URL: http://www.elsevier.nl/locate/entcs/volume10.html 16 pages

Computing with Contexts
A simple approach

David Sands

Department of Computing Science,
Chalmers University of Technology and Göteborg University,
S-412 96 Göteborg, Sweden; dave@cs.chalmers.se

Abstract

This article describes how the use of a higher-order syntax representation of contexts [due to A. Pitts] combines smoothly with higher-order syntax for evaluation rules, so that definitions can be extended to work over contexts. This provides "for free" — without the development of any new language-specific context calculi — evaluation rules for contexts which commute with hole-filling. We have found this to be a useful technique for directly reasoning about operational equivalence. A small illustration is given based on a unique fixed-point induction principle for a notion of guarded context in a functional language.

1 About contexts

The notion of a context is widely used in programming language semantics — for example in the definition of operational equivalences, or program transformation, and in certain styles of operational semantics definitions.

A context is just a term with some *holes*. The holes are place-holders for missing subterms. Each hole may occur zero or more times, and the process of *filling a hole with a term* is the textual operation of replacing all occurrences of a hole by the corresponding term. For most of this article we will consider contexts with just one hole, and that hole may occur zero or more times in the context.

The difference between hole-filling and the usual notion of substitution arises when the language contains binding operators; filling a hole with a term may cause variable in the term to be bound.

For example, the lambda-calculus context $(\lambda x.[\,])\lambda x.\lambda y.[\,]$ is a context with one hole, written $[\,]$, and this hole occurs twice. Call this context C. Filling the hole in C with the term $x\ y$, which is typically denoted by $C[x\ y]$, results in the term $(\lambda x.x\ y)\lambda x.\lambda y.x\ y$. Note that the resulting term has one free occurrence of the variable y, and that the variable x has been *captured* — in this example by two different lambda abstractions. Because of this possibility

of variable-capture (even in the case when a hole occurs just once in a term), such contexts cannot be identified up to renaming of bound variables, since renaming does not "commute with hole filling". In this example:

$$\begin{array}{ccc} (\lambda x.[\,])\lambda x.\lambda y.[\,] & \xrightarrow{\text{fill with } x\ y} & (\lambda x.x\ y)\lambda x.\lambda y.x\ y \\ \alpha\text{-convert}\downarrow & & \vdots \\ (\lambda z.[\,])\lambda x.\lambda y.[\,] & \xrightarrow{\text{fill with } x\ y} & (\lambda z.x\ y)\lambda x.\lambda y.x\ y \end{array}$$

Note that the terms of the right-hand side are not α-convertible. Such problems arise when one attempts to argue properties about the behaviour of "terms-in-context". For example, in direct proofs about contextual equivalence.

Representations of Contexts: Previous work

Talcott and Mason *et al* [MT91,AMST97,Tal97] present some direct proofs about contextual equivalence in a lambda calculus with effects. In order to be made rigorous, these arguments require the development of a calculus of generalised contexts. This development is based on Talcott's earlier work on a theory of binding structures [Tal92,Tal93]; related work is reported by Mason [Mas96].

The Talcott/Mason approach takes the following form. For the particular term language under study one must

(i) introduce a generalised definition of contexts where holes are decorated with (generalised) substitutions;

(ii) establish basic definitions for generalised contexts, such as substitution and hole-filling;

(iii) lift the definition of computation (reduction) up to generalised contexts

(iv) establish the main result: that reduction "commutes with hole filling"

One can motivate the Talcott/Mason representation of contexts by considering the last point: the problem of extending the definition of reduction to work on contexts in such a way that it commutes with hole-filling. Consider the lambda-calculus context $(\lambda x[\,])I$ (where I is the identity function, $\lambda x.x$). If one extended β-reduction naïvely to contexts we would obtain:

$$(\lambda x[\,])I \to_\beta [\,]$$

This is clearly not adequate, since e.g. filling the hole with x does not "commute" with this reduction:

$$\begin{array}{ccc} (\lambda x.[\,])I & \xrightarrow{\quad\beta\quad} & [\,] \\ \text{fill with } x\downarrow & & \downarrow\text{fill with } x \\ (\lambda x.x)I & \xrightarrow[\beta]{} I \quad \not\equiv & x \end{array}$$

The problem here is that the reduction step "forgets" the term I. The solution

is to decorate holes with explicit substitutions, so that e.g.,

$$(\lambda x[\,])I \rightarrow_\beta [\,]^{[I/x]}$$

But once this extension is made, the range of substitutions must also be permitted to contain generalised contexts, so that e.g.,

$$(\lambda y.[\,])[\,]^{[I/x]} \rightarrow_\beta [\,]^{[[\,]^\theta/y]} \qquad \text{where } \theta = [I/x]$$

Summary
In this note we show how an alternative approach to representing contexts significantly simplifies, and to some extent generalises the "context calculus" that is required to compute with contexts.

