Towards Operational Semantics of Contexts in Functional Languages

David Sands
DIKU, University of Copenhagen*

Abstract

We consider operational semantics of contexts (terms with holes) in the setting of lazy functional languages, with the aim of providing a balance between operational and compositional reasoning, and a framework for semantics-based program analysis and manipulation.

Introduction

In this note we initiate a new direction in the semantics of functional programs. The approach is based on operational semantics; our aims are to provide a operational route to high-level semantic issues, such as program analysis and source-to-source transformation. We investigate the idea of giving a direct operational semantics to *program contexts*—that is, "incomplete" programs containing a number of *holes* in the place of some subexpressions.

The idea of providing an operational semantics for contexts has been studied by Larsen (et al) for process algebras [Lar86][LX91]. In that setting, a context is viewed as an *action transducer*, which consumes actions provided by its internal processes (the holes) and produces externally observable actions. The operational semantics of contexts contains transitions of the form

$$\mathcal{C} \xrightarrow[a]{b} \mathcal{C}'$$

which is interpreted as: *by consuming action a, context $\mathcal{C}$ can produce action b and change into $\mathcal{C}'$.*

We describe some initial steps towards providing an operational semantics for contexts in a functional setting.

Functional Action-Transducers

In the process setting a context is viewed as an *action transducer*. What is the corresponding notion for a context in the functional setting? In a functional language, the role of an "action" is played by the *observables* of the language: namely a lazy data constructor— `cons`, `true`, "λ".

*Universitetsparken 1, 2100 København Ø, DENMARK. e-mail: `dave@diku.dk`

We take a bold step, and demand that the "actions" should themselves be contexts—but not arbitrary contexts. They should be contexts built from the observables of the language. We will call these *observable contexts.* Observable contexts will be ranged-over by $\mathcal{O}$, $\mathcal{O}'$, etc. For some context $\mathcal{C}$ containing occurrences of a single hole, if we have a *transduction* of the form:

$$\mathcal{C} \xrightarrow[\mathcal{O}]{\mathcal{O}'} \mathcal{C}'$$

then we will require that:

$$\mathcal{C}[\mathcal{O}] \simeq \mathcal{O}'[\mathcal{C}'] \qquad (*)$$

where the notation $\mathcal{C}[e]$ denotes context $\mathcal{C}$ with e placed "in the hole", and $\simeq$ denotes the usual operational equivalence, extended to capture-free contexts in the obvious way. This equation does not capture everything we expect from a context semantics. What we expect is that the transduction should be as lazy as possible — ie. we could not have found a smaller "input" $\mathcal{O}$ that would have given the same observable "output" $\mathcal{O}'$. If the language is *sequential*[Ber78] then we expect the semantics to give us the *minimal* $\mathcal{O}$.

For example, a context of the form `if [] = 0 then` $\mathcal{C}_1$ `else (leaf` $\mathcal{C}_2$`)` (containing multiple occurrences of a single hole), where `leaf` is a constructor, would have a transduction:

$$\texttt{if } [\,] = \texttt{0 then } \mathcal{C}_1 \texttt{ else (leaf } \mathcal{C}_2\texttt{)} \xrightarrow[\texttt{suc []}]{\texttt{leaf []}} \mathcal{C}_2 \circ \texttt{suc}[\,]$$

where $\circ$ is context composition, so $\mathcal{C}_2 \circ \texttt{suc}[\,]$ denotes the context $\mathcal{C}_2[\texttt{suc}[\,]]$.

Problems Giving a full operational semantics for contexts is difficult because:

- contexts can consume without producing an observable.

  For example `if` $[\,] = 0 \ldots$ can consume a constructor without necessarily being able to produce anything. (Similar situation would arise in Larsen's work if one did not consider the silent action τ to be observable.)

- The number of occurrences of a given hole may increase under transduction (assuming we use some mechanism like β-reduction in our semantics).

- The number of distinct holes in a context can increase under transduction, in the presence of n-ary constructors.

  eg. If $\mathcal{C}_1$ has a single hole, and $\mathcal{C}_1 \xrightarrow[\texttt{cons}[\,]_1\,[\,]_2]{} \mathcal{C}_2$ then $\mathcal{C}_2$ has two distinct holes.

- How do we treat higher-order functions?

