) programming languages. Data from external data sources (such as XML, JSON or CSV files, web services, databases and many more) can be integrated by developing a compiler extension (*type provider*). Such provider enables the compiler to check that the structure of external data matches the expectations of the developer and it also provides autocomplete for the IDE (editor), which can offer information about available data.
- Petricek et al. (2013) presents the *flat calculus* outlined in the earlier section in detail. It looks at three examples of context-dependent computations (dynamically scoped parameters, resources and data-flow computations) and explains how the *flat calculus* unifies all three systems. Furthermore, it extends the semantic model of Uustalu and Vene (2008) to capture fine-grained information about context using indexed comonads.

In addition to the above, we are also developing a practical implementation of the programming model as an extension to the programming language F#. The extension follows a pragmatic approach – we aim to develop an extensions that practitioners can easily evaluate to provide feedback. This is done by adapting the F# *computation expression* syntax (see Petricek and Syme (2012)) and extending the type checking mechanism to accommodate custom structures such as sets (of resources), versions (of platforms) and other.

### Long-term outlook

In the long-term, we envision a programming language, together with additional tooling, that is capable of building programs that run as distributed computations in diverse environments and across different platforms. The compile-time checking provides developers with useful information (which functions can be reused in certain environments) and prevents bugs (attempting to access unavailable resource) and security issues.

Programming languages of the future will be able to use such information in order to cross-compile single program for a wide range of platforms (JVM, .NET, HTML5, JavaScript, native). Compilation of a program will also split program into components for different execution environments (server-side, client-side, mobile, etc.). Sadly, designing and developing such language (with sound theory and complete tool-chain) is well beyond the scope of a single 3-year PhD topic.

## References

D. K. Gifford and J. M. Lucassen (1986). Integrating Functional and Imperative Programming. In Proceedings of Conference on LISP and Functional Programming, 1986. ISBN 0897912004.

E. Moggi (1991). Notions of Computation and Monads. Information Computation,
Vol 93: 55–92. July 1991. ISSN 0890-5401

P.Wadler and P. Thiemann (2003). The Marriage of Effects and Monads. ACM Transactions
on Computational Logic, Vol 4:1–32, January 2003.

T. Murphy, VII., K. Crary, and R. Harper (2008). Type-safe distributed programming with ML5. In Proceedings of Conference on Trustworthy Global Computing, 108–123, 2008.

T. Uustalu and V. Vene (2008). Comonadic Notions of Computation. In Electronic Notes in Theoretical Computer Science. Vol 203:263–284, June 2008. ISSN 1571-0661.

D. Syme, K. Battocchi, K. Takeda, D. Malayeri, T. Petricek (2013). Themes in information-rich functional programming for internet-scale. In Proceedings of DDFP 2013.

T. Petricek, D. Orchard and A. Mycroft (2013). Coeffects: Unified static analysis of context-dependence. To appear in Proceedings of ICALP 2013.

T. Petricek, D. Syme (2012). Syntax Matters: Writing abstract computations in F#. Available in pre-proceedings of TFP 2012. http://www.cl.cam.ac.uk/~tp322/drafts/notations.html

StackOverflow (2011), by user 'brechtvhb'. Method x has no supported translation to SQL.
http://stackoverflow.com/questions/5309338/ (Retrieved 3 March 2013)