- The first simplification is that the representation of contexts — which is due to Pitts [Pit94] — is based on higher-order syntax (i.e. typed lambda-calculus as a syntactic meta-language), so no new calculus needs to be developed;
- The second simplification is that we show how many common definitions involving terms, e.g., evaluation relations, reduction, abstract-machine steps and sets of terms or contexts specified by grammars, can be "automatically" extended to contexts in such a way that they commute with hole-filling "for free".
- It generalises previous approaches in the sense that it is not tied to a particular syntax or a particular relation involving terms (i.e. reduction).

The author has already made extensive use of these techniques in a number of proofs about operational semantics; the initial motivations for this work were the proofs about the GDSOS operational semantics rule-format presented in [San97]. The proofs[1] of many of the results reported there build on the ideas presented in this note.

There are a few alternative context-calculi that have appeared in the literature. Lee and Friedman [LF96] propose a calculus in which contexts are regarded as concrete representations (source code) for terms. More recently, Hashimoto and Ohori [HO98] describe a context calculus which extends lambda calculus to include first-class contexts (via context abstraction and context application (= hole-filling). Their main result is that the calculus is confluent, and this is achieved with the help of a type system. Their calculus involves labelling hole variables with renamings, which is reminicent of Talcott's approach.

The remainder of this note is organised as follows: In section 2 we introduce Pitts' representation of contexts. In section 3 we show how this representation combines smoothly with the use of higher-order syntax in term-based definitions (e.g. operational semantics rules), so that definitions extend to

[1] Due to space limitations, these proofs do not appear in the conference article

contexts "for free". By way of illustration, in the concluding section we look at a small application of the ideas to the proof of a unique fixed-point theorem (in the style of guarded-recursion theorems in process calculus) for a lazy lambda-calculus with constructors.

2 A Second-order Representation of Contexts

The definition of contexts which we adopt here was introduced by Pitts in [Pit94]. Pitts' main motivation for adopting a non-standard definition of contexts appears to be that standard contexts cannot be identified up to α-equivalence.

Pitts solves this problem by using *function variables* to represent holes, and to represent hole filling by *substitution* of meta-abstractions for these function variables. This representation does indeed enable contexts to be identified up to renaming of bound variables; the key point of this note is that the representation allows hole-filling to "commute" with many other relations involving terms, not just α-equivalence.

The basic idea can be illustrated by some examples. A hole will be represented by an application of some function-variable ξ to a vector of variables. Each function variable has a given *arity*, which dictates exactly how many arguments it expects. For example, the conventional context $(\lambda x[\,])I$ could be represented by $(\lambda x.\xi(x))I$ (so in this case ξ has arity 1). This representation of the context can be α-converted in the usual way. Hole-filling is represented by substituting a *meta-abstraction* for ξ. Filling $(\lambda x[\,])I$ with x will now be represented by applying the substitution $[{}^{(x)x}/_{\xi}]$

$$\begin{aligned}((\lambda x.\xi(x))I)[{}^{(x)x}/_{\xi}] &\equiv ((\lambda y.\xi(y))I)[{}^{(x)x}/_{\xi}] \\ &\equiv ((\lambda y.(x)x \cdot (y))I) \\ &\equiv ((\lambda y.y)I)\end{aligned}$$

In the second line we have informally written a meta-syntactic application $(x)x \cdot (y)$ to represent the application of the abstraction $(x)x$ to the variable y is reduced to y. Since we only need second-order function-variables (i.e., function variables which range over abstractions of terms) these meta-beta-reductions can be incorporated into the definition of substitution itself.

Notice also with this example that we can now β-reduce the context:

$$((\lambda x.\xi(x))I) \to_{\beta} \xi(I)$$

and now we get $\xi(I)[{}^{(x)x}/_{\xi}] \equiv I$ as we hoped. As we shall see in the next section, the fact that this works boils down to a simple associativity property of substitutions (the standard lambda-calculus substitution lemma).

Term Syntax

In the general definitions which follow we will adopt a type-theory-style abstract syntax for terms (see e.g. [HL78,MN95,NPS90,PE88]). For specific examples we will use the familiar terms from the lambda-calculus, and their usual concrete syntax.

First we fix a countably infinite set *Var* of ordinary variables. A language L is specified by a set of operators O of a fixed arity. As usual, the arity specifies the number of *operands* for each operator, but it specifies more than just this, since we wish to specify the syntax of operators with binding. Each operand is possibly an abstraction, i.e., a list of zero or more distinct variables followed by a term, where the variables are considered bound in the term. The arity of an operator is therefore given by a sequence of natural numbers; the length of the sequence is the number of operands, and the natural numbers are the number of bound variables associated with the corresponding operand. For example, the terms of the lambda calculus would be represented in this syntax by the set of operators $\{\lambda, \boldsymbol{apply}\}$ with $\text{arity}(\lambda) = (1)$, $\text{arity}(\boldsymbol{apply}) = (0, 0)$.

Let x, y, etc., range over *Var*, and let $\boldsymbol{p}$, $\boldsymbol{q}$ range over O.