- Contexts can capture variables by means of binding operators

  eg.
  ```
  case e of
        nil  ⇒ e'
    cons h t ⇒ C2
  ```

A Simplified Case

As a first step we study only a very simple language. We avoid almost all of the above problems by considering a language with the following features:

- First-order functions
- No binding operators
- Unary constructors and constants (= nullary constructors) as the only values.

The language we consider consists of first-order recursion equations with possible pattern-matching on the first argument (non-nested), based on unary or nullary constructors. Here are some example definitions:

```
add 0 x        =  x
add (suc y) x  =  suc (add y x)

twice x        =  add x x
```

Notation Let u range over both unary constructors (eg. `suc`) and constants (eg. `0`,`true` etc.). The observable contexts are then given by

$$\mathcal{O} \quad ::= \quad [\,] \quad | \quad u \quad | \quad u\,\mathcal{O}$$

For simplicity of presentation, we will only consider contexts with a single hole (occurring zero or more times). (We will consider an expression to be a context with zero occurrences of the hole.) For this restricted language, the extension to handle *polyadic* contexts (contexts with several distinct holes) is straightforward (eg. borrowing the notations from [LX91]).

In the context transductions, if u is a unary constructor, then we write observable context $u\,[\,]$ as simply u, and occurrences of the trivial context $[\,]$ will simply be omitted from the transductions, so we will write $\mathcal{C} \xrightarrow{\texttt{suc}} \mathcal{C}'$ in place of $\mathcal{C} \xrightarrow[\texttt{[]}]{\texttt{suc[]}} \mathcal{C}'$. If u is a constant, then u can also be denoted $u()$. We add the unit expression $()$ to the language of contexts.

Language Rules

We define the following transductions involving terms of the language:

$$u\,\mathcal{C} \xrightarrow{u} \mathcal{C} \tag{1}$$

$$\texttt{f}\ \vec{\mathcal{C}} \longrightarrow \texttt{e}\{\vec{\texttt{x}} := \vec{\mathcal{C}}\} \qquad \texttt{if f}\ \vec{x} = e \tag{2}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}]{u} \mathcal{C}_2}{\mathtt{f}\ \mathcal{C}_1\ \vec{\mathcal{C}} \xrightarrow[\mathcal{O}]{} e\{y := \mathcal{C}_2\}\{\vec{x} := \vec{\mathcal{C}} \circ \mathcal{O}\}} \quad \text{if } \mathtt{f}\ (u\ y)\ \vec{x} = e \tag{3}$$

In the last rule, the composition $\vec{\mathcal{C}} \circ Con'$ denotes the vector of contexts obtained by composing each context in $\vec{\mathcal{C}}$ with Con'. To check that the rule satisfies the desired property $(*)$, assume that from the antecedent we have

$$\mathcal{C}_1 \circ \mathcal{O} \simeq u\ \mathcal{C}_2$$

then

$$\begin{array}{rcl} (\mathtt{f}\ \mathcal{C}_1\ \vec{\mathcal{C}}) \circ \mathcal{O} & \simeq & \mathtt{f}\ (\mathcal{C}_1 \circ \mathcal{O})\ (\vec{\mathcal{C}} \circ \mathcal{O}) \\ & \simeq & \mathtt{f}\ (u\ \mathcal{C}_2)\ (\vec{\mathcal{C}} \circ \mathcal{O}) \\ & \simeq & e\{y := \mathcal{C}_2\}\{\vec{x} := \vec{\mathcal{C}} \circ \mathcal{O}\} \end{array}$$

These rules are straightforward, since they follow the "small-step" semantics of the language (ie. they are non compositional.) We recover the ability to reason compositionally using the following context rules:

Context Rules

$$[\,] \xrightarrow[u]{u} [\,] \quad (u \text{ unary}) \tag{4}$$

$$[\,] \xrightarrow[u]{u} () \quad (u \text{ constant}) \tag{5}$$

$$\mathcal{C} \longrightarrow \mathcal{C} \tag{6}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_1']{\mathcal{O}_1} \mathcal{C}_2 \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_2']{\mathcal{O}_2} \mathcal{C}_3}{\mathcal{C}_1 \xrightarrow[\mathcal{O}_1' \circ \mathcal{O}_2']{\mathcal{O}_1 \circ \mathcal{O}_2} \mathcal{C}_3} \tag{7}$$

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_2]{\mathcal{O}_1} \mathcal{C}_1' \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_2} \mathcal{C}_2'}{\mathcal{C}_1 \circ \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_1} \mathcal{C}_1' \circ \mathcal{C}_2'} \tag{8}$$

The last rule is the *uniform rule* of [Lar89][LX91], and it characterises all the transductions of composed contexts.

