ELSEVIER Science of Computer Programming 46 (2003) 99–135

Science of Computer Programming

www.elsevier.com/locate/scico

Erratic Fudgets: a semantic theory for an embedded coordination language[☆]

Andrew Moran[a,*], David Sands[b], Magnus Carlsson[b]

[a]Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Beaverton, OR 97006, USA

[b]Department of Computing Science, Chalmers University of Technology and the University of Göteborg, S-412 96 Göteborg, Sweden

Abstract

The powerful abstraction mechanisms of functional programming languages provide the means to develop domain-specific programming languages within the language itself. Typically, this is realised by designing a set of combinators (higher-order reusable programs) for an application area, and by constructing individual applications by combining and coordinating individual combinators. This paper is concerned with a successful example of such an embedded programming language, namely Fudgets, a library of combinators for building graphical user interfaces in the lazy functional language Haskell. The Fudget library has been used to build a number of substantial applications, including a web browser and a proof editor interface to a proof checker for constructive type theory. This paper develops a semantic theory for the non-deterministic stream processors that are at the heart of the Fudget concept. The interaction of two features of stream processors makes the development of such a semantic theory problematic:

(i) the sharing of computation provided by the lazy evaluation mechanism of the underlying host language, and
(ii) the addition of non-deterministic choice needed to handle the natural concurrency that reactive applications entail.

We demonstrate that this combination of features in a higher-order functional language can be tamed to provide a tractable semantic theory and induction principles suitable for reasoning about contextual equivalence of Fudgets. © 2002 Elsevier Science B.V. All rights reserved.

[☆] This work was conducted while the first author was employed as a post-doctoral researcher at the Department of Computer Science, Chalmers University of Technology and University of Göteborg.

[*] Corresponding author.

E-mail addresses: moran@cse.ogi.edu, moran@galois.com (A. Moran), dave@cs.chamers.se (D. Sands), magnus@cs.chalmers.se (M. Carlsson).

1. Introduction

Fudgets are a collection of combinators for developing graphical user interfaces in the lazy functional programming language Haskell. Typical fudget programs exhibit the clean separation between *computation* (the standard Haskell code) and *coordination* (the management of the user interface) that is the hallmark of the so-called coordination languages. A small fudget program can be found in Appendix A.

This paper is concerned with the semantics of fudgets, or rather, with the semantics of the high-level language of stream processors that lie at the heart of fudgets. A stream processor is a higher-order process that communicates with its surroundings through a single input stream and a single output stream.

The implementation of stream processors used in the Fudget library could serve as a semantic basis for formal reasoning about stream processor and fudget programs. But as a semantics, the current implementation itself leaves a lot to be desired. For one thing, it is implemented deterministically, and as a consequence many intuitively correct transformation laws, such as symmetry of parallel composition, are in fact unsound. We aim to develop a more abstract semantics of stream processors, which better captures their conceptually concurrent, non-deterministic nature, and which relates to concrete implementations by a notion of refinement.

We build our semantic theory for fudgets on top of a lazy functional language with erratic non-deterministic choice. Erratic choice is a simple internal choice operator. In modelling non-determinism with erratic choice, we are choosing to ignore aspects of fairness. This can be viewed as a shortcoming of our model since we can neither express a bottom-avoiding merge operator for streams, nor prevent starvation when merging streams.

Despite the simplicity of erratic choice, it exhibits a non-trivial interaction with lazy evaluation. By *lazy evaluation* we mean the standard *call-by-need* implementation technique for non-strict languages whereby the argument in each instance of a function application is evaluated at most once. In the standard semantic theories of functional languages this evaluation mechanism (also known as graph reduction) is modelled by the much simpler *call-by-name* evaluation, in which an argument is evaluated as many times as it is required. In the presence of non-determinism, there is an observable difference between these evaluation mechanisms, and so a semantic theory must take care to model these details accurately. Consider the following desirable equivalence:

$$2 * x \cong x + x.$$

This does not hold in a call-by-name theory when non-determinism is present. To see this, consider the situation when x is bound to $3 \oplus 4$, where $\oplus$ is erratic choice. The right-hand side may evaluate to 6, 7, or 8, but the left-hand side may result in 6 or 8, but *not* 7. However, the equivalence *does* hold in a call-by-need theory for non-determinism, since the result of $3 \oplus 4$ will be shared between both occurrences of x in the right-hand side. So while the call-by-need theory may lack arbitrary β-reduction, it does retain other useful equivalences.

1.1. Contributions

We develop an operational theory for $\Lambda^{\oplus}_{\text{NEED}}$, a call-by-need lambda calculus with recursive lets, constructors, case expressions, and an erratic choice operator. The stream processor calculus of [9] is presented and a translation into $\Lambda^{\oplus}_{\text{NEED}}$ is given. We show that congruences and reductions in the stream processor calculus are equivalences and refinements, respectively, in the theory. Some specific contributions are:

- A *context lemma* for call-by-need and non-determinism, meaning we can establish equivalence and refinement by considering just computation in a restricted class of contexts, the evaluation contexts;
- a rich inequational theory for call-by-need and non-determinism;
- a *unique fixed-point induction* proof rule, which allows us to prove properties of recursive programs, and
- validation of congruences and reductions in the stream processor calculus proposed by Carlsson and Hallgren [9].

1.2. Organisation

The remainder of the paper is organised as follows. We begin with a discussion of related work in Section 2. Then Section 3 introduces stream processors and the stream processor calculus. The language $\Lambda^{\oplus}_{\text{NEED}}$ is presented, along with the translation from stream processor calculus into $\Lambda^{\oplus}_{\text{NEED}}$. The operational semantics of $\Lambda^{\oplus}_{\text{NEED}}$ is presented in Section 4. This is used to define notions of *convergence* and *divergence*, leading to contextual definitions of *refinement* and *observational equivalence*. The context lemma is then stated, and a number of laws from the inequational theory are presented. An equivalence sensitive to the cost of evaluation is defined, leading to the statement of the unique fixed-point induction rule. Section 5 deals with the correctness of the translation of stream processors into $\Lambda^{\oplus}_{\text{NEED}}$. It begins with an example of the use of unique fixed-point induction; the rest of the proofs has a simple calculational flavour. Section 6 summarises the technical development of the theory and presents proofs of the main theorems. Section 7 concludes, and we discuss of future avenues of research.

2. Related work

2.1. Fudget calculi

The Fudgets thesis [9] introduces a calculus of stream processors, which is presented in the CHAM-style as a collection of structural congruences and a set of reduction rules. This calculus has much in common with non-deterministic dataflow languages (see e.g., [22,26,39,40,41]), apart from the fact that it is higher-order (i.e., stream processors may be sent as messages).

Independent of this work, Colin Taylor has developed theories of “core fudgets” using two approaches: an encoding into the pi calculus, and a direct operational semantics of “core fudgets” [55,56]. Taylor is somewhat dismissive of his earlier pi-calculus approach, since (like many pi calculus semantics) it suffers from the rather low-level nature of the encoding. In particular, he notes that “the pi calculus encodings are often large and non-intuitive”. Taylor’s direct approach [56] is based on a labelled transition system in which the core fudget combinators such as parallel composition are given a direct operational interpretation. The theory which is built on top of this is based on higher-order bisimulation. Taylor’s theory can be seen to verify many of the congruence rules of Carlsson and Hallgren’s calculus. The difference with our theory stems from the fact that Taylors calculus completely abstracts away the underlying functional language. This abstraction presupposes that the semantics of the combinators is orthogonal to the underlying functional computations. As we have argued, in the presence of call-by-need computation, there is an observable interaction between the sharing present in the functional computation, and the non-determinism implied by the concurrency of parallel composition.

2.1.1. *Other functional GUIs*

There are at least two other functional approaches to GUI in a functional language for which an underlying theory exists (for a “core” language). The toolkit eXene, by Reppy and Gansner [17] is written on top of Concurrent ML (CML) [45], a multi-threaded extension to Standard ML. A number of authors have considered the semantics for a fragment of CML, e.g. [14,15,37].

The GUI library *Haggis* [15] is built upon *Concurrent Haskell* [23]. Fudgets have been implemented in Haggis, and a theory for concurrent Haskell has been outlined in [23]. This suggests that a theory of fudgets could be developed using this route.

2.1.2. *Non-determinism in functional languages*

The standard denotational approach to modelling non-deterministic constructs is to use a *powerdomain* construction (see e.g., [8,10,19,20,43,51,52,53]), domain-theoretic analogues of the powerset operator. The Plotkin, or convex, powerdomain [43] models erratic choice very well, but like all powerdomain semantics ignores the issue of sharing. An alternative to erratic choice is McCarthy’s *amb*, a bottom-avoiding non-deterministic operator which embodies a certain kind of fairness, but the denotational approach has well-documented problems modelling McCarthy’s *amb* (see e.g., [34]). The only serious attempt at a denotational semantics for McCarthy’s *amb*, that due to Broy [8], is developed for a first-order language only, and does not consider sharing.

Lassen and Moran [29] also consider McCarthy’s *amb*, in the context of a call-by-name lambda calculus. Though able to construct a powerful equational theory for that language (including proof rules for recursion) by taking an operational approach, the language in question is not call-by-need. The choice operator considered in this article is simpler (and arguably less useful) and thus we are able to build on the semantic theory for deterministic call-by-need developed in [36].

Others have studied lambda calculi with various forms of non-determinism, parallelism, and message-passing (see e.g. [7,11,12,13,20,21,33,49]), but without also describing sharing, which is a crucial element of this work. Kutzner and

Schmidt-Schauß [27] recently presented a reduction-calculus for call-by-need with erratic non-determinism based on an extension of the call-by-need lambda calculus of [4]. We believe that our theory of equivalence subsumes the calculus of Kutzner and Schmidt-Schauß's in the same way that the deterministic part is subsumed by the improvement theory of [36]. Kutzner and Schmidt-Schauß's theory is inadequate for our purposes. Firstly, it only considers a pure lambda calculus with no constructors. Secondly, it does not consider cyclic recursion (so it cannot, for example, express the cyclic Y-combinator). Thirdly, and most importantly, it does not include any proof principles for reasoning about recursive definitions. Our semantic theory has none of these deficiencies.

3. Essence of Fudgets

The Fudget library is implemented in the purely functional language Haskell [42], in which all computations are deterministic. Conceptually, a fudget program should be regarded as a set of concurrent processes exchanging messages with each other and the outside world. In a style that is typical for a higher-order, functional programming language, combinators in the Fudget library are used to express high-level communication patterns between fudgets.

Taking an abstract view, the essence of the fudget concept is the *stream processor*, which has truly parallel and non-deterministic properties.

3.1. The stream processor calculus

A stream processor can be seen as a process that consumes messages in an input stream, producing messages in an output stream. However, the streams are not manipulated directly by the process, only the messages are (a more appropriate name would actually be *message processor*). There are seven ways of constructing a stream processor:

$$\begin{array}{rcll} S,T,U & ::= & S\,!\,T & (\mathit{Put}) \\ & | & x\,?\,S & (\mathit{Get}) \\ & | & S \lessdot T & (\mathit{Feed}) \\ & | & S \ll T & (\mathit{Serial}) \\ & | & S \mid T & (\mathit{Parallel}) \\ & | & \ell\, S & (\mathit{Loop}) \\ & | & x & (\mathit{Var}) \end{array}$$

The first three forms are stream processors that deal with input and output of single messages. $S\,!\,T$ outputs the message S and then becomes the stream processor T. $x\,?\,S$ waits for a message, and binds the variable x to that message (in S) and then becomes S. $S \lessdot T$ feeds the message T to S, that is, T is the first message that S will consume on its input stream.

The following three constructions are used for coordinating the message flow between stream processors (see also Fig. 1). $S \ll T$ connects the output stream of T to the input

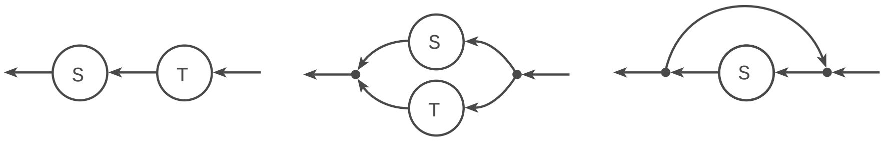


Fig. 1. Composing stream processors: serial and parallel composition, and loop.

stream of S, and thus acts as a serial composition. $S \mid T$ puts the stream processors S and T in parallel: consumed messages are broadcast to both S and T, and messages produced by S and T are merged into one stream. Feedback can be introduced in a network of stream processors by $\ell\, S$, which feeds all messages output from S back to its input.

Finally, variables (that is, messages) can be used as stream processors, which make the calculus higher order: stream processors can be sent as messages from one stream processor and "plugged in" in another stream processor.

The stream processor calculus defines a number of congruence rules to be used freely in order to enable application of the reaction rules to follow. In addition to the commutativity of $\mid$ and the associativity of $\mid$ and $\ll$, we have the following congruences (where M and N range over stream processors used as messages):

$$
\begin{aligned}
(S \ll T) \lessdot M &\sim S \ll (T \lessdot M) \quad (\textit{Input-}\ll),\\
S \ll (M\,!\,T) &\sim (S \lessdot M) \ll T \quad (\textit{Internal-}\ll),\\
(M\,!\,S) \ll T &\sim M\,!\,(S \ll T) \quad (\textit{Output-}\ll),\\
(S \mid T) \lessdot M &\sim (S \lessdot M) \mid (T \lessdot M) \quad (\textit{Input-}\mid),\\
(M\,!\,S) \lessdot N &\sim M\,!\,(S \lessdot N) \quad (\textit{Output-}\lessdot),\\
(x\,?\,S) \lessdot M &\sim S[M/x] \quad (\textit{Input-}?).
\end{aligned}
$$

When used in a left-to-right fashion, these rules mirror propagation of messages in a network of stream processors.