The terms of L, T, ranged over by M, N are defined inductively as follows:

$$\frac{}{x \in T} \qquad \frac{M_1 \in T \cdots M_n \in T}{\boldsymbol{p}(\,(\bar{x}_1)M_1, \ldots, (\bar{x}_n)M_n) \in T} \quad \text{arity}(\boldsymbol{p}) = (k_1, \ldots k_n)$$

each $\bar{x}_i$ is a list of k_i distinct variables.

For example, the term $(\lambda x.y)z$ would be written in this abstract syntax as $\boldsymbol{apply}(\boldsymbol{\lambda}((x)y), z)$.

Contexts

Now we can be more precise about the definition of contexts. We follow [Pit94] quite closely, albeit with a more general term-syntax, (but a more casual treatment of free variables).

Contexts are an extension of terms to include hole-variables. Fix a countably infinite set *HVar* of hole variables. Each hole variable ξ, has an associated arity (which we will also denote $\text{arity}(\xi)$) which is a natural number. Hole variables of arity n will range over abstractions of the form: $(x_1, \ldots, x_n)M$.

The contexts $T^\star$, ranged over by $\mathbb{C}$, $\mathbb{D}$, $\mathbb{C}'$ etc are defined inductively as follows:

$$\frac{}{x \in T^\star} \qquad \frac{\mathbb{C}_1 \in T^\star \cdots \mathbb{C}_n \in T^\star}{\xi(\,\mathbb{C}_1, \ldots, \mathbb{C}_n) \in T^\star} \quad \text{arity}(\xi) = n$$

$$\frac{\mathbb{C}_1 \in T^\star \cdots \mathbb{C}_n \in T^\star}{\boldsymbol{p}(\,(\bar{x}_1)\mathbb{C}_1, \ldots, (\bar{x}_n)\mathbb{C}_n) \in T^\star} \quad \text{arity}(\boldsymbol{p}) = (k_1, \ldots k_n)$$

each $\bar{x}_i$ is a list of k_i distinct variables.

Hole Filling

Hole filling is defined by substitution. The usual definition of substitution of terms for variables is routinely extended to substitution of contexts for variables. We use the notation $\mathbb{C}[\bar{\mathbb{C}}/\bar{x}]$ to denote the result of simultaneous substitution of contexts $\bar{\mathbb{C}} = \mathbb{C}_1, \ldots, \mathbb{C}_n$ for some distinct variables $\bar{x} = x_1, \ldots, x_n$.

Substitution in the case of hole variables requires a little more attention. If ξ is a hole variable of arity k, then we need to define the result of substituting a meta-abstraction of the form $(x_1, \ldots, x_k)\mathbb{D}$ for occurrences of ξ in a context $\mathbb{C}$.

The definition of substitution is much as one would expect, inductively following the term-structure, and avoiding free-variable capture along the way. The interesting case is the following:

$$\xi(\mathbb{C}_1, \ldots, \mathbb{C}_k)[(x_1, \ldots, x_k)\mathbb{D}/\xi] = \mathbb{D}[\bar{\mathbb{C}}'/\bar{x}]$$
$$\text{where } \bar{\mathbb{C}}' = \mathbb{C}_1[(x_1, \ldots, x_k)\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(x_1, \ldots, x_k)\mathbb{D}/\xi]$$

This step can be (informally) broken down into two stages,

(i) the substitution of $(\bar{x})\mathbb{D}$ for ξ to yield
$$(\bar{x})\mathbb{D} \cdot (\mathbb{C}_1[(\bar{x})\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(\bar{x})\mathbb{D}/\xi])$$

(ii) the meta-β-reduction of the application to give
$$\mathbb{D}[\mathbb{C}_1[(\bar{x})\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(\bar{x})\mathbb{D}/\xi]/\bar{x}]$$

Of course this is only informal, since the meta-application which appears after step 1 is not part of the syntax.

About substitution

Something which should be borne in mind when considering the presentation of syntax that we are using is that it is really just a fragment of a simply typed lambda-calculus. There is just one base-type, o, representing the type of terms. An operator of arity e.g., $(0, 2)$ can be thought of as a constant of type $(o \times ((o \times o) \to o)) \to o$. An ordinary variable has type o, and a hole-variable of arity n can be thought of as having type

$$\underbrace{(o \times \cdots \times o)}_{\text{n}} \to o$$

Thinking of our syntax as a typed lambda calculus one should note that we only consider terms which are head-normal forms. One could add meta-application and projections to the syntax to obtain a more complete syntactic metalanguage (as in Martin-Löf's theory of arities [NPS90]) although we do not consider this necessary for present purposes.

It should now be no surprise that certain standard results from the lambda-calculus carry over to contexts. We mention one such result which will be

useful in the next section: a cut-down version of the *substitution lemma*, which states a commutativity property of substitutions:

Lemma 2.1 (Substitution Lemma) *If* $y \notin \text{FV}(\mathbb{D}) \setminus \bar{x}$ *then*

$$\mathbb{C}[{}^{(\bar{x})}\mathbb{D}/_{\xi}][{}^{N}/_{y}] \equiv \mathbb{C}[{}^{N}/_{y}][{}^{(\bar{x})}\mathbb{D}/_{\xi}]$$

Representing conventional contexts

If we are to use this alternative – and more general – representation of contexts in order to facilitate reasoning about e.g. contextual (operational) equivalence, then it is important to understand the connection to the conventional notion of context.