Properties

Closed expressions can be viewed as contexts containing zero holes. In this way the rules above can be seen to subsume the usual large-step and small-step structural operational semantics. Suppose the large step semantics defines an evaluation relation $\Downarrow$ (we omit the routine definition), then we have the following:

$$e \Downarrow a \iff e \xrightarrow{u} e' \quad \wedge \quad (u\ e' \equiv a)$$

For example, if `I` is the identity function, then we have the following example proof:

$$\text{compose}\ \dfrac{2\ \dfrac{}{\mathtt{I\ (suc\ 0)} \longrightarrow \mathtt{(suc\ 0)}} \qquad 1\ \dfrac{}{\mathtt{(suc\ 0)} \xrightarrow{\mathrm{suc}} \mathtt{0}}}{\mathtt{I\ (suc\ 0)} \xrightarrow{\mathrm{suc}} \mathtt{0}}$$

But this is not the whole story for evaluation of closed expressions. Unlike the structural operational semantics for $\Downarrow$, the *proof* of $e \xrightarrow{u} e'$ is not unique.

An important point is that by use of the composition rule (8) we can vary the compositionality. This means that when we need to prove a property of a function application `f` e, we can split this into a context `f` [ ] and the sub-term e, and derive the transition of the composed system in terms of these components.

As an example, using the functions defined earlier, consider the term `twice (I (suc0))`. We can prove that

$$7\ \dfrac{2\ \dfrac{}{\mathtt{twice\ [\,]} \rightarrow \mathtt{add\ [\,]\ [\,]}} \qquad A}{\mathtt{twice\ [\,]} \xrightarrow[\mathrm{suc}]{\mathrm{suc}} \mathtt{add\ [\,]\ (suc\ [\,])}}$$

where A is the sub-proof:

$$7\ \dfrac{4{,}1\ \dfrac{[\,] \xrightarrow[\mathrm{suc}]{\mathrm{suc}} [\,]}{\mathtt{add\ [\,]\ [\,]} \xrightarrow[\mathrm{suc}]{} \mathtt{suc\ add\ [\,]\ (suc\ [\,])} \xrightarrow{\mathrm{suc}} \mathtt{add\ [\,]\ (suc\ [\,])}}}{\mathtt{add\ [\,]\ [\,]} \xrightarrow[\mathrm{suc}]{\mathrm{suc}} \mathtt{add\ [\,]\ (suc\ [\,])}}$$

and so using the composition rule we obtain:

$$8\ \dfrac{\mathtt{twice\ [\,]} \xrightarrow[\mathrm{suc}]{\mathrm{suc}} \mathtt{add\ [\,]\ (suc\ [\,])} \qquad \mathtt{I\ (suc\ 0)} \xrightarrow{\mathrm{suc}} \mathtt{0}}{\mathtt{twice\ (I\ (suc\ 0))} \xrightarrow{\mathrm{suc}} \mathtt{add\ 0\ (suc\ 0)}}$$

Note that in the example, because of the use of the compositional rule, there is only one sub-proof for the expression `I (suc 0)`, whereas under the standard call by name SOS we would have two sub-proofs.

We conjecture that proofs which always treat function calls compositionally in this way (we need to generalise to n-ary contexts to do this for n-ary functions) have size proportional to the number of evaluation steps required under standard call-by-need computation. This form of proof corresponds to the *demand function* used in Bjerner and Holmström's call-by-need time-analysis [BH89].

Further Work

In the remainder of this paper we consider the directions for further development, which mostly concern tackling the problems of richer languages.