Whereas the congruence rules in the last section can be freely used in any direction without changing the behaviour of a stream processor, the reaction rules are irreversible, and introduce non-determinism. The reason is that by applying a reaction rule, we make a choice of *how the message streams should be merged*. There are two places where merging occurs, in the output from a parallel composition, and in the input to a stream processor in a loop.

$$
\begin{aligned}
(M\,!\,S) \mid T &\to M\,!\,(S \mid T) \quad (\textit{Output-}\mid),\\
\ell(M\,!\,S) &\to M\,!\,\ell\,(S \lessdot M) \quad (\textit{Output-}\ell),\\
(\ell\, S) \lessdot M &\to \ell\,(S \lessdot M) \quad (\textit{Input-}\ell).
\end{aligned}
$$

Remember that we defined $\mid$ to be commutative, which means that we only need one output rule for $\mid$. Note also that there is one congruence rule that one might find tempting to include, which we do not:

$$(S \mid T) \lll U \sim (S \lll U) \mid (T \lll U).$$

On the left-hand side, S and T will receive the same input stream. On the right-hand side, U is duplicated, and there is no guarantee that the two occurrences produce the same stream of messages, so S and T may be passed different input streams.[1]

The stream processor equivalences and reduction rules can be viewed as a *chemical abstract machine*-style operational semantics [6] for the language of stream processors.

With this as a basis, one could develop a theory of equivalence, e.g., based on bisimilarity, with which to further investigate the language. This is essentially the approach taken by Taylor [56] (using a more conventional SOS-style definition). However, as we have mentioned, this approach ignores interactions with the features of the language in which the stream processors are embedded, namely a lazy functional language. Instead of considering the stream processor calculus in isolation, we show how it can be realised by simple encodings into a call-by-need functional language with an additional erratic non-deterministic choice operator. By developing the theory of contextual equivalence for this language, we will show that the laws of the calculus are sound with respect to this implementation, and that the reduction rules are refinements. First we must introduce our language and its operational semantics.

3.2. Implementing the stream processors

We will embed the stream processors into an untyped lambda calculus with recursive let bindings, structured data, case expressions, and a non-deterministic choice operator. We work with a restricted syntax in which arguments to functions (including constructors) are always variables:

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid M\,x \mid c\vec{x} \\ & \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \\ & \mid \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \\ & \mid M \oplus N. \end{aligned}$$

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, e.g., [24,25], and in [30,50]. We call the language $\Lambda^{\oplus}_{\text{NEED}}$.

All constructors have a fixed arity, and are assumed to be saturated. By $c\vec{x}$ we mean $cx_1 \cdots x_n$. The only values are lambda expressions and fully applied constructors. Throughout, x, y, z, and w will range over variables, c over constructor names, and V and W over values. We will write $\mathsf{let}\ \{\vec{x}=\vec{M}\}\ \mathsf{in}\ N$ as a shorthand for

$$\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N,$$

[1] This rule is already a classic; it has been mistakenly included by both Moran [34] and Taylor [56]. Taylor retracts the rule in his Ph.D. thesis [55].

where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (so our lets are recursive). Similarly we write $\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \rightarrow N_i\}$ for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\vec{x}_1 \rightarrow N_1|\cdots|c_m\,\vec{x}_m \rightarrow N_m\},$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i \rightarrow N_i\}$.

For examples, working with a restricted syntax can be cumbersome, so it is sometimes useful to lift the restriction. Where we do this it should be taken that

$$M\,N \equiv \mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x, \quad x \text{ fresh},$$

whenever N is not a variable. Similarly for constructor expressions. We will encode streams using the "cons" constructor, written in the Haskell-style infix as $M{:}N$ (read as M "consed onto" N). The nil-stream (the empty stream) is written as nil.

We use two kinds of definition in this section. The notation

$$f\,x_1\cdots x_n \overset{\text{def}}{=} M$$

means that $f\,x_1\cdots x_n$ should be considered as a synonym for M, and the x_i are bound in M. The notation

$$f\,x_1\cdots x_n = M$$

will be used when f occurs free in M to denote a recursive definition of f; f should be considered to be defined by a recursive let thus:

$$\mathsf{let}\ \{f = \lambda x_1,\ldots,x_n.M\}\ \mathsf{in}\ f.$$

3.2.1. Erratic merge

The parallel combination of stream processors will depend upon the ability to merge incoming streams. We define *erratic merge* thus:

$$\begin{aligned} \mathit{merge}\ \mathit{xs}\ \mathit{ys} = {} & \begin{pmatrix} \mathsf{case}\ \mathit{xs}\ \mathsf{of} \\ \quad \mathsf{nil} \ \rightarrow \mathit{ys} \\ \quad z{:}zs \rightarrow z{:}\mathit{merge}\ zs\ \mathit{ys} \end{pmatrix} \\ & \oplus \begin{pmatrix} \mathsf{case}\ \mathit{ys}\ \mathsf{of} \\ \quad \mathsf{nil} \ \rightarrow \mathit{xs} \\ \quad z{:}zs \rightarrow z{:}\mathit{merge}\ \mathit{xs}\ zs \end{pmatrix} \end{aligned}$$

Essentially, *merge* non-deterministically chooses which of its operands to evaluate first. This is not fair: it may lead to starvation of a branch (since the same stream may be chosen *ad infinitum*). Neither is it bottom-avoiding: divergence in either input stream can lead to divergence for the merge as a whole.

3.2.2. Translating stream processors

We implement stream processors by translating them into $\Lambda^{\oplus}_{\mathrm{NEED}}$. Fundamental stream processors have simple translations. All variables introduced on the right-hand sides are assumed to be fresh:

$$
\begin{aligned}
[\![x]\!] &= \lambda i\,.xi,\\
[\![M\,\mathbf{!}\,S]\!] &= \lambda i.[\![M]\!]\mathbf{:}([\![S]\!]i),\\
[\![x\,\mathbf{?}\,S]\!] &= \lambda i.\mathsf{case}\ i\ \mathsf{of}\\
&\qquad\quad \mathsf{nil} \rightarrow \mathsf{nil}\\
&\qquad\quad x\mathbf{:}xs \rightarrow [\![S]\!]xs,\\
[\![S \lessdot M]\!] &= \lambda i.[\![S]\!]([\![M]\!]\mathbf{:}i).
\end{aligned}
$$

Note that in $x\,\mathbf{?}\,S$, x may be free in S. The three combinator forms are as follows:

$$
\begin{aligned}
[\![S \lll T]\!] &= \lambda i.[\![S]\!]\ ([\![T]\!]\,i),\\
[\![S \mid T]\!] &= \lambda i.merge([\![S]\!]\,i)\,([\![T]\!]\,i),\\
[\![\ell\, S]\!] &= \lambda i.\mathsf{let}\ \{o = [\![S]\!]\ (merge\ oi)\}\ \mathsf{in}\ o.
\end{aligned}
$$

Serial composition is just function composition. We use erratic merge to merge the output streams for parallel composition, and to merge the incoming stream with the feedback stream for the loop construct.

4. The operational theory

The operational semantics is presented in the form of an abstract machine semantics that correctly describes both sharing and erratic non-determinism. Using this semantics to define notions of *convergence* and *divergence*, we define what it means for one term to be *refined* by another, and what it means for two terms to be equivalent. These definitions are contextual in nature, which makes proving refinement or equivalence difficult (since one must prove the relationship holds for *all* program contexts). However, we are able to show that one need only prove that the relationship holds for a much smaller set of contexts, the so-called *evaluation contexts*. This result is then used to validate a set of algebraic laws. Lastly, we introduce the notion of *cost equivalence*, which is a cost-sensitive version of contextual equivalence. A powerful unique fixed-point proof rule is shown to be valid for cost equivalence. It is this proof rule that will allow us to establish the properties of erratic merge required to prove the correctness of the list-based implementation of stream processors.

4.1. The abstract machine

The semantics presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [50], augmented with rules for erratic choice.

$$\langle \Gamma\{x = M\},\ x,\ S\rangle \to \langle \Gamma,\ M,\ \#x : S\rangle \qquad (\mathit{Lookup})$$
$$\langle \Gamma,\ V,\ \#x : S\rangle \to \langle \Gamma\{x = V\},\ V,\ S\rangle \qquad (\mathit{Update})$$
$$\langle \Gamma,\ M\,x,\ S\rangle \to \langle \Gamma,\ M,\ x : S\rangle \qquad (\mathit{Unwind})$$
$$\langle \Gamma,\ \lambda x.M,\ y : S\rangle \to \langle \Gamma,\ M[{}^{y}\!/\!_{x}],\ S\rangle \qquad (\mathit{Subst})$$
$$\langle \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ \mathit{alts},\ S\rangle \to \langle \Gamma,\ M,\ \mathit{alts} : S\rangle \qquad (\mathit{Case})$$
$$\langle \Gamma,\ c_j\,\vec{y},\ \{c_i\,\vec{x}_i \to N_i\} : S\rangle \to \langle \Gamma,\ N_j[{}^{\vec{y}}\!/\!_{\vec{x}_j}],\ S\rangle \qquad (\mathit{Branch})$$
$$\langle \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\ S\rangle \to \langle \Gamma\{\vec{x} = \vec{M}\},\ N,\ S\rangle \quad \vec{x} \cap \mathrm{dom}(\Gamma, S) = \emptyset \qquad (\mathit{Letrec})$$
$$\langle \Gamma,\ M \oplus N,\ S\rangle \to \langle \Gamma,\ M,\ S\rangle \qquad (\mathit{Left})$$
$$\langle \Gamma,\ M \oplus N,\ S\rangle \to \langle \Gamma,\ N,\ S\rangle \qquad (\mathit{Right})$$
$$\langle \Gamma,\ x,\ S\rangle \to \langle \Gamma,\ x,\ S\rangle \quad x \in \mathrm{dom}\, S \qquad (\mathit{Black\ Hole})$$

Fig. 2. The abstract machine semantics for non-deterministic call-by-need.

Transitions are over configurations consisting of a heap, containing bindings, the expression currently being evaluated, and a stack. The heap is a partial function from variables to terms, and denoted in an identical manner to a collection of let-bindings. The stack may contain variables (the arguments to applications), case alternatives, or *update markers* denoted by $\#x$ for some variable x. Update markers ensure that a binding to x will be recreated in the heap with the result of the current evaluation; this is how sharing is maintained in the semantics.

We write $\langle \Gamma,\ M,\ S\rangle$ for the abstract machine configuration with heap Γ, expression M, and stack S. We denote the empty heap by $\emptyset$, and the addition of a group of bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$. The stack written $b : S$ will denote a stack S with b pushed on the top. The empty stack is denoted by ε, and the concatenation of two stacks S and T by ST (where S is on top of T).

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. A configuration is *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle \Gamma,\ M,\ S\rangle$ to be closed, any free variable in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$.

The abstract machine semantics is presented in Fig. 2; we implicitly restrict the definition to well-formed configurations. There are seven rules, which can be grouped as follows. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this evaluation is finished; we update the heap with x bound to the resulting value. It is this rule that lies at the heart of laziness: all subsequent uses of x will now share this value.

Rules (*Unwind*) and (*Subst*) concern function application: rule (*Unwind*) pushes an argument onto the stack while the function is being evaluated; once a lambda expression has been obtained, rule (*Subst*) retrieves the argument from the stack and substitutes it into the body of that lambda expression.

Rules (*Case*) and (*Branch*) govern the evaluation of case expressions. Rule (*Case*) initiates evaluation of the case expression, with the case alternatives pushed onto the stack. Rule (*Branch*) uses the result of this evaluation to choose one of the branches of the case, performing substitution of the constructor's arguments for the branch's pattern variables.

Rule (*Letrec*) adds a set of bindings to the heap. The side condition ensures that no inadvertent name capture occurs, and can always be satisfied by a local α-conversion.

The evaluation of choice expressions is described by rules (*Left*) and (*Right*). This is *erratic* choice, so we just pick one of the branches.

The last rule, (*Black Hole*), concerns self-dependent expressions (such as $\mathsf{let}\ x = x\ \mathsf{in}\ x$). If we come to evaluate x when it is bound in S (i.e., there is an update marker for x on the stack S), then x is self-dependent.[2] By rewriting a self-dependent configuration to itself, we identify black holes with non-termination, which simplifies the development to follow.

Definition 1 (*Convergence*). For closed configurations $\langle \Gamma, M, S\rangle$,

$$\langle \Gamma, M, S\rangle \Downarrow^n \stackrel{\text{def}}{=} \exists \Delta, V.\langle \Gamma, M, S\rangle \to^n \langle \Delta,\ V,\ \varepsilon\rangle,$$

$$\langle \Gamma, M, S\rangle \Downarrow \stackrel{\text{def}}{=} \exists n.\langle \Gamma, M, S\rangle \Downarrow^n .$$

We will also write $M\Downarrow$ and $M\Downarrow^n$ identifying closed M with the initial configuration $\langle \emptyset,\ M,\ \varepsilon\rangle$.

Closed configurations which do not converge simply reduce indefinitely. For this language, this is the same as having arbitrarily long reduction sequences.[3]

Definition 2 (*Divergence*). For closed configurations $\langle \Gamma,\ M,\ S\rangle$,

$$\langle \Gamma, M, S\rangle \Uparrow \stackrel{\text{def}}{=} \forall k.\langle \Gamma, M, S\rangle \to^k .$$

Since the language is non-deterministic, a given expression may have many different convergent and divergent behaviours.

4.2. Program contexts

The starting point for an operational theory is usually an approximation and an equivalence defined in terms of *program contexts*. Program contexts are usually introduced as "programs with holes", the intention being that an expression is to be "plugged into" all of the holes in the context. The central idea is that to compare the behaviour of two terms one should compare their behaviour in all program contexts. We will use

[2] An alternative definition would be to check whether or not x is bound in Γ. However, the underlying technical development in Section 6 relies upon the extension of the operational semantics to *open* terms. If the side-condition was instead $x \notin \operatorname{dom} \Gamma$ then open terms would also diverge.

[3] This is due to the fact that $\to$ is finitely branching. If we were working with a fair choice like McCarthy's *amb*, this definition would not be sufficient.

contexts of the following form:

$$
\begin{aligned}
\mathbb{C},\mathbb{D} ::= {} & [\cdot] \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}\,x \mid c\,\vec{x} \\
& \mid \mathsf{let}\ \{\vec{x} = \vec{\mathbb{C}}\}\ \mathsf{in}\ \mathbb{D} \\
& \mid \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to \mathbb{D}_i\} \\
& \mid \mathbb{C} \oplus \mathbb{D}.
\end{aligned}
$$

Our contexts may contain zero or more occurrences of the hole, and as usual the operation of filling a hole with a term can cause variables in the term to become captured.