Conventional contexts correspond to a strict subset of contexts — namely those in which all occurrences of a hole variable ξ occur as $\xi(\bar{x})$ for some distinct variables $\bar{x}$.

For a conventional context C containing zero or more occurrences of a hole $[\,]$, let $\mathit{traps}(C)$ denote the set of the variables which are in scope at at least one occurrence of the hole in C – i.e. the set of variables which may become trapped (captured) at some occurrences when the hole is filled. So for example $\mathit{traps}((\lambda x.[\,])(\lambda y.[\,])) = \{x, y\}$. Let $\overline{\mathit{traps}}(C)$ denote some canonical vector of the trapped variables (e.g., listed from left-to-right according to the bindings in C).

For each context $\mathbb{C}$, we inductively define the mapping $\langle\cdot\rangle_{\mathbb{C}}$ which takes a conventional context to a generalised context by replacing holes with the context $\mathbb{C}$:

$$\begin{aligned}
\langle x\rangle_{\mathbb{C}} &= x \\
\langle \boldsymbol{p}(\,(\bar{x}_1)C_1, \ldots, (\bar{x}_n)C_n)\rangle_{\mathbb{C}} &= \boldsymbol{p}(\,(\bar{x}_1)\langle C_1\rangle_{\mathbb{C}}, \ldots, (\bar{x}_n)\langle C_n\rangle_{\mathbb{C}}) \\
\langle [\,]\rangle_{\mathbb{C}} &= \mathbb{C}
\end{aligned}$$

In other words, mixing the conventional and general context notations we could say that $\langle C\rangle_{\mathbb{C}} = C[\mathbb{C}]$.

A conventional context C can be represented by $\langle C\rangle_{\xi(\bar{x})}$, where $\bar{x}$ are the captured variables of C. Then the operation of filling C with the term M is represented by the substitution $[{}^{(\bar{x})}M/_{\xi}]$. The following lemma makes this claim precise:

Lemma 2.2 *For any conventional context* C*, and any term* M*, if* $\overline{\mathit{traps}}(C) = \bar{x}$ *and* ξ *is any hole variable of arity* $|\bar{x}|$*, then*

$$C[M] \equiv \langle C\rangle_{\xi(\bar{x})}[{}^{(\bar{x})}M/_{\xi}]$$

The proof is by induction on the structure of C.

The extended definition of contexts (henceforth called simply contexts) subsume conventional contexts; conventional contexts correspond to contexts with one hole variable, and for which all occurrences are identically of the form $\xi(\bar{x})$ for some vector of distinct variables $\bar{x}$.

3 Extending Definitions from Terms to Contexts

The purpose of this section is to show how typical syntax-oriented definitions can be lifted to operate over contexts in a natural way.

We proceed by considering a tiny example: an evaluation relation for the lazy lambda calculus.

$$\frac{M \Downarrow \lambda x.M' \quad M'[N/x] \Downarrow N'}{MN \Downarrow N'}$$

$$\frac{}{\lambda x.M \Downarrow \lambda x.M}$$

We would like to lift this definition to contexts in the following obvious way:

$$\frac{\mathbb{C} \Downarrow \lambda x.\mathbb{C}' \quad \mathbb{C}'[\mathbb{D}/x] \Downarrow \mathbb{D}'}{\mathbb{C}\mathbb{D} \Downarrow \mathbb{D}'}$$

$$\frac{}{\lambda x.\mathbb{C} \Downarrow \lambda x.\mathbb{C}}$$

What is more, for this definition to be useful we would like to be sure that hole-filling and the evaluation relation commute. We could just knuckle down and prove this, but the point we wish to make in this section is that this will *always* work for syntax oriented definitions, by virtue of the representation of contexts. To see why this is so, we will need to be more formal about the rules which make up such inductive definitions.

Formal rules

In order give a precise meaning of rules such as those above we switch to the second-order syntax for terms and introduce a syntax for *meta terms*. For the terms of a given language, fix a countable set of *metavariables Mvar* ranged over by $\mathcal{X}$, $\mathcal{Y}$, etc. Metavariables will range over both terms and abstractions of terms, and will be used to formalise rules such as those above. Metavariables will be assumed disjoint from hole variables and ordinary variables. Just as for hole variables, with each metavariable $\mathcal{X}$, we associate an arity which is a natural number. The idea is that metavariables of arity 0 will range over terms, while meta variables of higher arity will range over abstractions. Value metavariables always have arity 0.