Polyadic Contexts

The above semantics is easily extended to handle polyadic contexts, but if we go beyond just unary constructors then the extension quickly becomes notationally complex. The problem is that consuming an observable context may give rise to several new holes, and producing a observable context means that a transduction may result in several contexts. Our proposal for dealing with these problems is to adopt a different kind of transduction. Instead of requiring that a transduction

$$\mathcal{C} \xrightarrow[\mathcal{O}]{\mathcal{O}'} \mathcal{C}'$$

implies that $\mathcal{C} \circ \mathcal{O} \simeq \mathcal{O}' \circ \mathcal{C}'$, the requirement is that

$$\mathcal{C} \circ \mathcal{O} \simeq \mathcal{O}' \circ \mathcal{C}' \circ \mathcal{O}$$

In this way the "type" of the holes in the derived context $\mathcal{C}'$ is the same as in $\mathcal{C}$. The addition of n-ary constructors means that $\mathcal{C}'$ must be a vector of contexts.

With this interpretation of context transductions, the uniform composition rule now has the form:

$$\frac{\mathcal{C}_1 \xrightarrow[\mathcal{O}_2]{\mathcal{O}_1} \mathcal{C}_1' \quad \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_2} \mathcal{C}_2'}{\mathcal{C}_1 \circ \mathcal{C}_2 \xrightarrow[\mathcal{O}_3]{\mathcal{O}_1} \mathcal{C}_1' \circ \mathcal{O}_2 \circ \mathcal{C}_2'}$$

Higher-Order Functions

If we focus on contexts which cannot capture variables, then higher-order functions can be thought of as introducing an extra hole. We anticipate that to deal with polyadic contexts in their full generality we will need a notation along the lines of Martin-Lö f's theory of arities, so $(X)\mathcal{C}$ will denote a context with a single hole named X. Then a lambda-context could have a transition

$$(\vec{Y})\lambda y.\mathcal{C} \xrightarrow{\lambda} (X)(\vec{Y})\mathcal{C} \qquad X \notin \vec{Y}.$$

Abstract Contexts and Relativised Equivalence

Two natural directions are to consider static analysis problems, and notions of relativised equivalence. We should consider:

- Bisimulation-like characterisations of context-equivalence along the lines of [Abr90][How89].
- Relativised equivalences $\simeq_{\mathcal{C}}$:

$$e \ \simeq_{\mathcal{C}} \ e' \iff \mathcal{C}[e] \simeq \mathcal{C}[e']$$

- Semantics for abstract observable contexts (whose meaning is a set of contexts);
- A notion of *environment* as a dual to (abstract) observable contexts, thus generalising the demand semantics of [San93].

These points should enable an operational formalisation of *context analysis* of the form of [Hug87][WH87].

Guarded Contexts

In a somewhat orthogonal study we have considered context semantics for a restricted class of contexts (a form of guarded contexts) for a higher-order language with binding operators and arbitrary lazy constructors. The principle technical problem in this context semantics is to handle holes which occur under bound variables. This operational semantics of contexts finds immediate application to the problem of correct folding in program transformation. It also provides a simple form of "applicative bisimulation up to context" proof technique, a la Sangiorgi [San94].

Acknowlegement An earlier investigation of the subject of this note was undertaken together with Sebastian Hunt a few years ago. Our attempt failed, because we were over-ambitious in trying to give *only* compositional rules. But a number of ideas crystalised from our attempt, and have influenced the current development. One idea in particular —that the "actions" should themselves be contexts—is due to Sebastian.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[Ber78] G. Berry. Stable models of typed lambda calculi. In *5th Coll. on Automata Languages and Programming*, LNCS 62. Springer-Verlag, 1978.

[BH89] B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and Computer Architecture, conference proceedings*, pages 157–165. ACM press, 1989.

[How89] D. J. Howe. Equality in lazy computation systems. In *Fourth annual symposium on Logic In Computer Science*, pages 198–203. IEEE, 1989.

[Hug87] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, March 1987.

[Lar86] K. G. Larsen. *Context-Dependent Bisimulation Between Processes*. PhD thesis, Department of Computing, University of Edinburgh, 1986.

[Lar89] K. G. Larsen. Compositinal theories based on an operational semantics of contexts. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in LNCS. Springer-Verlag, 1989.

[LX91] K. G. Larsen and L. Xinxin. Compositinality through an operational semantics of contexts. *J. Logic and Computation*, 1(6):761–795, 1991.

[San93] D. Sands. A naïve time analysis and its theory of cost equivalence. TOPPS report D-173, DIKU, 1993. To appear: Journal of Logic and Computation, 1995.

[San94] D. Sangiorgi. On the bisimulation proof method. Technical report, University of Edinburgh, 1994.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, pages 385–407, Portland, Oregon, September 1987.