An *evaluation context* is a context in which the hole is the target of evaluation; in other words, evaluation cannot proceed until the hole is filled. Evaluation contexts have the following form:

$$
\begin{aligned}
\mathbb{E} ::= {} & \mathbb{A} \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{A} \\
& \mid \mathsf{let}\ \{\vec{y} = \vec{M}, \\
& \qquad x_0 = \mathbb{A}_0[x_1], \\
& \qquad x_1 = \mathbb{A}_1[x_2], \\
& \qquad \ldots, \\
& \qquad x_n = \mathbb{A}\} \\
& \quad \mathsf{in}\ \mathbb{A}[x_0] \\
\mathbb{A}_0 ::= {} & [\cdot] \mid \mathbb{A}x \mid \mathsf{case}\ \mathbb{A}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to M_i\}.
\end{aligned}
$$

$\mathbb{E}$ ranges over evaluation contexts, and $\mathbb{A}$ over what we call *applicative contexts*. Our evaluation contexts are strictly contained in those mentioned in the letrec extension of Ariola and Felleisen [3]: there they allow $\mathbb{E}$ to appear anywhere we have an $\mathbb{A}$. Our "flattened" definition corresponds exactly to configuration contexts (with a single hole) of the form $\langle \Gamma,\ [\cdot],\ S\rangle$.

4.3. Refinement and observational equivalence

We define refinement and observational equivalence via contexts in the following way.

Definition 3 (*Refinement*). We say that M *is refined by* N, written $M \lesssim N$, if for all $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,

$$\mathbb{C}[N]\Downarrow \implies \mathbb{C}[M]\Downarrow \quad \wedge \quad \mathbb{C}[N]\Uparrow \implies \mathbb{C}[M]\Uparrow .$$

Whenever $M \lesssim N$, then N's convergent behaviours can be matched by M and N does not exhibit divergent behaviours not already present in M. We say that M and N are *observationally equivalent*, written $M \cong N$, when $M \lesssim N$ and $N \lesssim M$.

This definition suffers from the same problem as any contextual definition: to prove that two terms are related requires one to examine their behaviour in *all* contexts. For

$$\begin{aligned}
(\lambda x.M)\, y &\cong M[{}^{y}\!/\!_{x}] && (\beta)\\
\mathsf{case}\ c_j\, \vec{y}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to M_i\} &\cong M_j[{}^{\vec{y}}\!/\!_{\vec{x}_j}] && (\mathit{case}\text{-}\beta)\\
\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathrm{D}}[x]\}\ \mathsf{in}\ \mathrm{C}[x] &\cong \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathrm{D}}[V]\}\ \mathsf{in}\ \mathrm{C}[V] && (\mathit{value}\text{-}\beta)\\
\mathrm{E}[\mathsf{case}\ M\ \mathsf{of}\ \{\mathit{pat}_i \to N_i\}] &\cong \mathsf{case}\ M\ \mathsf{of}\ \{\mathit{pat}_i \to \mathrm{E}[N_i]\} && (\mathit{case}\text{-}\mathrm{E})\\
\mathrm{E}[\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N] &\cong \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathrm{E}[N] && (\mathit{let}\text{-}\mathrm{E})\\
\mathsf{let}\ \{\vec{x} = \vec{L}, \vec{y} = \vec{M}\}\ \mathsf{in}\ N &\cong \mathsf{let}\ \{\vec{y} = \vec{M}\}\ \mathsf{in}\ N, \quad \text{if } \vec{x} \cap \mathsf{FV}(\vec{M}, N) = \emptyset && (\mathit{gc})\\
\mathsf{let}\ \{\vec{x} = \vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{y} = \vec{M}\}\ \mathsf{in}\ N &\cong \mathsf{let}\ \{\vec{x} = \vec{L}, \vec{y} = \vec{M}\}\ \mathsf{in}\ N && (\mathit{let}\text{-}\mathit{flatten})\\
\mathsf{let}\ \{x = \mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' &\cong \mathsf{let}\ \{x = \mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N, \vec{y} = \vec{L}\}\ \mathsf{in}\ N' && (\mathit{let}\text{-}\mathit{let})\\
\lambda x.\mathsf{let}\ \{\vec{y} = \vec{V}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N &\cong \mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ \lambda x.\mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N\\
& && (\mathit{let}\text{-}\mathit{float}\text{-}\mathit{val})\\
\mathsf{let}\ \{x = M, \vec{y} = \vec{N}\}\ \mathsf{in}\ x &\cong \mathsf{let}\ \{x = M, \vec{y} = \vec{N}\}\ \mathsf{in}\ M, \quad \text{if } x \notin \mathsf{FV}(M, \vec{N})\\
& && (\mathit{inline})\\
\mathsf{let}\ \{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2, \vec{z} = \vec{M}\}\ \mathsf{in}\ N &\cong \mathsf{let}\ \{\vec{x} = \vec{V}\sigma_2\sigma_3, \vec{z} = \vec{M}\sigma_3\}\ \mathsf{in}\ N\sigma_3,\\
& \sigma_1 = [{}^{\vec{y}}\!/\!_{\vec{w}}], \sigma_2 = [{}^{\vec{x}}\!/\!_{\vec{w}}], \sigma_3 = [{}^{\vec{x}}\!/\!_{\vec{y}}], && (\mathit{value}\text{-}\mathit{copy})\\
M \oplus N &\cong N \oplus M && (\oplus\text{-}\mathit{comm})\\
L \oplus (M \oplus N) &\cong (L \oplus M) \oplus N && (\oplus\text{-}\mathit{assoc})\\
\mathrm{E}[M \oplus N] &\cong \mathrm{E}[M] \oplus \mathrm{E}[N] && (\oplus\text{-}\mathrm{E})\\
M \oplus N &\lesssim M && (\oplus\text{-}\mathit{left})
\end{aligned}$$

Fig. 3. Selected laws of the equational theory.

this reason, it is common to seek to prove a *context lemma* [32] for an operational semantics: one tries to show that to prove M observationally approximates N, one only need compare their behaviour with respect to a much smaller set of contexts.

We have established the following context lemma for call-by-need and erratic choice:

Lemma 1 (Context lemma). *For all terms M and N, if for all evaluation contexts $\mathbb{E}$ such that $\mathbb{E}[M]$ and $\mathbb{E}[N]$ are closed,*

$$\mathbb{E}[N]\Downarrow \implies \mathbb{E}[M]\Downarrow \quad \wedge \quad \mathbb{E}[N]\Uparrow \implies \mathbb{E}[M]\Uparrow$$

then $M \lesssim N$.

We defer the proof to Section 6.

The context lemma says that we need only consider the behaviour of M and N with respect to evaluation contexts. Despite the presence of non-determinism, the language has a rich equational theory. A selection of laws is presented in Fig. 3. Throughout, we follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables.

4.4. Unique fixed-point induction

The context lemma allows a number of basic equivalences to be established. But the basic equivalences thus provable are typically insufficient to prove anything interesting about recursively defined entities. One can directly apply the context lemma, although reasoning this way is somewhat tedious. An alternative is to develop an operational counterpart to the denotational fixed-point induction rule, which characterises a recursive function (say) in terms of its "finite approximations". In [36], Moran and Sands show that call-by-need supports a fixed-point induction rule, based on a cost-sensitive preorder. This work could be adapted to the present setting by studying a convex-approximation relation.[4]

We will focus instead on a rather different proof rule for recursion based on unique fixed-points. 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 equation has a unique fixed point means that one can prove equivalence of two terms by showing that they both satisfy the recursion equation. The usual syntactic characterisation is that of *guarded recursion*: if recursive calls are syntactically "guarded" by an observable action then the fixed-point of the definition is unique.[5]

In a functional language the use of unique fixed-points is rather uncommon. The natural notion of "guard" is a constructor; however, many recursive definitions are "unguarded"—for example, the standard *filter* function, which selects all elements from a list which satisfy a predicate. Additionally, the presence of destructors make the usual notions of guardedness rather ineffective. Consider, for example, the equation

$$x \cong 1{:}(\mathsf{tail}\ (\mathsf{tail}\ x)).$$

Despite the fact that on the right-hand side, x is guarded by the cons-constructor, there are many deterministic solutions for this equation. For example, both $1{:}\bot$ and the infinite list of ones are solutions.

To recover a usable unique fixed-point theorem, we work with a finer, more intentional theory of equivalence. The motivation is that using a finer equivalence will give us fewer solutions to recursive equations, which in turn will give us a unique fixed-point proof rule.

The finer equivalence we use is called *cost equivalence* [46].

Definition 4 (*Cost Equivalence*). We say that M is *cost equivalent to* N, written $M \doteq N$, if for all $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,

$$\mathbb{C}[M] \Downarrow^n \iff \mathbb{C}[N] \Downarrow^n \quad \wedge \quad \mathbb{C}[M] \Uparrow \iff \mathbb{C}[N] \Uparrow .$$

Clearly $\doteq$ is contained in $\cong$. The point is that cost equivalence equations have unique solutions—provided that the recursion is guarded by at least one computation step. To make this precise, we need to introduce a very useful syntactic representation/

[4] The present refinement relation does not support a fixed-point induction principle since divergence is not a bottom element in this ordering.

[5] The technique of guarding recursion to force uniqueness of fixed-points may also be seen in metric semantics, see e.g., [5].

characterisation of a computation step, the *tick*:

$$^{\checkmark}M \stackrel{\text{def}}{=} \mathsf{let}\ \{\,\}\ \mathsf{in}\ M.$$

Clearly, $\checkmark$ adds one unit to the cost of evaluating M without otherwise changing its behaviour, so that $M \cong {}^{\checkmark}M$. Note that

$$M \Downarrow \iff {}^{\checkmark}M \Downarrow \qquad M \Downarrow^{n} \iff {}^{\checkmark}M \Downarrow^{n+1}.$$

We will write ${}^{k\checkmark}M$ to mean that M has been slowed down by k ticks. As an example of a cost equivalence involving the tick, we have the following:

$$(\lambda x.M)y \doteq {}^{2\checkmark}M[y/x]. \tag{β}$$

Now we are in a position to state the unique fixed-point induction principle, which is a variation on the proof rule of “improvement up to context” [47], and Lassen and Moran’s cost equivalence induction [29,34].

Theorem 2 (Unique fixed-point induction). *For any M, N, $\mathbb{C}$, and substitution σ, the following proof rule is sound*:

$$\frac{M \doteq {}^{\checkmark}\mathbb{C}[M\sigma] \quad N \doteq {}^{\checkmark}\mathbb{C}[N\sigma]}{M \doteq N}.$$

The proof is given in Section 6. It may seem at first sight that cost equivalence is too fine to be applicable. The reason that this appears not to be the case in practice is that firstly, proof techniques such as the context lemma extend in a very straightforward manner to cost equivalence, and secondly, for every equivalence in Fig. 3, there is a corresponding cost equivalence which can be obtained by adding an appropriate number of ticks (in appropriate places) to the left- and right-hand sides. We saw one example of this above (β). Rather than list all of these we take just a couple of examples:

$$\mathsf{case}\ c_j\,\vec{y}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to M_i\} \doteq {}^{2\checkmark}M_j[\vec{y}/\vec{x}_j] \quad (\textit{case-}\beta),$$

$$\mathbb{E}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \doteq \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{E}[N_i]\} \quad (\textit{case-}\mathbb{E}).$$

Collectively, We refer to these rules as the *tick algebra*. An example use of the tick algebra, in conjunction with unique fixed-point induction is the proof of the commutativity of merge, is given in the next section.

5. Correctness of the translation

The calculus of stream processors presented in Section 3 (and introduced in [9]) represents the designers’ intuition regarding the “essence of fudgets”. The purpose of this section is to demonstrate that the semantics we have given to this calculus in terms of the non-deterministic lazy language is consistent with the laws and reductions of the calculus: every congruence is an equivalence, and each reduction rule is a refinement. More precisely,

Theorem 3. *For any stream processors S and T,*

(i) $S \sim T \Longrightarrow [\![S]\!] \cong [\![T]\!]$,
(ii) $S \rightarrow T \Longrightarrow [\![S]\!] \lesssim [\![T]\!]$.

5.1. Properties of merge

In order to establish the theorem we begin by considering properties of the merge operation.

First a notational abbreviation: given a set of recursive definitions $\vec{f} = \vec{M}$ we will write a derivation of the form

$$\begin{aligned} \vec{f} \vdash M_1 &\doteq M_2 \\ &\vdots \\ &\doteq M_n \end{aligned}$$

to mean

$$\begin{aligned} \mathsf{let}\ \{\vec{f} = \vec{M}\}\ \mathsf{in}\ M_1 &\doteq \mathsf{let}\ \{\vec{f} = \vec{M}\}\ \mathsf{in}\ M_2 \\ &\vdots \\ &\doteq \mathsf{let}\ \{\vec{f} = \vec{M}\}\ \mathsf{in}\ M_n. \end{aligned}$$

This applies to $\equiv$, $\cong$, and $\lesssim$ also.

The two key properties of *merge* that we will establish are that it is commutative and associative.