Definition 3.1 *To define the meta-terms for a given language we define an indexed set of meta-abstractions* $\{MT_i\}_{i \geq 0}$*, (ranged over by* $\mathcal{M}$*,* $\mathcal{N}$*, etc.).* MT_0 *are the meta-terms proper, used to denote terms in formal definitions. For each* $k > 0$*,* MT_k *is the set of meta-terms which represent k-variable abstractions of terms. The raw syntax of meta-terms follows the syntax of terms, with the exception of a meta-application operator, which appears in the form* $\mathcal{X} \cdot (\mathcal{M}_1, \ldots, \mathcal{M}_n)$*. These sets are given inductively by the following*

rules:

$$\frac{}{x \in MT_0} \qquad \frac{}{\mathcal{X} \in MT_n}\ \text{arity}(\mathcal{X}) = n$$

$$\frac{\mathcal{M} \in MT_0}{(x_1, \ldots, x_n)\mathcal{M} \in MT_n}$$

$$\frac{\mathcal{M}_1 \in MT_0 \cdots \mathcal{M}_n \in MT_0}{\mathcal{X} \cdot (\mathcal{M}_1, \ldots, \mathcal{M}_n) \in MT_0}\ \text{arity}(\mathcal{X}) = n$$

$$\frac{\mathcal{M}_1 \in MT_{k_1} \ldots \mathcal{M}_n \in MT_{k_n}}{\boldsymbol{p}(\mathcal{M}_1, \ldots, \mathcal{M}_n) \in MT_0}\ \text{arity}(\boldsymbol{p}) = (k_1 \ldots k_n)$$

One can easily see that meta-terms include the terms of the language. Meta-terms are used to formalise syntax-oriented definitions. The rules above can be formalised as:

$$\frac{\mathcal{X} \Downarrow \lambda\mathcal{X}_1 \quad \mathcal{X}_1 \cdot (\mathcal{Y}) \Downarrow \mathcal{Z}}{\boldsymbol{apply}(\mathcal{X}, \mathcal{Y}) \Downarrow \mathcal{Z}}$$

$$\frac{}{\lambda\mathcal{X}_1 \Downarrow \lambda\mathcal{X}_1}$$

A *raw instance* of a rule is obtained by applying a substitution to the metavariables in a rule. A substitution replaces metavariables by term-abstractions of the corresponding arity. For example, the substitution $\sigma = [{}^{(x)xx}/_{\mathcal{X}_1}]$ applied to the rule-schema (which incidentally has zero premises) $\lambda\mathcal{X}_1 \Downarrow \lambda\mathcal{X}_1$ gives $\lambda(x)xx \Downarrow \lambda(x)xx$ The *valid* instances (usually just called rule-instances) are defined inductively in the obvious way as a the raw instances for which each premise is a conclusions for some valid instance. There may be other side-conditions, e.g., that all instances involve only closed terms.

Extending Instances to Contexts

Any collection of rule schemas can thus be viewed as inductively generating certain sets. Lifting these definitions to contexts is now trivial: *simply allow the instances of a rule to contain hole variables.* In other words we allow substitution instances of a rule to replace metavariables by contexts (or abstractions of contexts). We will call such an instance a *context rule instance.*

Let us consider a concrete example. Given the formal rule for beta-reduction:

$$(\lambda\mathcal{X}_1)\mathcal{Y} \rightarrow_\beta \mathcal{X}_1 \cdot (\mathcal{Y})$$

where for the sake of readability we have written application in the usual implicit form, we can construct a rule instance by applying the substitution $[{}^{(x)\xi(x),\ \xi(x)}/_{\mathcal{X}_1,\ \mathcal{Y}}]$ which gives:

$$(\lambda(x)\xi(x))\ \xi(x) \rightarrow_\beta \xi(\xi(x))$$

Generalising "evaluation/reduction commutes with hole filling"

The generalisation of the idea that "evaluation/reduction commutes with hole filling" is that filling the holes in a valid rule-instance yields a valid rule instance.

Theorem 3.2 *Let $\frac{P}{c}$ denote a formal rule with a set of premises P and a conclusion c. Let σ be some substitution of terms for metavariables such that $(\frac{P}{c})\sigma$ is a generalised instance of the rule. Now suppose that τ is a hole-filling (a substitution of hole-variables for term abstractions of the corresponding arity). Then the following are identical rule-instances:*

$$\left(\left(\frac{P}{c}\right)\tau\right)\sigma \equiv \left(\frac{P}{c}\right)(\sigma\tau)$$

where $(\sigma\tau)$ denotes the application of substitution τ to the range of σ.

Proof. It is easy to see that valid rule-instances are closed under substitution, and hence that $((\frac{P}{c})\sigma)\tau$ is a valid rule instance. Since metavariables and hole variables are distinct, then the equivalence above follows immediately from the substitution lemma. □

Other Syntactic Categories

The idea of extending definitions to work over contexts is widely applicable. Definitions in operational semantics often involve the construction of several syntactic categories which either contain terms or restrict the set of terms in some way. For example,