Proposition 4.

$$merge\ xs\ ys \cong merge\ ys\ xs \quad (merge\text{-}comm),$$

$$merge\ xs\ (merge\ ys\ zs) \cong merge\ (merge\ xs\ ys)\ zs \quad (merge\text{-}assoc).$$

Proof. We prove only (*merge-comm*); (*merge-assoc*) is similar. We use unique fixed-point induction. Let $\mathbb{C}$ be the context:

$$^{5}\!\checkmark \begin{pmatrix} \mathsf{case}\ ys\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \rightarrow xs \\ z{:}ys \rightarrow z{:}[\cdot] \end{pmatrix} \oplus \begin{pmatrix} \mathsf{case}\ xs\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \rightarrow ys \\ z{:}xs \rightarrow z{:}[\cdot] \end{pmatrix}.$$

Then since

$$
\begin{aligned}
&\mathit{merge} \vdash \mathit{merge}\ \mathit{xs}\ \ \mathit{ys}\\
&\qquad \doteq ({}^{2\checkmark}\lambda \mathit{xs}, \mathit{ys}.\ldots \oplus \ldots)\mathit{xs}\ \mathit{ys} && (\mathit{value}\text{-}\beta)\\
&\qquad \doteq^{6\checkmark} \begin{pmatrix} \mathsf{case}\ \mathit{xs}\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to \mathit{ys} \\ \quad z\mathbf{:}\mathit{xs} \to z\mathbf{:}\mathit{merge}\ \mathit{xs}\ \ \mathit{ys} \end{pmatrix}\\
&\qquad\quad \oplus \begin{pmatrix} \mathsf{case}\ \mathit{ys}\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to \mathit{xs} \\ \quad z\mathbf{:}\mathit{ys} \to z\mathbf{:}\mathit{merge}\ \mathit{xs}\ \ \mathit{ys} \end{pmatrix} && (\beta) \times 2\\
&\qquad \doteq^{6\checkmark} \begin{pmatrix} \mathsf{case}\ \mathit{ys}\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to \mathit{xs} \\ \quad z\mathbf{:}\mathit{ys} \to z\mathbf{:}\mathit{merge}\ \mathit{xs}\ \ \mathit{ys} \end{pmatrix}\\
&\qquad\quad \oplus \begin{pmatrix} \mathsf{case}\ \mathit{xs}\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to \mathit{ys} \\ \quad z\mathbf{:}\mathit{xs} \to z\mathbf{:}\mathit{merge}\ \mathit{xs}\ \ \mathit{ys} \end{pmatrix} && (\oplus\text{-}\mathit{comm})\\
&\qquad \equiv {}^{\checkmark}\mathbb{C}[\mathit{merge}\ \mathit{xs}\ \ \mathit{ys}]
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{merge} \vdash \mathit{merge}\ \ \mathit{ys}\ \mathit{xs}\\
&\qquad \doteq ({}^{2\checkmark}\lambda \mathit{xs}, \mathit{ys}.\ldots \oplus \ldots)\mathit{xs}\mathit{ys} && (\mathit{value}\text{-}\beta)\\
&\qquad \doteq^{6\checkmark} \begin{pmatrix} \mathsf{case}\ \mathit{ys}\ \mathsf{of} \\ \qquad \mathsf{nil} \to \mathit{xs} \\ \quad z\mathbf{:}\mathit{zs} \to z\mathbf{:}\mathit{merge}\ \mathit{zs}\ \mathit{xs} \end{pmatrix}\\
&\qquad\quad \oplus \begin{pmatrix} \mathsf{case}\ \mathit{xs}\ \mathsf{of} \\ \qquad \mathsf{nil} \to \mathit{ys} \\ \quad z\mathbf{:}\mathit{zs} \to z\mathbf{:}\mathit{merge}\ \ \mathit{ys}\ \mathit{zs} \end{pmatrix} && (\beta) \times 2\\
&\qquad \equiv {}^{\checkmark}\mathbb{C}[\mathit{merge}\ \ \mathit{ys}\ \mathit{xs}], && (\text{rename}),
\end{aligned}
$$

the desired result follows by unique fixed-point induction and the soundness of $\doteq$. □

Lemma 5.

$$\mathit{merge}(x\mathbf{:}\mathit{xs})\mathit{ys} \lesssim x\mathbf{:}\mathit{merge}\ \mathit{xs}\ \ \mathit{ys} \quad (\mathit{merge}\text{-}\mathit{eval}).$$

Proof. By calculation:

$$
\begin{aligned}
&\mathit{merge} \vdash \mathit{merge}\ (x\mathbin{:}xs)ys \\
&\quad \cong (\lambda xs, ys, \ldots . \oplus \ldots)\ (x\mathbin{:}xs)ys && (\mathit{value}\text{-}\beta) \\
&\quad \cong \begin{pmatrix} \mathsf{case}\ x\mathbin{:}xs\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to ys \\ \quad z\mathbin{:}zs \to z\mathbin{:}\mathit{merge}\ zs\ \ ys \end{pmatrix} \\
&\qquad \oplus \begin{pmatrix} \mathsf{case}\ ys\ \mathsf{of} \\ \quad \mathsf{nil}\ \ \to xs \\ \quad z\mathbin{:}zs \to z\mathbin{:}\mathit{merge}\ xs\ \ zs \end{pmatrix} && (\beta) \times 2 \\
&\quad \lesssim \mathsf{case}\ x\mathbin{:}xs\ \mathsf{of} && (\oplus\text{-}\mathit{left}) \\
&\qquad \mathsf{nil}\ \ \to ys \\
&\qquad z\mathbin{:}zs \to z\mathbin{:}\mathit{merge}\ zs\ \ ys \\
&\quad \cong x\mathbin{:}\mathit{merge}\ xs\ \ ys && (\mathit{case}\text{-}\beta) \qquad \square
\end{aligned}
$$

5.2. The congruences

The commutativity and associativity of | follow in a straightforward way from the corresponding properties of *merge*. The associativity of $\lll$ simply follows from the associativity of function composition. In the sequel, we make the following overloaded use of the combinators of the stream processor calculus:

$$x\,!\,s \stackrel{\text{def}}{=} \lambda i.x\mathbin{:}(s\,i)$$

$$x\,?\,M \stackrel{\text{def}}{=} \lambda i.\mathsf{case}\ i\ \mathsf{of}\ \{\mathsf{nil} \to \mathsf{nil}\,|\,x\mathbin{:}xs \to M\,xs\}, \quad i,xs \notin \mathsf{FV}(M)$$

$$s \lessdot x \stackrel{\text{def}}{=} \lambda i.s\ (x\mathbin{:}i)$$

$$s \lll t \stackrel{\text{def}}{=} \lambda i.s\ (t\,i)$$

$$s\,|\,t \stackrel{\text{def}}{=} \lambda i.\mathit{merge}(s\,i)\ (t\,i)$$

$$\ell\,s \stackrel{\text{def}}{=} \lambda i.\mathsf{let}\ \{o = s(\mathit{merge}\ oi)\}\ \mathsf{in}\ o.$$

Lemma 6. *For any terms S and T and variables x and y,*

(i) $(S \lll T) \lessdot x \cong S \lll (T \lessdot x)$.
(ii) $S \lll (x\,!\,T) \cong (S \lessdot x) \lll T$.
(iii) $(x\,!\,S) \lll T \cong x\,!\,(S \lll T)$.
(iv) $(S\,|\,T) \lessdot x \cong (S \lessdot x)\,|\,(T \lessdot x)$.

$$
\begin{array}{ll}
(S \mid T) \lessdot x & \\
\equiv \lambda i.\mathsf{let}\ y = x{:}i & \text{(defn.)} \\
\qquad\quad w = \lambda i.\mathsf{let}\ z_1 = S\,i & \\
\qquad\qquad\qquad\qquad z_2 = T\,i & \\
\qquad\qquad\quad \mathsf{in}\ \mathit{merge}\ z_1\,z_2 & \\
\quad \mathsf{in}\ w\,y & \\
\cong \lambda i.\mathsf{let}\ y = x{:}i & (\mathit{value}\text{-}\beta), (\beta), \text{simplify} \\
\qquad\quad z_1 = S\,y & \\
\qquad\quad z_2 = T\,y & \\
\quad \mathsf{in}\ \mathit{merge}\ z_1\,z_2 & \\
\cong \lambda i.\mathsf{let}\ y_1 = x{:}i & (\mathit{value}\text{-}\mathit{copy}), \text{(rename)} \\
\qquad\quad z_1 = S\,y_1 & \\
\qquad\quad y_2 = x{:}i & \\
\qquad\quad z_2 = T\,y_2 & \\
\quad \mathsf{in}\ \mathit{merge}\ z_1\,z_2 & \\
\cong \lambda i.\mathsf{let}\ w_1 = \lambda i.\mathsf{let}\ y_1 = x{:}i & (\mathit{value}\text{-}\beta), (\beta), \text{simplify} \\
\qquad\qquad\qquad\quad \mathsf{in}\ S\,y_1 & \\
\qquad\quad z_1 = w_1\,i & \\
\qquad\quad w_2 = \lambda i.\mathsf{let}\ y_2 = x{:}i & \\
\qquad\qquad\qquad\quad \mathsf{in}\ S\,y_2 & \\
\qquad\quad z_2 = w_2\,i & \\
\quad \mathsf{in}\ \mathit{merge}\ z_1\,z_2 & \\
\equiv (S \lessdot x) \mid (T \lessdot x) & \text{(defn.)}
\end{array}
$$

Fig. 4. Proof of Lemma 6(iv).

$$
\begin{array}{ll}
(x\,?\,S) \lessdot y) & \\
\equiv \lambda i.\mathsf{let}\ w = y{:}i & \text{(defn.)} \\
\qquad\quad z = \lambda i.\mathsf{case}\ i\ \mathsf{of} & \\
\qquad\qquad\qquad \mathsf{nil} \quad \to \mathsf{nil} & \\
\qquad\qquad\qquad x{:}xs \to S\,xs & \\
\quad \mathsf{in}\ z\,w & \\
\cong \lambda i.\mathsf{let}\ w = y{:}i & (\mathit{value}\text{-}\beta), (\beta) \\
\quad \mathsf{in}\ \mathsf{case}\ w\ \mathsf{of} & \\
\qquad\quad \mathsf{nil} \quad \to \mathsf{nil} & \\
\qquad\quad x{:}xs \to S\,xs & \\
\equiv \lambda i.S[{}^{y}/_{x}]\,i & (\mathit{case}\text{-}\beta)
\end{array}
$$

Fig. 5. Proof of Lemma 6 (vi).

(v) $(x\,!\,S) \lessdot y \cong x\,!\,(S \lessdot y)$.
(vi) $(x\,?\,S) \lessdot y \cong \lambda i.S[y/x]i$.

Proof. The proof of (iv) is to be found in Fig. 4 and the proof of (vi) in Fig. 5. The others have similar proofs. □

5.3. The reduction rules

Throughout, we will use the syntactic equivalences for application and constructors mentioned in Section 3.2 without reference, but only where such usage does not obscure the calculation in question.

$$
\begin{array}{rll}
\ell\,(x\,!\,S) \equiv & \lambda i.\mathsf{let}\ t = \lambda i.\mathsf{let}\ z = S\,i\ \mathsf{in}\ x{:}z & \text{(defn.)}\\
& \quad y = \mathit{merge}\ o\,i & \\
& \quad o = t\,y & \\
& \mathsf{in}\ o & \\
\cong & \lambda i.\mathsf{let}\ o = x{:}z & (\mathit{value}\text{-}\beta), (\beta), \text{simplify}\\
& \quad y = \mathit{merge}\ o\,i & \\
& \quad z = S\,y & \\
& \mathsf{in}\ o & \\
\cong & \lambda i.\mathsf{let}\ o = x{:}z & (\mathit{value}\text{-}\beta)\\
& \quad y = \mathit{merge}\ o\,i & \\
& \quad z = S\,y & \\
& \mathsf{in}\ x{:}z & \\
\lesssim & \lambda i.\mathsf{let}\ y = x{:}w & (\mathit{merge}\text{-}\mathit{eval})\\
& \quad w = \mathit{merge}\ z\,i & \\
& \quad z = S\,y & \\
& \mathsf{in}\ x{:}z &
\end{array}
$$

Fig. 6. First half of proof of Lemma 7 (ii).

Lemma 7. *For any terms S and T and variables x,*

(i) $(x\,!\,S) \mid T \lesssim x\,!\,(S \mid T)$.
(ii) $\ell\,(x\,!\,S) \lesssim x\,!\,(\ell\,(S \lessdot x))$.
(iii) $(\ell\,S) \lessdot x \lesssim \ell(S \lessdot x)$.

Proof. We show (ii) only. The result follows since we have that (modulo renaming)

$$
\begin{array}{rll}
\ell\,(x\,!\,S) \lesssim & \lambda i.\mathsf{let}\ \ y = x{:}w & \text{(Fig. 6)}\\
& \qquad w = \mathit{merge}\ zi & \\
& \qquad z = Sy & \\
& \quad \mathsf{in}\ x{:}z & \\
\cong & x\,!\,(\ell\,(S \lessdot x)) & \text{(Fig. 7)} \qquad \square
\end{array}
$$

The proof of Theorem 3 now follows from the above lemmata by an easy induction on the structure of the stream processors. The only other properties needed in the proof are that (i) the image of the translation $[\![\cdot]\!]$ is always a lambda abstraction, and that (ii) the refinement and congruence relation are closed under substitution of values (in particular, lambda abstractions) for variables.

6. Proofs of main theorems

This section gives an outline of the technical development and proofs of the main results. Most proofs follow a direct style reasoning which is reminiscent of proofs about functional languages with effects by Mason and Talcott et al. [1,31,54]. In order to make this style of proof rigourous we generalise the abstract machine semantics so that it works on *configuration contexts*—configurations with holes. To ensure that transitions on configuration contexts are consistent with hole filling one must work with

$$
\begin{array}{rlr}
x\,!\,(\ell\ (S \prec x)) \equiv & \lambda i.\mathsf{let}\ t = \lambda i.\mathsf{let}\ y = x{:}i\ \mathsf{in}\ S\,y & \text{(defn.)}\\
& \quad w_1 = \lambda i.\mathsf{let}\ z = \mathit{merge}\ o\,i, o = t\,z\ \mathsf{in}\ o & \\
& \mathsf{in}\ \mathsf{let}\ w_2 = w_1\ i\ \mathsf{in}\ x{:}w_2 & \\
\cong & \lambda i.\mathsf{let}\ t = \lambda i.\mathsf{let}\ y = x{:}i\ \mathsf{in}\ S\,y & (\mathit{let\text{-}let})\\
& \quad w_1 = \lambda i.\mathsf{let}\ z = \mathit{merge}\ o\,i, o = t\,z\ \mathsf{in}\ o & \\
& \quad w_2 = w_1\,i & \\
& \mathsf{in}\ x{:}w_2 & \\
\cong & \lambda i.\mathsf{let}\ t = \lambda i.\mathsf{let}\ y = x{:}i\ \mathsf{in}\ S\,y & (\mathit{value}\text{-}\beta), (\beta)\\
& \quad w_2 = \mathsf{let}\ z = \mathit{merge}\ o\,i, o = t\,z\ \mathsf{in}\ o & \\
& \mathsf{in}\ x{:}w_2 & \\
\cong & \lambda i.\mathsf{let}\ t = \lambda i.\mathsf{let}\ y = x{:}i\ \mathsf{in}\ S\,y & (\mathit{let\text{-}let})\\
& \quad w_2 = o & \\
& \quad z = \mathit{merge}\ o\,i & \\
& \quad o = t\,z & \\
& \mathsf{in}\ x{:}w_2 & \\
\cong & \lambda i.\mathsf{let}\ z = \mathit{merge}\ o\,i & (\mathit{value}\text{-}\beta), (\beta), \text{simplify}\\
& \quad o = \mathsf{let}\ y = x{:}z\ \mathsf{in}\ S\,y & \\
& \mathsf{in}\ x{:}o & \\
\cong & \lambda i.\mathsf{let}\ y = x{:}w & (\mathit{let\text{-}let}), (\text{rename})\\
& \quad w = \mathit{merge}\ z\,i & \\
& \quad z = S\,y & \\
& \mathsf{in}\ x{:}z &
\end{array}
$$

Fig. 7. Second half of proof of Lemma 7 (ii).

a more general representation of contexts. One such approach is described in [54]. We use an alternative approach to generalising contexts which is due to Pitts [44].

6.1. Concerning divergence

The usual definition of an infinite reduction sequence is the following:

$$\langle \Gamma,\ M,\ S\rangle \rightarrow^{\omega} \overset{\text{def}}{=} \exists\, \{\Sigma_i\}_{i\in\mathbb{N}} \text{ such that}$$

$$\Sigma_0 \equiv \langle \Gamma,\ M,\ S\rangle \wedge \forall i \in \mathbb{N}.\Sigma_i \rightarrow \Sigma_{i+1}$$

To see that $\Uparrow$ is indeed equivalent to this definition, first note that the existence of such a set allows us to construct arbitrarily long reduction sequences easily. The other direction is a little more involved. Suppose that, for all n, $\langle \Gamma,\ M,\ S\rangle \rightarrow^{n}$, but that $\langle \Gamma,\ M,\ S\rangle \not\rightarrow^{\omega}$. The tree rooted at $\langle \Gamma,\ M,\ S\rangle$ has infinitely many nodes, but since $\langle \Gamma,\ M,\ S\rangle \not\rightarrow^{\omega}$ there is no infinite path through that tree. Since $\rightarrow$ is finitely branching and has no infinite path, the tree must have finitely many nodes, by König's Lemma. But the assumption that for all n, $\langle \Gamma,\ M,\ S\rangle \rightarrow^{n}$ implies that the tree has infinitely many nodes; a contradiction.

6.2. Substituting contexts

Following Pitts [44], we use second-order syntax to represent (and generalise) the traditional definition of contexts given in Section 4.2. We give a fuller description in [48]; other examples of their use are to be found in [28,34]. The idea is that instead of holes $[\cdot]$ we use *second-order variables*, ranged over by ξ, applied to some vector of variables. The syntax of generalised contexts is

$$\begin{aligned}\mathbb{C},\mathbb{D} ::= {} & \xi\cdot[\vec{x}]\\ & \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}x \mid c\vec{x}\\ & \mid \mathsf{let}\ \{\vec{x}=\vec{\mathbb{D}}\}\ \mathsf{in}\ \mathbb{C}\\ & \mid \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\vec{x}_i \to \mathbb{D}_i\}\\ & \mid \mathbb{C}\oplus\mathbb{D}.\end{aligned}$$

$\mathbb{V}$ and $\mathbb{W}$ will range over value contexts, $\mathbb{\Gamma}$ and $\mathbb{\Delta}$ over heap contexts, and $\mathbb{S}$ and $\mathbb{T}$ over stack contexts. Each "hole variable" ξ has a fixed arity, and ranges over meta-abstractions of the form $(\vec{x})M$ where the length of $\vec{x}$ is the arity of ξ. In the meta-abstraction $(\vec{x})M$, the variables $\vec{x}$ are bound in M. Hole-filling is now a general non-capturing substitution: $[(\vec{x})M/\xi]$. The effect of a substitution is as expected (remembering that the $\vec{x}$ are considered bound in $(\vec{x})M$). Coupled with the meta-abstraction is of course meta-application, written $\xi\cdot[\vec{x}]$. We restrict application of ξ to variables so that hole-filling cannot violate the restriction on syntax. In the definition of substitution we make the following identification:

$$(\vec{x})M\cdot[\vec{y}] \equiv M[\vec{y}/\vec{x}].$$

This definition of context generalises the usual definition since we can represent a traditional context $\mathbb{C}$ by $\mathbb{C}[\xi\cdot[\vec{x}]]$ where $\vec{x}$ is a vector of the capture-variables of $\mathbb{C}$; filling $\mathbb{C}$ with a term M is then represented by $(\mathbb{C}[\xi\cdot[\vec{x}]])[(\vec{x})M/\xi]$.

Example. The traditional context $\mathsf{let}\ x=[\cdot]\ \mathsf{in}\ \lambda y.[\cdot]$ can be represented by $\mathsf{let}\ x=\xi\cdot[(x,y)]\ \mathsf{in}\ \lambda y\cdot\xi\cdot[(x,y)]$. Filling the hole with the term xy is represented by

$$\begin{aligned}&(\mathsf{let}\ x=\xi\cdot[(x,y)]\ \mathsf{in}\ \lambda y.\xi\cdot[(x,y)])[(x,y)xy/\xi]\\ &\quad\equiv \mathsf{let}\ z=(x,y)\ xy\cdot[(z,y)]\ \mathsf{in}\ \lambda w.(x,y)\ xy\cdot[(x,w)]\\ &\quad\equiv \mathsf{let}\ z=zy\ \mathsf{in}\ \lambda w.xw\end{aligned}$$

which is α-equivalent to what we would have obtained by the usual hole-filling with capture. Note that the generalised representation permits contexts to be identified up to α-conversion.

Henceforth, we work only with generalised contexts. We will write $\mathbb{C}[(\vec{x})M]$ to mean $\mathbb{C}[(\vec{x})M/\xi]$ when $\mathbb{C}$ contains just a single hole variable ξ. We assume that the arities of hole variables are always respected.

We implicitly generalise our definitions of improvement to work with generalised contexts. This is not quite identical to the earlier definition since with generalised contexts, when placing a term in a hole we obtain a substitution instance of the term. This means in particular that improvement is now closed under substitution (variable-for-variable) by definition—a useful property. This difference is a relatively minor technicality which we will gloss over in this section.

6.3. Open uniform computation

The basis of our proofs will be to compute with configurations containing holes and free variables. Thanks to the capture-free representation of contexts, this means that normal reduction can be extended to contexts with ease. See [48] for a thorough treatment of generalised contexts and how they support generalisation of inductive definitions over terms.

Firstly, in order to fill the holes in a configuration we need to identify configurations up to renaming of the heap variables (recalling that update-markers on the stack are also binding occurrences of heap variables). In what follows, $\mathbb{\Sigma}$ will range over configuration contexts $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$.

We tacitly extend the operational semantics to open configurations with holes. Note that holes can only occur in the stack within the branches of case alternatives. In what follows θ will range over substitutions composed of variable-for-variable substitutions and substitutions of the form $[(\vec{x}_i)M_i/\xi_i]$. We will write $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \rightarrow^n \mathbb{\Sigma} \not\rightarrow$ to mean that configuration context $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$ reduces in n steps to configuration context $\mathbb{\Sigma}$, which does not reduce any further. $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \Uparrow$ has the obvious meaning.

We have the following key property.

Lemma 8 (Extension). *If* $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \rightarrow^k \langle \mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\rangle$ *then*

(i) *for all* $\mathbb{\Gamma}'$ *and* $\mathbb{S}'$ *such that* $\langle \mathbb{\Gamma}'\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\mathbb{S}'\rangle$ *is well-formed,* $\langle \mathbb{\Gamma}'\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\mathbb{S}'\rangle \rightarrow^k \langle \mathbb{\Gamma}'\mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\mathbb{S}'\rangle$, *and*
(ii) *for all* θ, $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle\theta \rightarrow^k \langle \mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\rangle\theta$.

Proof. (i) follows by inspection of possible open reductions over configuration contexts.

(ii) amounts to the standard substitution lemma; see [48] for a general argument. □

The following *open uniform computation* property is central. It allows us to evaluate open configuration contexts until either the computation is finished, or we find ourselves in an "interesting" case.

Lemma 9 (Open uniform computation). *If well-formed and well-typed configuration context* $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \xrightarrow{n} \mathbb{\Sigma} \not\rightarrow$ *then* $\mathbb{\Sigma}$ *has one of the following forms*:

(i) $\langle \mathbb{\Delta},\ \mathbb{V},\ \varepsilon\rangle$;
(ii) $\langle \mathbb{\Delta},\ \xi_i \cdot [\vec{y}],\ \mathbb{T}\rangle$, *for some hole* ξ_i, *or*
(iii) $(\mathbb{\Delta},\ x,\ \mathbb{T})$, $x \in \mathsf{FV}(\mathbb{\Gamma}, \mathbb{C}, \mathbb{S})$.

Proof. Assume $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \rightarrow^n \mathbb{\Sigma} \not\rightarrow$. We consider the reduction of $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$ and proceed by induction on n with cases on the structure of $\mathbb{C}$. We show four illustrative cases only. The others are similar.

$\mathbb{C} \equiv \xi_i \cdot [\vec{y}]$. This is a type (ii) context, so we are done.

$\mathbb{C} \equiv x$. There are three possibilities here: either x is bound by $\mathbb{\Gamma}$, by $\mathbb{S}$, or neither. In the first case, $\mathbb{\Gamma} \equiv \mathbb{\Delta}\{x = \mathbb{D}\}$. By (*Lookup*), $\langle \mathbb{\Delta}\{x = \mathbb{D}\},\ x,\ \mathbb{S}\rangle$ reduces to $\langle \mathbb{\Delta},\ \mathbb{D},\ \#x : \mathbb{S}\rangle$. By the inductive hypothesis, we know that $\langle \mathbb{\Delta},\ \mathbb{D},\ \#x : \mathbb{S}\rangle$ reduces to a configuration context of type (i), (ii), or (iii), and therefore $\langle \mathbb{\Delta}\{x = \mathbb{D}\},\ x,\ \mathbb{S}\rangle$ does also, as required. The second case cannot occur since it would imply, by (*Black Hole*), that $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \Uparrow$ exclusively (i.e., without any other behaviours), contradicting the initial assumption. In the last case, $\langle \mathbb{\Gamma},\ x,\ \mathbb{S}\rangle$ is a type (iii) context, and we are done.

$\mathbb{C} \equiv \mathbb{V}$. There are four sub-cases, depending upon the structure of $\mathbb{S}$; we consider only the case when $\mathbb{S} \equiv x : \mathbb{T}$. Since $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$ is well-typed, $\mathbb{V} \equiv \lambda y.\mathbb{D}$, and by (*Subst*), $\langle \mathbb{\Gamma},\ \lambda y.\mathbb{D},\ x : \mathbb{T}\rangle$ reduces to $\langle \mathbb{\Gamma},\ \mathbb{D}[x/y],\ \mathbb{T}\rangle$. The inductive hypothesis applies, and the result follows as above.

$\mathbb{C} \equiv \mathbb{D} \oplus \mathbb{E}$. There are two possibilities here: either $\langle \mathbb{\Gamma},\ \mathbb{D} \oplus \mathbb{E},\ \mathbb{S}\rangle \to \langle \mathbb{\Gamma},\ \mathbb{D},\ \mathbb{S}\rangle$ or $\langle \mathbb{\Gamma},\ \mathbb{D} \oplus \mathbb{E},\ \mathbb{S}\rangle \to \langle \mathbb{\Gamma},\ \mathbb{E},\ \mathbb{S}\rangle$. In either case, the inductive hypothesis applies, and the desired result follows. □

6.4. Translation

The relationship between terms and configurations is characterised by a translation function from configurations to terms, defined inductively on the stack:

$$\begin{aligned}
\mathsf{trans}\langle \emptyset,\ M,\ \varepsilon\rangle &= M,\\
\mathsf{trans}\langle \{\vec{x} = \vec{M}\},\ N,\ \varepsilon\rangle &= \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\\
\mathsf{trans}\langle \Gamma,\ M,\ x : S\rangle &= \mathsf{trans}\langle \Gamma,\ M x,\ S\rangle,\\
\mathsf{trans}\langle \Gamma,\ M,\ \#x : S\rangle &= \mathsf{trans}\langle \Gamma\{x = M\},\ x,\ S\rangle,\\
\mathsf{trans}\langle \Gamma,\ M,\ alts : S\rangle &= \mathsf{trans}\langle \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ alts,\ S\rangle.
\end{aligned}$$

We can extend the definition of trans to cover open configurations and configuration contexts. The following lemma clarifies the relationship.

Lemma 10 (Translation). *For all $\mathbb{D}$, $\mathbb{\Gamma}$, $\mathbb{C}$, $\mathbb{S}$ such that $\mathbb{D} = \mathsf{trans}\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$, there exists $k \geqslant 0$ such that $\langle \emptyset,\ \mathbb{D},\ \varepsilon\rangle \to^k \langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$.*

Proof. Simple induction on $\mathbb{S}$. □

6.5. Proof. the context lemma

The proof of the context lemma relies upon three lemmas, the first of which is the simplest.

Lemma 11. *$M \lesssim N$ if and only if for all $\mathbb{\Sigma}$,*

$$\mathbb{\Sigma}[(\vec{x})N] \Downarrow\ \Longrightarrow\ \mathbb{\Sigma}[(\vec{x})M] \Downarrow$$

and

$$\Sigma[(\vec{x})N]\Uparrow \implies \Sigma[(\vec{x})M]\Uparrow .$$

Proof (*Sketch*). ($\Leftarrow$). Trivial; let $\Sigma = \langle \emptyset,\ \mathbb{C},\ \varepsilon\rangle$.