- The definition of *configurations* in an SOS-style semantics or in the definition of an abstract machine containing e.g. *stacks* of terms or *environments* (finite mappings from variables to terms). The definition of a rule-instance is essentially the same, and so there are no problems in allowing instances to contain hole-variables.
- The definition of particular subsets of terms, e.g., the *values* in a call-by-value functional language $V ::= \mathit{constant} \mid \langle V_1, V_2\rangle \mid \lambda x.M \mid \ldots$ can be lifted to "value contexts" in the obvious way; value metavariables used in computation rules must then be instantiated with value contexts.
- A popular style of small-step operational semantics is to use *evaluation contexts* to describe a deterministic reduction strategy. An evaluation context, usually specified by a grammar, is a context with exactly one hole. For example, the evaluation contexts for a call-by-value lambda calculus with strict pairing and left-to-right evaluation might be specified by

  $$E ::= [\,] \mid E\,M \mid V\,E \mid \langle E, M\rangle \mid \langle V, E\rangle \mid \ldots$$

  The evaluation rules can then be specified by e.g.

  $$E[(\lambda x M)V] \to E[M[{}^{V}/_{x}]]$$

  Formalising this style of definition presents no problems either: evaluation contexts (for this language at least) are just abstractions of one variable

(the hole). Note that in this particular example there are three definitions which need to be generalised: values, evaluation contexts and the evaluation rules themselves. The only precaution is to treat the hole variable in the definition of evaluation contexts (which in this example can be taken to have arity zero) as being distinct from all other hole variables.

4 Applications

A typical application of "direct reasoning about contexts" is to prove properties about a contextually-defined equivalence relation. Examples of this style of "direct" reasoning can be found in e.g., [MT91,AMST97]; using the approach described here we can make such arguments rigorous with almost no overhead in building a language-specific context calculus. We used this technique for several of the results described in [San97], where various theorems about operational preorders are established for any language whose operational semantics rules fit a certain *rule format*.

In [MS98,Mor98] the approach described in this article is used to establish a *context lemma* for call-by-need lambda calculi (in the latter including a form of fair nondeterminism); in these applications the semantics is based on an abstract machine, rather than a term-based computation model.

Most applications of "context evaluation" build upon a small-step operational semantics of some kind. This is natural since the larger the computation step which is used, the less likely that the computation step can be applied to a context. In the remainder of this article we consider an example application where a large-step semantics still yields some useful computations on contexts.

4.1 A Unique Fixed-Point Induction Theorem

A well-known proof technique in e.g. process algebra involves syntactically characterising a class of recursion equations which have a unique solution. Knowing that a recursive definition has a unique fixed point means that one can prove equivalence with a recursively defined entity by showing that the recursion equation is satisfied.

The usual syntactic characterisation is that *guarded recursion*: if recursive calls are syntactically "guarded" by an observable action then the fixed-point of the definition is unique.

We illustrate related notion of *guarded context* for a lazy lambda-calculus with constants, and show – with the help of context evaluation – that guarded contexts enjoy a unique fixed-point property. The process calculus notions of guardedness are something of a panacea when it comes to reasoning about recursion. Although the functional notion of guardedness falls a long way short of this, there are still some interesting instances.

We will take an extension of the lazy lambda calculus [Abr90]: The syntax

of expressions (M, N etc) is as follows:

$$\begin{array}{lll} M ::= x \mid M\,N \mid \lambda x.M & & \text{(Var; Apply; Lambda)} \\ \quad \mid \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{c}_1(\bar{x}_1) \Rightarrow M_1 \ldots \mathbf{c}_n(\bar{x}_n) \Rightarrow M_n\} & & \text{(Case)} \\ \quad \mid \mathbf{c}(\bar{M}) & & \text{(Constructors)} \end{array}$$

Constructors have a fixed arity (≥ 0), and we implicitly assume that instances of constructor expressions and patterns always respect the arity. We will use a deductive ("big-step") style of semantics;

The results of computations are *weak head-normal forms* (WHNF): either constructor terms $\mathbf{c}(\bar{L})$ or lambda-abstractions $\lambda x.M$. Let V, W range over WHNF's. Now we define the *convergence* relation between closed terms by the evaluation rules in figure 1. As usual we write $M{\Downarrow}$ to mean $\exists N.\, M{\Downarrow}N$ Let

$$\frac{}{\lambda x.M{\Downarrow}\lambda x.M} \qquad \frac{}{\mathbf{c}(\bar{M}){\Downarrow}\mathbf{c}(\bar{M})} \qquad \frac{M{\Downarrow}\lambda x.M \quad M[{}^{M_2}/_{x}]{\Downarrow}V}{M_1\ M_2{\Downarrow}V}$$

$$\frac{L{\Downarrow}\mathbf{c_i}(\bar{L}) \quad M_i[{}^{\bar{L}}/_{\bar{x}_i}]{\Downarrow}V}{\mathbf{case}\ L\ \mathbf{of}\ \{\cdots \mathbf{c_i}(\bar{x}_i) \Rightarrow M_i \cdots\} \Downarrow V}$$

Fig. 1. Convergence relation

$\underset{\sim}{\sqsubseteq}$ denote the operational preordering defined by $M \underset{\sim}{\sqsubseteq} N$ if and only if for all conventional contexts C such that $C[M]$ and $C[N]$ are closed expressions, then $C[M]{\Downarrow}$ implies $C[N]{\Downarrow}$. Let $\cong$ denote the corresponding equivalence relation.