($\Rightarrow$). We show only the divergence half; the convergence follows similarly. Assume that the divergence half of $M \lesssim N$, i.e., that for all $\mathbb{D}$,

$$\langle \emptyset,\ \mathbb{D}[(\vec{x})N],\ \varepsilon\rangle \Uparrow \implies \langle \emptyset,\ \mathbb{D}[(\vec{x})M],\ \varepsilon\rangle \Uparrow$$

and suppose $\Sigma[(\vec{x})N]\Uparrow$, i.e.

$$\forall n.\langle \mathbb{\Gamma}[(\vec{x})N],\ \mathbb{C}[(\vec{x})N],\ \mathbb{S}[(\vec{x})N]\rangle \rightarrow^{n} . \tag{6.1}$$

Let $\mathbb{D} \equiv \mathsf{trans}(\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S})$. By translation, there exists $k \geqslant 0$ such that $\langle \emptyset,\ \mathbb{D},\ \varepsilon\rangle \rightarrow^{k} \langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$. Therefore, by (6.1), we have $\forall n.\langle \emptyset,\ \mathbb{D}[(\vec{x})N],\ \varepsilon\rangle \rightarrow^{n+k}$. By assumption, we have that $\forall n.\langle \emptyset,\ \mathbb{D}[(\vec{x})M],\ \varepsilon\rangle \rightarrow^{n+k}$, and the result follows by translation. □

Lemma 12. *If for all Γ, and S,*

$$\langle \Gamma,\ (\vec{x})N\cdot[\vec{y}],\ S\rangle \Downarrow \implies \langle \Gamma, (\vec{x})M\cdot[\vec{y}], S\rangle \Downarrow$$

then for all Σ,

$$\Sigma[(\vec{x})N]\Downarrow \implies \Sigma[(\vec{x})M]\Downarrow,$$

where $\vec{x} \supseteq \mathsf{FV}(M,N)$.

Proof. Assume the premise and suppose $\Sigma[(\vec{x})N]\Downarrow^{n}$. We proceed via induction on n. By open uniform computation, Σ reduces in $k \geqslant 0$ steps to at least one[6] of

(1) $\langle \mathbb{\Delta},\ \mathbb{V},\ \varepsilon\rangle$, (2) $\langle \mathbb{\Delta},\ \xi\cdot[\vec{y}],\ \mathbb{S}\rangle$.

(A type (iii) context is not possible since Σ is closed.) In case (1), we are done. In case (2), we have

$$\langle \mathbb{\Delta}[(\vec{x})N],\ N[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\rangle \Downarrow^{n-k} \tag{6.2}$$

and

$$\Sigma[(\vec{x})M] \rightarrow^{k} \langle \mathbb{\Delta}[(\vec{x})M],\ M[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})M]\rangle. \tag{6.3}$$

[6] There are many possibilities here. There can be many different instances of cases (1) and (2) to which Σ reduces, and indeed it may also diverge. However, since $\Sigma[(\vec{x})M]\Downarrow$ we *know* that there must be an instance of at least one of cases (1) and (2). The possibility of divergence is irrelevant here.

By open uniform computation, $\langle \mathbb{\Delta}, M[\vec{y}/\vec{x}], \mathbb{S}\rangle$ reduces in $k' \geqslant 0$ steps to at least one of:

$$(2.1) \quad \langle \mathbb{\Delta}', \mathbb{W}, \varepsilon\rangle, \quad (2.2)\ \langle \mathbb{\Delta}', \xi\cdot[\vec{z}], \mathbb{T}\rangle.$$

(Again, a type (iii) context cannot arise because $(\mathbb{\Delta}, M[\vec{y}/\vec{x}], \mathbb{S})$ is closed, and the possibility of divergence is irrelevant.) In case (2.1), we have that

$$\langle \mathbb{\Delta}[(\vec{x})M], M[\vec{y}/\vec{x}], \mathbb{S}[(\vec{x})M]\rangle \to^{k'} \langle \mathbb{\Delta}'[(\vec{x})M], \mathbb{W}[(\vec{x})M], \varepsilon\rangle, \tag{6.4}$$

so

$$\begin{aligned} &\langle \mathbb{\Delta}'[(\vec{x})M], \mathbb{W}[(\vec{x})M], \varepsilon\rangle \Downarrow \\ &\quad\Longrightarrow \langle \mathbb{\Delta}[(\vec{x})M], M[\vec{y}/\vec{x}], \mathbb{S}[(\vec{x})M]\rangle \Downarrow && (6.4)\\ &\quad\Longrightarrow \mathbb{\Sigma}[(\vec{x})M] \Downarrow && (6.3)\end{aligned}$$

as required. In case (2.2), we know that $k' > 0$, since $M[\vec{y}/\vec{x}] \not\equiv \xi\cdot[\vec{z}]$. We have

$$\langle \mathbb{\Delta}[(\vec{x})N], N[\vec{y}/\vec{x}], \mathbb{S}[(\vec{x})N]\rangle \to^{k'} \langle \mathbb{\Delta}'[(\vec{x})N], N[\vec{z}/\vec{x}], \mathbb{T}[(\vec{x})N]\rangle \tag{6.5}$$

and

$$\langle \mathbb{\Delta}[(\vec{x})M], M[\vec{y}/\vec{x}], \mathbb{S}[(\vec{x})M]\rangle \to^{k'} \langle \mathbb{\Delta}'[(\vec{x})M], M[\vec{z}/\vec{x}], \mathbb{T}[(\vec{x})M]\rangle. \tag{6.6}$$

Therefore,

$$\begin{aligned} &\langle \mathbb{\Delta}[(\vec{x})N], N[\vec{z}/\vec{x}], \mathbb{T}[(\vec{x})N]\rangle \Downarrow^{n-k-k'} && ((6.5), (6.2))\\ &\quad\Longrightarrow \langle \mathbb{\Delta}'[(\vec{x})M], N[\vec{z}/\vec{x}], \mathbb{T}[(\vec{x})M]\rangle \Downarrow && (\text{I.H.})\\ &\quad\Longrightarrow \langle \mathbb{\Delta}'[(\vec{x})M], M[\vec{z}/\vec{x}], \mathbb{T}[(\vec{x})M]\rangle \Downarrow && (\text{ass.})\\ &\quad\Longrightarrow \langle \mathbb{\Delta}[(\vec{x})M], M[\vec{y}/\vec{x}], \mathbb{S}[(\vec{x})M]\rangle \Downarrow && (6.6)\\ &\quad\Longrightarrow \mathbb{\Sigma}[(\vec{x})M] && (6.3)\end{aligned}$$

as required. □

Lemma 13. *If for all Γ and S,*

$$\langle \Gamma, (\vec{x})N\cdot[\vec{y}], S\rangle \Uparrow \implies \langle \Gamma, (\vec{x})M\cdot[\vec{y}], S\rangle \Uparrow$$

then for all $\mathbb{\Sigma}$,

$$\mathbb{\Sigma}[(\vec{x})N] \Uparrow \implies \mathbb{\Sigma}[(\vec{x})M] \Uparrow,$$

where $\vec{x} \supseteq \mathsf{FV}(M, N)$.

Proof. Assume the premise. We show that

$$\forall k, \mathbb{\Gamma}, \mathbb{C}, \mathbb{S}. \langle \mathbb{\Gamma}[(\vec{x})N],\ \mathbb{C}[(\vec{x})N],\ \mathbb{S}[(\vec{x})N] \rangle \Uparrow$$
$$\Longrightarrow \langle \Gamma[(\vec{x})M],\ \mathbb{C}[(\vec{x})M],\ \mathbb{S}[(\vec{x})M] \rangle \to^{k}.$$

by complete induction on k. The base case is immediate, since $\Sigma[(\vec{x})M] \to^0$ holds for any Σ and M.

Let $\Sigma \equiv \langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \rangle$ and suppose $\Sigma[(\vec{x})N] \Uparrow$. Clearly, at least one of the following must be the case:

(1) $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \rangle \Uparrow$,
(2) $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \rangle \to^n \Sigma' \not\to$ and $\Sigma'[(\vec{x})N] \Uparrow$.

(There are of course other possibilities since $\langle \mathbb{\Gamma}[(\vec{x})N],\ \mathbb{C}[(\vec{x})N],\ \mathbb{S}[(\vec{x})N] \rangle$ may converge, but these are not relevant here.) In case (1),

$$\langle \mathbb{\Gamma}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M],\ \mathbb{S}[(\vec{x})M] \rangle \Uparrow,$$

which implies $\langle \mathbb{\Gamma}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M],\ \mathbb{S}[(\vec{x})M] \rangle \to^k$, and we are done.

For case (2), by open uniform computation, $\Sigma' \equiv \langle \mathbb{\Delta},\ \xi \cdot [\vec{y}],\ \mathbb{T} \rangle$ (since a type (i) context would contradict $\Sigma'[(\vec{x})N] \Uparrow$, and a type (iii) context cannot occur since $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \rangle$ is closed). Therefore,

$$\langle \mathbb{\Gamma}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M],\ \mathbb{S}[(\vec{x})M] \rangle \to^n \langle \mathbb{\Delta}[(\vec{x})M],\ M[\vec{y}/\vec{x}],\ \mathbb{T}[(\vec{x})M] \rangle. \tag{6.7}$$

By the assumption, $\langle \mathbb{\Delta}[(\vec{x})N],\ N[\vec{y}/\vec{x}], \mathbb{T}[(\vec{x})N] \rangle \Uparrow$ implies that

$$\langle \mathbb{\Delta}[(\vec{x})N],\ M[\vec{y}/\vec{x}],\ \mathbb{T}[(\vec{x})N] \rangle \Uparrow \tag{6.8}$$

Consider $\langle \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T} \rangle$. By (6.8), at least one of the following must be the case:

(2.1) $\langle \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T} \rangle \Uparrow$, or
(2.2) $\langle \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T} \rangle \to^{n'} \Sigma''$, and $\Sigma''[(\vec{x})N] \Uparrow$.

(Again, the other possibilities are irrelevant here.) In case (2.1),

$$\langle \mathbb{\Delta}[(\vec{x})M],\ M[\vec{y}/\vec{x}],\ \mathbb{T}[(\vec{x})M] \rangle \Uparrow,$$

which implies $\langle \mathbb{\Delta}[(\vec{x})M],\ M[\vec{y}/\vec{x}],\ \mathbb{T}[(\vec{x})M] \rangle \to^k$, and therefore, by (6.7), we are done.

For case (2.2), by open uniform computation, $\Sigma'' \equiv \langle \mathbb{\Delta}',\ \xi \cdot [\vec{z}],\ \mathbb{T}' \rangle$ (for all similar reasons to the above). Therefore,

$$\langle \mathbb{\Delta}[(\vec{x})M],\ M[\vec{y}/\vec{x}],\ \mathbb{T}[(\vec{x})M] \rangle \to^{n'} \langle \mathbb{\Delta}'[(\vec{x})M],\ M[\vec{z}/\vec{x}],\ \mathbb{T}'[(\vec{x})M] \rangle. \tag{6.9}$$

where $n' > 0$, since $M[\vec{y}/\vec{x}] \not\equiv \xi \cdot [\vec{z}]$. So, recalling that $\Sigma'' \equiv \langle \mathbb{\Delta}',\ \xi \cdot [\vec{z}],\ \mathbb{T}' \rangle$, $\Sigma' \equiv \langle \mathbb{\Delta},\ M[\vec{y}/\vec{x}],\ \mathbb{T} \rangle$, and $\Sigma \equiv \langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \rangle$,

$$\Sigma''[(\vec{x})N] \Uparrow$$
$$\Longrightarrow \forall k' < k. \Sigma''[(\vec{x})M] \to^{k'} \tag{I.H.}$$

$$\Longrightarrow \forall k' < k.\Sigma'[(\vec{x})M] \to^{k'+n'} \qquad (6.9)$$

$$\Longrightarrow \forall k' < k.\Sigma[(\vec{x})M] \to^{k'+n'+n} \qquad (6.7)$$

$$\Longrightarrow \Sigma[(\vec{x})M] \to^{k} \qquad n' > 0$$

as required. □

The generalised statement of the context lemma is:
For all terms M and N, if

$$\forall \Gamma, S, \sigma, n.\langle \Gamma,\ N\sigma,\ S\rangle \Downarrow \implies \langle \Gamma,\ M\sigma,\ S\rangle \Downarrow$$

and

$$\forall \Gamma, S, \sigma.\langle \Gamma,\ N\sigma,\ S\rangle \Uparrow \implies \langle \Gamma,\ M\sigma,\ S\rangle \Uparrow$$

then $M \lesssim N$.

This follows from Lemmas 11–13, and the fact that $M\sigma \equiv (\vec{x})M \cdot [\vec{y}]$ for $\sigma = [\vec{y}/\vec{x}]$.

There is a cost-sensitive analogue to the context lemma; the divergence half is as above, and the convergence half is similar to Lemma A.5 in [35].

6.6. Validating the equational theory

We present a proof of the validity of the cost equivalence variant of (*value-β*) and present and prove a lemma for establishing the correctness of the cost equivalence forms of (*case-*$\mathbb{E}$) and (*let-*$\mathbb{E}$). The corresponding results for observational equivalence follow by the soundness of $\doteq$ and the fact that ${}^{\checkmark}M \cong M$. The proof of the correspondence between evaluation contexts and configuration contexts of the form $\langle \Gamma,\ [\cdot],\ S\rangle$ is as for [35]. The proofs of the more complex laws (such as a cost equivalence version of (*value-copy*)) have a structure similar to that for (*value-β*), except they require more extensive use of open uniform computation.

6.6.1. Proof (value-β)

We show the cost equivalence variant of (*value-β*):

$$\mathsf{let}\ \{x = V,\ \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \doteq \mathsf{let}\ \{x = V,\ \vec{y} = \vec{\mathbb{D}}[{}^{2\checkmark}V]\}\ \mathsf{in}\ \mathbb{C}[{}^{2\checkmark}V].$$

Let $W \equiv {}^{2\checkmark}V$ throughout. By the cost-sensitive analogue to the context lemma, it suffices to show

$$\forall \mathbb{\Gamma}, \mathbb{S}.\langle \mathbb{\Gamma}[x]\{x = V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \Downarrow^{n}$$
$$\iff \langle \mathbb{\Gamma}[W]\{x = V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle \Downarrow^{\leqslant n} \qquad (6.10)$$

and

$$\forall \mathbb{\Gamma}, \mathbb{S}.\langle \mathbb{\Gamma}[W]\{x = V\}, \mathbb{C}[W], \mathbb{S}[W]\rangle \Uparrow \iff \langle \mathbb{\Gamma}[x]\{x = V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \Uparrow, \qquad (6.11)$$

where $x \notin \mathrm{dom}(\mathbb{\Gamma}, \mathbb{S})$, and the only hole is $[\cdot]$, a non-capturing hole.

We show the forward direction of (6.10) first; the reverse direction is similar. Suppose $\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \Downarrow^n$. We proceed by induction on n. By open uniform computation, $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle$ reduces in $k \geqslant 0$ steps to at least one of

$$(1)\ \ \langle \mathbb{\Delta},\ \mathbb{V},\ \varepsilon\rangle, \quad (2)\ \langle \mathbb{\Delta},\ [\cdot],\ \mathbb{T}\rangle, \quad (3)\ \langle \mathbb{\Delta},\ x,\ \mathbb{T}\rangle.$$

In case (1), we are done. In case (2), by extension, (*Lookup*) and (*Update*), we have

$$\begin{aligned}&\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle\\ &\quad \rightarrow^k \langle \mathbb{\Delta}[x]\{x=V\},\ x,\ \mathbb{T}[x]\rangle\\ &\quad \rightarrow^2 \langle \mathbb{\Delta}[x]\{x=V\},\ V,\ \mathbb{T}[x]\rangle,\end{aligned}$$

and by extension and the definition of W,

$$\begin{aligned}&\langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle\\ &\quad \rightarrow^k \langle \mathbb{\Delta}[W]\{x=V\},\ W,\ \mathbb{T}[W]\rangle\\ &\quad \rightarrow^2 \langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle.\end{aligned}$$

Since $(\mathbb{\Delta}[x]\{x=V\},\ V,\ \mathbb{T}[x]\rangle \Downarrow^{n-(k+2)}$, by the inductive hypothesis we have $\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle \Downarrow^{n-(k+2)}$, and the result follows.

In case (3), we have $(\mathbb{\Delta}[x]\{x=V\},\ V,\ \mathbb{T}[x]\rangle \Downarrow^{n-k-2}$, as above. Furthermore, by extension, (*Lookup*) and (*Update*), we have

$$\begin{aligned}&\langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle\\ &\quad \rightarrow^k \langle \mathbb{\Delta}[W]\{x=V\},\ x,\ \mathbb{T}[W]\rangle\\ &\quad \rightarrow^2 \langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle.\end{aligned}$$

From the inductive hypothesis, we have $\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle \Downarrow^{\leqslant n-k-2}$, and the result follows.

We show the forward direction of (6.11) only; the reverse direction is similar. This amounts to proving

$$\begin{aligned}&\forall k,\ \mathbb{\Gamma},\ \mathbb{C}, \mathbb{S}. \langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle \Uparrow\\ &\quad \Longrightarrow\ \langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \rightarrow^k .\end{aligned}$$

We proceed via complete induction on k. The base case is immediate. Suppose $\langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle \Uparrow$. At least one of the following must be the case:

(1) $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S})\Uparrow$, or
(2) $\langle \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\rangle \rightarrow^n \langle \mathbb{\Delta},\ \mathbb{D},\ \mathbb{T},\rangle$, and $\langle \mathbb{\Delta}[W]\{x=V\},\ \mathbb{D}[W],\ \mathbb{T}[W]\rangle \Uparrow$.

In case (1), the result follows immediately. In case (2), by open uniform computation, $\langle \mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\rangle$ can take on the following forms:

$$(2.1)\ \langle \mathbb{\Delta},\ [\cdot],\ \mathbb{T}\rangle, \quad (2.2)\ \langle \mathbb{\Delta},\ x,\ \mathbb{T}\rangle.$$

In case (2.1), by extension and the definition of W, we have

$$\begin{aligned}&\langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle\\ &\quad \to^{n}\langle \mathbb{\Delta}[W]\{x=V\},\ W,\ \mathbb{T}[W]\rangle\\ &\quad \to^{2}\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle\end{aligned}$$

and

$$\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{n}\langle \mathbb{\Delta}[x]\{x=V\},\ x,\ \mathbb{T}[x]\rangle. \tag{6.12}$$

Then the result follows, since

$$\begin{aligned}&\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle \Uparrow\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Delta}[x]\{x=V\},\ V,\ \mathbb{T}[x]\rangle \to^{k'} && \text{(I.H.)}\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Delta}[x]\{x=V\},\ x,\ \mathbb{T}[x]\rangle \to^{k'+2} && (\bigstar)\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{k'+2+n}\\ &\quad \Longrightarrow \langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{k}, && (6.12)\end{aligned}$$

where ($\bigstar$) follows by (*Update*) and (*Lookup*).

In case (2.2), by extension, (*Update*), and (*Lookup*), we have

$$\begin{aligned}&\langle \mathbb{\Gamma}[W]\{x=V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\rangle\\ &\quad \to^{n}\langle \mathbb{\Delta}[W]\{x=V\},\ x,\ \mathbb{T}[W]\rangle\\ &\quad \to^{2}\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle\end{aligned}$$

and

$$\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{n}\langle \mathbb{\Delta}[x]\{x=V\},\ x,\ \mathbb{T}[x]\rangle. \tag{6.13}$$

Then the result follows, since

$$\begin{aligned}&\langle \mathbb{\Delta}[W]\{x=V\},\ V,\ \mathbb{T}[W]\rangle \Uparrow\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Delta}[x]\{x=V\},\ V,\ \mathbb{T}[x]\rangle \to^{k'} && \text{(I.H.)}\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Delta}[x]\{x=V\},\ x,\ \mathbb{T}[x]\rangle \to^{k'+2} && (\bigstar)\\ &\quad \Longrightarrow \forall k'<k.\langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{k'+2+n}\\ &\quad \Longrightarrow \langle \mathbb{\Gamma}[x]\{x=V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\rangle \to^{k}, && (6.13)\end{aligned}$$

where ($\bigstar$) follows by (*Update*) and (*Lookup*).

6.6.2. *A lemma for* (*case*-$\mathbb{E}$) *and* (*let*-$\mathbb{E}$)

The following lemma can be used to validate (*case*-$\mathbb{E}$) and (*let*-$\mathbb{E}$). $\mathsf{CV}(\mathbb{E})$ denotes the *capture variables* of $\mathbb{E}$.

Lemma 14. *For all* $\mathbb{E}$, *there exist* Γ, S, *such that* $\mathrm{dom}(\Gamma, S) \subseteq \mathsf{CV}(\mathbb{E})$ *and*

$$\forall \Delta, T. \langle \Delta,\ \mathbb{E},\ T\rangle \to^* \langle \Delta\Gamma,\ [\cdot],\ ST\rangle.$$

Proof. By virtue of the correspondence between evaluation contexts and configurations, there exist Γ and S such that $\mathsf{trans}\langle \Gamma,\ [\cdot],\ S\rangle \equiv \mathbb{E}$, so by translation $\langle \emptyset,\ \mathbb{E},\ \varepsilon\rangle \to^* \langle \Gamma,\ [\cdot],\ S\rangle$, and thus by extension, provided $\mathrm{dom}(\Gamma, S) \subseteq \mathsf{CV}(\mathbb{E})$, $\langle \Delta,\ \mathbb{E},\ T\rangle \to^* \langle \Delta\Gamma,\ [\cdot],\ ST\rangle$. □

6.7. *Proof. Unique fixed-point induction*

We will use the following lemma to prove unique fixed-point induction.

Lemma 15. *For all* k, *if* $M \doteq N$ *then for all* Σ,

$$\Sigma[(\vec{x})M] \to^k \iff \Sigma[(\vec{x})N] \to^k .$$

Proof. Assume $M \doteq N$, i.e., that for all Σ, n,

$$\Sigma[(\vec{x})M] \Downarrow^n \iff \Sigma[(\vec{x})N] \Downarrow^n \tag{6.14}$$

and

$$\forall k. \Sigma[(\vec{x})M] \to^k \iff \forall k. \Sigma[(\vec{x})N] \to^k . \tag{6.15}$$

Suppose $\Sigma[(\vec{x})M] \to^k$ for some k. There are two (by no means mutually exclusive) possibilities:

(i) The partial sequence is a subsequence of a convergent sequence. Then (6.14) yields the desired result.
(ii) The partial sequence is a subsequence of a divergent sequence. Then (6.15) yields the desired result. □

Recall the statement of unique fixed-point induction:
For any M *and* N, *the following proof rule is sound*:

$$\frac{M \doteq {}^{\checkmark}\mathbb{C}[M] \quad N \doteq {}^{\checkmark}\mathbb{C}[N]}{M \doteq N}.$$

We generalise $\mathbb{C}[M]$ to $\mathbb{C}[(\vec{x})M]$. Assume the premises hold, i.e., that for all Σ,

$$\Sigma[(\vec{x})M] \Downarrow^n \iff \Sigma[{}^{\checkmark}\mathbb{C}[(\vec{x})M]] \Downarrow^n, \tag{6.16}$$

$$\Sigma[(\vec{x})N] \Downarrow^n \iff \Sigma[{}^{\checkmark}\mathbb{C}[(\vec{x})N]] \Downarrow^n, \tag{6.17}$$

$$\Sigma[{}^{\checkmark}\mathbb{C}[(\vec{x})M]]\Uparrow \iff \Sigma[(\vec{x})M]\Uparrow, \tag{6.18}$$

$$\Sigma[{}^{\checkmark}\mathbb{C}[(\vec{x})N]]\Uparrow \iff \Sigma[(\vec{x})N]\Uparrow . \tag{6.19}$$

It suffices to show, under the assumption of the premises, that

$$\forall\Sigma.\Sigma[(\vec{x})M]\Downarrow^{n} \iff \Sigma[(\vec{x})N]\Downarrow^{n} \tag{6.20}$$

and

$$\forall k,\Sigma.\Sigma[(\vec{x})M]\Uparrow \iff \Sigma[(\vec{x})N]\rightarrow^{k} \tag{6.21}$$

(where the Σ are such that $\Sigma[(\vec{x})M]$ and $\Sigma[(\vec{x})N]$ are closed).

We first show the forward direction of (6.20). Suppose $\Sigma[(\vec{x})M]\Downarrow^{n}$. We proceed by induction on n. By open uniform computation, Σ reduces in $k\geqslant 0$ to one of

(1) $\langle\mathbb{\Delta},\ \mathbb{V},\ \varepsilon\rangle$, (2) $\langle\mathbb{\Delta},\ \xi\cdot[\vec{y}],\ \mathbb{T}\rangle$.

In case (1), we are done. In case (2), first note that, letting $\sigma=[\vec{y}/\vec{x}]$, $\mathbb{C}[(\vec{x})M]\sigma\equiv\mathbb{C}\sigma[(\vec{x})M]$ since $\vec{x}\supseteq\mathsf{FV}(M)$, and similarly for N. Then we have that

$$\Sigma[(\vec{x})N]\rightarrow^{k}\langle\mathbb{\Delta}[(\vec{x})N],\ N\sigma,\ \mathbb{T}[(\vec{x})N]\rangle \tag{6.22}$$

and

$$\begin{aligned}
&\langle\mathbb{\Delta}[(\vec{x})M],\ M\sigma,\ \mathbb{T}[(\vec{x})M]\Downarrow^{n-k}\\
&\iff\langle\mathbb{\Delta}[(\vec{x})M],\ {}^{\checkmark}\mathbb{C}[(\vec{x})M]\sigma,\ \mathbb{T}[(\vec{x})M]\rangle\Downarrow^{n-k} && (6.16)\\
&\iff\langle\mathbb{\Delta}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M]\sigma,\ \mathbb{T}[(\vec{x})M]\rangle\Downarrow^{n-(k+1)} && (\checkmark)\\
&\implies\langle\mathbb{\Delta}[(\vec{x})N],\ \mathbb{C}\sigma[(\vec{x})N],\ \mathbb{T}[(\vec{x})N]\rangle\Downarrow^{n-(k+1)} && (\text{I.H.})\\
&\iff\langle\mathbb{\Delta}[(\vec{x})N],\ {}^{\checkmark}\mathbb{C}[(\vec{x})N]\sigma,\ \mathbb{T}[(\vec{x})N]\rangle\Downarrow^{n-k} && (\checkmark)\\
&\iff\langle\mathbb{\Delta}[(\vec{x})N],\ N\sigma,\ \mathbb{T}[(\vec{x})N]\rangle\Downarrow^{n-k} && (6.17)\\
&\iff\Sigma[(\vec{x})N]\Downarrow^{n-k} . && (6.22)
\end{aligned}$$

The reverse direction is almost identical.

We show the forward direction of (6.21) by complete induction on k. Suppose $\Sigma[(\vec{x})M]\Uparrow$. Then at least one of the following must be the case:

(1) $\Sigma\Uparrow$, or
(2) $\Sigma\rightarrow^{n}\Sigma'\not\rightarrow$, and $\Sigma'[(\vec{x})N]\Uparrow$.

In case (1), the result follows directly. For case (2), by open uniform computation, $\Sigma'\equiv\langle\mathbb{\Gamma},\ \xi\cdot[\vec{y}],\ \mathbb{S}\rangle$. Then the result follows, since (letting $\Sigma''\equiv\langle\mathbb{\Gamma}[(\vec{x})N],\ [\cdot],$

$\mathbb{S}[(\vec{x})N]\rangle)$:

$$
\begin{aligned}
&\langle \mathbb{\Gamma}[(\vec{x})M],\ (\vec{x})M \cdot [\vec{y}],\ \mathbb{S}[(\vec{x})M]\rangle \Uparrow && \\
&\quad \Longrightarrow \langle \mathbb{\Gamma}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M],\ \mathbb{S}[(\vec{x})M]\rangle \Uparrow && (6.18),\ (\doteq\ \text{cong.}) \\
&\quad \Longrightarrow \forall k' < k. \mathbb{\Sigma}''[\mathbb{C}[(\vec{x})N]] \rightarrow^{k'} && (\text{I.H.}) \\
&\quad \Longleftrightarrow \forall k' < k. \mathbb{\Sigma}''[{}^{\checkmark}\mathbb{C}[(\vec{x})N]] \rightarrow^{k'+1} && (\checkmark) \\
&\quad \Longrightarrow \forall k' < k. \mathbb{\Sigma}''[(\vec{x})N \cdot [\vec{y}]] \rightarrow^{k'+1} && (\text{lemma 15}),\ (\text{ass.}) \\
&\quad \Longrightarrow \langle \Gamma[(\vec{x})N],\ (\vec{x})N \cdot [\vec{y}],\ \mathbb{S}[(\vec{x})N]\rangle \rightarrow^{k}. && (\checkmark)
\end{aligned}
$$

The reverse direction is almost identical.