Now we are in a position to define the guarded contexts:

Definition 4.1 *The* guarded contexts, $\mathbb{G}$, *are contexts containing at most one hole variable, and given by the following grammar:*

$$\mathbb{G} ::= M \mid \mathbf{c}(\mathbb{H}_1, \ldots, \mathbb{H}_n) \mid \lambda x.\mathbb{H} \mid \mathbf{case}\ L\ \mathbf{of}\ \mathbf{c_1}(\bar{x}_1) \Rightarrow \mathbb{G}_1 \ldots \mathbf{c_n}(\bar{x}_n) \Rightarrow \mathbb{G}_n$$

$$\mathbb{H} ::= \xi(\bar{M}) \mid \mathbb{G}$$

Since guarded contexts have just one hole variable, we will write $\mathbb{G}[(\bar{x})M]$ to denote $\mathbb{G}[{}^{(\bar{x})M}/_{\xi}]$.

Theorem 4.2 (Unique Fixed Point) *For all expressions M, N where* $\textsc{fv}(M) \cup \textsc{fv}(N) \subseteq \bar{x}$, *the following proof rule is valid:*

$$\frac{M \cong \mathbb{G}[(\bar{x})M] \qquad N \cong \mathbb{G}[(\bar{x})N]}{M \cong N}$$

Using the justification given in the previous section, we tacitly extend the syntactic categories and definitions of one-step reduction and of convergence to allow the occurrence of hole variables. We will let $\mathbb{E}$ range over evaluation

contexts containing hole variables; $\mathbb{V}$ and $\mathbb{W}$ over weak head normal forms with holes, and we extend the definitions of one-step reduction and of convergence to these extended syntactic categories.

The main point of the example is that the proof of the above theorem is facilitated by the following property of guarded contexts:

Lemma 4.3 *For all guarded contexts $\mathbb{G}$ and all abstractions $(\bar{x})M$ such that $\mathbb{G}[(\bar{x})M]$ is closed,*

$$\exists V.\, \mathbb{G}[(\bar{x})M]{\Downarrow}V \iff \exists \mathbb{V}.\, \mathbb{G}{\Downarrow}\mathbb{V}$$

where $\mathbb{V}$ is a guarded context (i.e. either of the form $\lambda x.\mathbb{H}$ or $\mathbf{c}(\mathbb{H}_1, \ldots, \mathbb{H}_n)$

Proof. The ($\Leftarrow$) direction follows easily from Theorem 3.2, since it follows from $\mathbb{G}{\Downarrow}\mathbb{V}$ that $\mathbb{G}[(\bar{x})M]{\Downarrow}\mathbb{V}[(\bar{x})M]$. For the ($\Rightarrow$) direction, we assume that $\mathbb{G}[(\bar{x})M]{\Downarrow}V$ and proceed by rule induction to show that $\exists \mathbb{V}.\, \mathbb{G}{\Downarrow}\mathbb{V}$. We proceed by cases according to the structure of $\mathbb{G}$.

Case $\mathbb{G} \equiv M$: then $\mathbb{V}$ is simply V.

Case $\mathbb{G} \equiv \lambda x.\mathbb{H}$: then $\mathbb{V}$ is just $\lambda x.\mathbb{H}$. The case for constructors follows similarly.

Case $\mathbb{G} \equiv$ case L of $\{\mathbf{c_1}(\bar{x}_1) \Rightarrow \mathbb{G}_1 \cdots \mathbf{c_n}(\bar{x}_n) \Rightarrow \mathbb{G}_n\}$: then the rule for case evaluation provides the following inductive hypotheses: $L{\Downarrow}\mathbf{c_i}(\bar{L})$ for some $\mathbf{c_i}$, and $\mathbb{G}_i[\bar{L}/\bar{x}_i]{\Downarrow}\mathbb{V}$ for a guarded value context $\mathbb{V}$. The latter case relies on the observation that guarded contexts are closed under substitution. From the case evaluation rule we conclude that $\mathbb{G}{\Downarrow}\mathbb{V}$.

□

We sketch how the proof of the can be completed using the technique of "simulation up to" (for this particular language this technique is described in [San96], and is a simple adaptation of the (bi)simulation-style proof method).

Definition 4.4 (Simulation up to $\underset{\sim}{\sqsubset}$) *A relation $\mathcal{R}$ is a* simulation up to $\underset{\sim}{\sqsubset}$ *if for all M, N, whenever $M\ \mathcal{R}\ N$, then for all closing substitutions σ, if $M\sigma{\Downarrow}V$ then $N\sigma{\Downarrow}W$ for some W such that one of the following two conditions hold:*

(i) *$V \equiv c(M_1 \ldots M_n)$, $W \equiv c(N_1 \ldots N_n)$ and $M_i \underset{\sim}{\sqsubset}; \mathcal{R}; \underset{\sim}{\sqsubset} N_i, i \in 1 \ldots n$*

(ii) *$V \equiv \lambda x.M'$ and $W \equiv \lambda x.N'$ and $M' \underset{\sim}{\sqsubset}; \mathcal{R}; \underset{\sim}{\sqsubset} N'$.*

Proposition 4.5 *If $\mathcal{R}$ is a simulation up to $\underset{\sim}{\sqsubset}$ then $\mathcal{R} \subseteq \underset{\sim}{\sqsubset}$*

Now we can complete the proof of the theorem by constructing a suitable relation and showing that it is a simulation up to $\underset{\sim}{\sqsubset}$ (equivalence follows by symmetry of the argument).