7. Conclusions and future work

We have presented a semantic theory of the stream processors at the heart of the Fudgets toolkit. The theory is built upon a call-by-need language with erratic non-deterministic choice. Our semantic theory correctly models sharing and its interaction with non-deterministic choice, and still contains proof principles for reasoning about recursive programs. We have shown that in this theory, the congruences and reduction rules of Carlsson and Hallgren's proposed stream processor calculus are equivalences and refinements, respectively. The semantic model improves on the calculus of Kutzner and Schmidt-Schauß, and unlike Taylor's theory of core fudgets it models the effects of shared computation.

7.1. Making parallel composition fair

As it stands, parallel composition is not fair: if either operand fails to produce output, then so may the composition as a whole, and an operand's output may never appear as output from the composition. This is because the merging of output streams is erratic. There are three different kinds of "fair" merge one might consider instead. Assuming each accepts two streams *xs* and *ys*, and produces a third, their behaviour may be described as follows:

Fair merge. Every element of *xs* and *ys* appears eventually (with relative order preserved) in the output.
Bottom-avoiding merge. If one of *xs* or *ys* is finite or partial, then every element of the other appears in the output.
Infinity-fair merge. If one of *xs* or *ys* is infinite, then every element of the other appears in the output.

The ideal merge for our purposes is the first: it is fair with respect to infinite lists (i.e., it avoids starvation in both operands), and is also able to avoid non-termination associated with partial streams. The second can avoid non-termination associated with partial lists, but may starve one of its input streams, if both streams are infinite. The third, infinity-fair merge, does not starve infinite input streams, but cannot avoid the divergence inherent in partial streams.

The first merge cannot be implemented with McCarthy's *amb* [38], and would require study in its own right. However, the last two can be implemented by *amb* (see e.g., chapter 2 of [34]). A call-by-need theory for *amb* would have all of the advantages of the current theory, as well as yielding bottom-avoiding implementations of $|$ and ℓ. As stated elsewhere, *amb* is a much more difficult operator to model. The technical development outlined in Section 6 does not carry through for *amb*: the divergence relation is far less tractable. Nevertheless, *amb* warrants further study.

7.2. *Streams as data structures*

In the current implementation, Fudgets are represented by a continuation-like data structure. An interesting difference from the rather direct stream representation used here is that it permits a stream processor to be unplugged from its point of use and moved to another location.[7] This has been used in the Fudget library to implement interaction by drag-and-drop, where the user can actually move a fudget from one place in the program to another. The feature can even be pushed further, allowing stream processors to migrate over networks to implement mobile agents in the full sense. Although the present theory allows stream processors to be passed as data, it does not permit this degree of freedom. We see no particular obstacle in working with this more concrete representation—although it is likely that we would need to impose a type discipline in order to be able to establish the expected laws.

7.3. *Full abstraction of the translation*

We have shown that "expected" equivalences and reduction rules of the stream processor calculus correspond to equivalence and refinements in the translated language. An interesting (and very useful) result would be to show the reverse: that equivalences (refinements) between target terms correspond to equations and reductions in the stream processor calculus. Among other things, this would require us to build an observational theory for the calculus. This is non-trivial, as it seems to require higher-order bisimilarity (since stream processors may be passed as messages to other stream processors), but is nonetheless an interesting avenue for future work.

Appendix A. A small Fudget program

The following Fudget program implements a little counter with a display and two push buttons, which can be used to increment or decrement the counter. The dataflow

[7] This open the possibility to dynamically change the topology of a running program, an important feature in other coordination languages such as MANIFOLD [2].

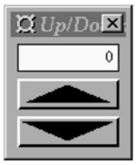
is specified in counterF, which defines a serial composition of a display, a stream processor keeping the integer state, and a parallel composition of the buttons (>==< is serial composition, >+< is parallel composition, where the output is tagged with Left or Right to indicate its source). The buttons output "clicks" when pressed, which are counted by the stream processor, which in turn updates and outputs its internal state to the display. mapstateF is a derived combinator which applies a state transforming function accepting a state and an input message, yielding a new state and a list of output messages.

```
import Fudgets

main = fudlogue (shellF "Up/Down Counter" counterF)

counterF = intDispF >==< mapstateF count 0 >==<
           (buttonF filledTriangleUp >+<
           buttonF filledTriangleDown)
count n (Left Click)  =(n+1,[n+1])
count n (Right Click) =(n-1,[n-1])
```

When run, this small program pops up a window like this:



References

[1] G.A. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, J. Functional Program. 7 (1997) 1–72.

[2] F. Arbab, I. Herman, P. Spilling, An overview of MANIFOLD and its implementation, Concurrency: Practice Exper. 5 (1) (1993) 23–70.

[3] Z.M. Ariola, M. Felleisen, The call-by-need lambda calculus, J. Functional Program. 7 (3) (1997) 265–301.

[4] Z. Ariola, M. Felleisen, J. Maraist, M. Ordersky, P. Walder, A call-by-need lambda calculus, in: Proc. POPL'95, ACM Press, New York, 1995, pp. 233–246.

[5] A. Arnold, M. Nivat, Metric interpretations of infinite trees and semantics of non-deterministic recursive programs, Theoret. Comput. Sci. 11 (1980) 181–205.

[6] G. Berry, G. Boudol, The chemical abstract machine, Theoret. Comput. Sci. 96 (1) (1992) 217–248.

[7] G. Boudol, A lambda-calculus for parallel functions, Rapports de Recherche No. 1231, INRIA Sophia-Antipolis, May 1990.

[8] M. Broy, A theory for nondeterminism, parallelism, communication, and concurrency, Theoret. Comput. Sci. 45 (1) (1986) 1–61.

[9] M. Carlsson, T. Hallgren, Fudgets—purely functional processes with applications to graphical user interfaces, Ph.D. thesis, Department of Computing Sciences, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, March 1998.

[10] W. Clinger, Nondeterministic call by need is neither lazy nor by name, in: Lisp and Functional Programming, 1982, pp. 226–234.

[11] U. de'Liguoro, A. Piperno, Must preorder in non-deterministic untyped λ-calculus, in: CAAP '92, Lecture Notes in Computer Science, vol. 581, 1992, pp. 203–220.

[12] U. de'Liguoro, A. Piperno, Non-deterministic extensions of untyped λ-calculus, Inform. Comput. 122 (2) (1995) 149–177.

[13] H. Erdogmus, R. Johnston, M. Ferguson, On the operational semantics of nondeterminism and divergence, Theoret. Comput. Sci. 159 (2) (1996) 271–317.

[14] W. Ferreira, M. Hennessy, Towards a semantic theory of CML (extended abstract), in: J. Wiedermann, P. Hajek (Eds.), Proc. MFCS'95, Lecture Notes in Computer Science, vol. 969, Springer, Berlin, 1995.

[15] W. Ferreira, M. Hennessy, A. Jeffrey, A theory of weak bisimulation for core CML, in: Proc. ICFP'96, ACM Press, New York, 1996, pp. 201–212.

[16] S. Finne, S.P. Jones, Composing Haggis, in: Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics, 1995.

[17] E.R. Gansner, J.H. Reppy, eXene, in: Proc. 1991 CMU Workshop on Standard ML, Carnegie Mellon University, 1991.

[18] A.D. Gordon, A.M. Pitts (Eds.), Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, Cambridge University Press, Cambridge, 1998.

[19] M.C.B. Hennessy, The semantics of call-by-value and call-by-name in a nondeterministic environment, SIAM J. Comput. 9 (1) (1980) 67–84.

[20] M.C.B. Hennessy, E.A. Ashcroft, A mathematical semantics for a nondeterministic typed λ-calculus, Theoret. Comput. Sci. 11 (1980) 227–245.

[21] M. Hennessy, E.A. Ashcroft, The semantics of non-determinism, in: Automata, Languages and Programming, 1973.

[22] T. Hildebrandt, P. Panagaden, G. Winskel, A relational model of non-deterministic dataflow, in: CONCUR '98, Lecture Notes in Computer Science, vol. 1466, Springer, Berlin, 1998.

[23] S.P. Jones, A. Gordon, S. Finne, Concurrent Haskell, in: Proc. POPL'96, ACM Press, New York, 1996, pp. 295–308.

[24] S.P. Jones, W. Partain, A. Santos, Let-floating: moving bindings to give faster programs: in: Proc. ICFP'96, ACM Press, New York, 1996, pp. 1–12.

[25] S.P. Jones, A. Santos, A transformation-based optimiser for Haskell, Sci. Comput. Program. 32 (1–3) (1998) 3–47.

[26] B. Jonsson, J.N. Kok, Comparing two fully abstract dataflow models, in: E. Odijk, M. Rem, J.-C. Syre (Eds.), Proc. Conf. on Parallel Architectures and Languages Europe, Lecture Notes in Computer Science, vols. 2, 366, Springer, Berlin, 1989, pp. 217–234.

[27] A. Kutzner, M. Schmidt-Schauß, A non-deterministic call-by-need lambda calculus, in: Proc. 1CFP'98, ACM Press, New York, 1998, pp. 324–335.

[28] S.B. Lassen, Relational reasoning about functions and nondeterminism, Ph.D. Thesis, Department of Computer Science, University of Aarhus, May 1998.

[29] S.B. Lassen, A.K. Moran, Unique fixed point induction for McCarthy's Amb, in: Proc. MFCS'99, Lecture Notes in Computer Science, vol. 1672, Springer, Berlin, 1999, pp. 198–208.

[30] J. Launchbury, A natural semantics for lazy evaluation, in: Proc. POPL'93, ACM Press, New York, 1993, pp. 144–154.

[31] I. Mason, C. Talcott, Equivalence in functional languages with effects, J. Functional Program. 1 (3) (1991) 287–327.

[32] R. Milner, Fully abstract models of the typed λ-calculus, Theoret. Comput. Sci. 4 (1977) 1–22.

[33] M.W. Mislove, F.J. Oles, A simple language supporting angelic nondeterminism and parallel composition, in: S. Brookes, M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics, 7th Internat. Conf., PA, USA, March 1991, Proc., Lecture Notes in Computer Science, vol. 598, Springer, Berlin, 1992, pp. 77–101.

[34] A.K. Moran, Call-by-name, call-by-need, and McCarthy's Amb, Ph.D. Thesis, Department of Computing Sciences, Chalmers University of Technology, Gothenburg, Sweden, September 1998.

[35] A.K. Moran, D. Sands, Improvement in a lazy context: an operational theory for call-by-need (extended version), extended version of [36]; available from the authors (Nov. 1998).

[36] A.K. Moran, D. Sands, Improvement in a lazy context: an operational theory for call-by-need, in: Proc. POPL'99, ACM Press, New York, 1999, pp. 43–56.

[37] F. Nielson, H.R. Nielson, From CML to its process algebra, Theoret. Comput. Sci. 155 (1) (1996) 179–219.

[38] P. Panangaden, V. Shanbhogue, McCarthy's amb cannot implement fair merge, Technical Report, TR88-913, Cornell University, Computer Science Department, May 1988.

[39] P. Panangaden, E.W. Stark, Computations, residuals, and the power of indeterminancy, in: ICALP '88, Lecture Notes in Computer Science, vol. 317, Springer, Berlin, 1988, pp. 439–454.

[40] D. Park, On the semantics of fair parallelism, in: Abstract Software Specifications, Lecture Notes in Computer Science, vol. 86, Springer, Berlin, 1980, pp. 504–526.

[41] D. Park, The fairness problem and nondeterministic computing networks, Mathematisch Centrum, Amsterdam, 1983, pp. 133–162.

[42] J. Peterson, K. Hammond, The Haskell Report, Version 1.4, Technical Report, Yale University, 1997.

[43] G.D. Plotkin, A powerdomain construction, SIAM J. Comput. 5 (3) (1976) 452–487.

[44] A.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.

[45] J.H. Reppy, CML: a higher-order concurrent language, in: Proc. PLDI'91, SIGPLAN Notices, vol. 26, ACM Press, New York, 1991, pp. 294–305.

[46] D. Sands, A naïve time analysis and its theory of cost equivalence, J. Logic Comput. 5 (4) (1995) 495–541.

[47] D. Sands, Improvement theory and its applications, in: Gordon, Pitts (Eds.), Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, Cambridge University Press, Cambridge, 1998, pp. 275–306.

[48] D. Sands, Computing with contexts: a simple approach, in: A.D. Gordan, A.M. Pitts, C.L. Talcott (Eds.), Proc. HOOTS II, Electronic Notes in Theoretical Computer Science, vol. 10, Elsevier Science Publishers, Amsterdam, 1998.

[49] D. Sangiorgi, The lazy lambda calculus in a concurrency scenario, Inform. Comput. 111 (1) (1994) 120–153.

[50] P. Sestoft, Deriving a lazy abstract machine, J. Functional Program. 7 (3) (1997) 231–264.

[51] H. Søndergaard, P. Sestoft, Non-determinism in functional languages, The Comput. J. 35 (5) (1992) 514–523.

[52] M.B. Smyth, Power domains, J. Comput. System Sci. 16 (23–26) (1978) 23–35.

[53] M.B. Smyth, Power domains and predicate transformers: a topological view, in: ICALP'83, Lecture Notes in Computer Science, vol. 154, 1983, pp. 662–676.

[54] C.L. Talcott, Reasoning about functions with effects, in: Gordon, Pitts (Eds.), Higher Order Operational Techniques in Semantics, Publications of the Newton Institute, Cambridge University Press, Cambridge, 1998, pp. 347–390.

[55] C. Taylor, Formalising and reasoning about Fudgets, Ph.D. Thesis, School of Computer Science and Information Technology, University of Nottingham, October 1998.

[56] C. Taylor, A theory for core Fudgets, in: Proc. ICFP'98, ACM Press, New York, 1998, pp. 75–85.