Suppose that $M \cong \mathbb{G}_0[(\bar{x})M]$ and $N \cong \mathbb{G}_0[(\bar{x})N]$. We can assume without loss of generality that $\mathrm{FV}(\mathbb{G}_0) \subseteq \bar{x}$. We will show that

$$R \stackrel{\text{def}}{=} \{\mathbb{G}[(\bar{x})M], \mathbb{G}[(\bar{x})N] \mid \mathrm{FV}(\mathbb{G}) \subseteq \bar{x}\}$$

is a simulation up to $\underset{\sim}{\sqsubset}$. Suppose that $\mathbb{G}[(\bar{x})M]\sigma{\Downarrow}$. Since $(\bar{x})M$ is a closed

abstraction, $\mathbb{G}[(\bar{x})M]\sigma \equiv \mathbb{G}\sigma[(\bar{x})M]$. Since $\mathbb{G}\sigma$ is also a guarded context by lemma 4.3 we have that either

(i) $\mathbb{G}\sigma\Downarrow \lambda y.\mathbb{H}_0$ for some $\mathbb{H}_0$, or

(ii) $\mathbb{G}\sigma\Downarrow \mathbf{c}(\mathbb{H}_1 \ldots \mathbb{H}_n)$ for some constructor $\mathbf{c}$ and some $\mathbb{H}_1 \ldots \mathbb{H}_n$.

By theorem 3.2 we know that $\mathbb{G}[(\bar{x})N]\sigma\Downarrow$, and it remains to show that, in each respective case that $\mathbb{H}_i[(\bar{x})M] \underset{\sim}{\sqsubset}; R; \underset{\sim}{\sqsubset} \mathbb{H}_i[(\bar{x})N]$. By definition each $\mathbb{H}_i$ is either a guarded context or a hole. In the former case we have immediately that $\mathbb{H}_i[(\bar{x})M]\ R\ \mathbb{H}_i[(\bar{x})N]$ and so we are done by reflexivity of $\underset{\sim}{\sqsubset}$. In the latter case $\mathbb{H}_i$ is of the form $\xi(\bar{L})$ so we have that

$$\xi(\bar{L})[(\bar{x})M] \equiv M[\bar{L}/\bar{x}] \cong \mathbb{G}_0[(\bar{x})M][\bar{L}/\bar{x}]\ R\ \mathbb{G}_0[(\bar{x})N][\bar{L}/\bar{x}] \cong N[\bar{L}/\bar{x}]$$

as required.

Example

We leave the following example as an exercise which is easily proved using the unique fixed-point rule:

$$\texttt{map f (iterate f x)} \cong \texttt{iterate f (f x)}$$

where `map :: (a -> b) -> [a] -> [b]` and `iterate :: (a -> a) -> a -> [a]` are the usual Haskell recursive functions.

Acknowledgements

Thanks to Andy Moran for many helpful comments and discussions, and to Søren Lassen for suggestions for improvements to an earlier draft.

References

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

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Inf.*, 11(1):31–55, January 1978.

[HO98] M. Hashimoto and A. Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, 1998.

[LF96] S. Lee and D. Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, May 1996.

[Mas96] I. Mason. Parametric computation. In *CATS'96*, 1996.

[MN95] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. Technical Report Sep26-1, Technical University of Munich, September 1995.

[Mor98] A. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb.* PhD thesis, Department of Computing Sciences, Chalmers, Sweden, September 1998.

[MS98] A. Moran and D. Sands. A context lemma for call-by-need. Working Note. Jan 1998. Revised May, 1998.

[MT91] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 199–208. ACM Press, June 1988.

[Pit94] Andrew M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, December 1994.

[San95] D. Sands. Total correctness by local improvement in program transformation. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, January 1995. ACM Press.

[San96] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):175–234, March 1996. Extended version of [San95].

[San97] D. Sands. From sos rules to proof principles: An operational metatheory for functional languages. In *24th ACM SIGPLAN-SIGACT Symposium on Principles on Programming Languages (POPL'97)*. ACM Press, 1997.

[Tal92] C. Talcott. Towards a theory of binding structures: An abstract algebra. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 201–215, London, May22–25 1992. Springer Verlag.

[Tal93] C. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, April 1993.

[Tal97] C. Talcott. Reasoning about functions with effects. In A. Gordon and A. Pitts, editors, *Higher-Order Operational Techniques in Semantics*. Cambridge University Press, 1998.