

Computing with Contexts

IAN A. MASON

iam@turing.une.edu.au

School of Mathematical and Computer Sciences, University of New England, NSW, Australia, 2351

Received 1997; Revised 1999

Editor: Carolyn Talcott

Abstract. We investigate a representation of contexts, expressions with holes in them, that enables them to be meaningfully transformed, in particular α -converted and β -reduced. In particular we generalize the set of λ -expressions to include holes, and on these generalized entities define β -reduction (i.e. substitution) and filling so that these operations preserve α -congruence and commute. The theory is then applied to allow the encoding of reduction systems and operational semantics of call-by-value calculi enriched with control, imperative and concurrent features.

Keywords: λ -calculus, contexts, operational semantics, theorem proving.

1. Introduction

Contexts, expressions with holes in them, play an increasingly important role in the lambda calculus and its applications to Computer Science. We list some recent examples of their use: They provide an elegant description of reduction strategies (e.g. lazy, left-first depth first) via evaluation and reduction contexts [8]. They feature in the operational semantics of control, imperative and concurrent features [7, 18, 3]. They are center stage in the definition of operational equivalence [20]. They are similarly prominent in the characterization of operational equivalence via context lemmas [19]. They have recently been used to great effect as modalities akin to Hoare triples in logics for functional programming languages with effects [12]. Even more recently they have been used as a powerful technique for establishing reasoning principles for a very general class of programming languages [27].

1.1. Related Approaches

There has also been a lot of recent interest in developing theories and calculi for manipulating and computing with contexts, and closely related concepts. We provide a few notable examples. Talcott [25, 26] presents a theory of (de Bruijn) binding structures akin to abstract algebras. Her notion of binding structures extends abstract data types by incorporating binding mechanisms and structures with holes. She provides three applications of her theory: term rewriting; a unification theory for binding structures; and a basis for an inference system for quantified first order logic. Her work is motivated by building tools such as theorem provers, rewriters, static analyzers, evaluators and transformers. Sands [23] outlines the use of higher-order syntax to represent and compute with contexts in a very general manner. In this approach holes are represented by function variables applied to the sequence of variables that the hole is supposed to trap. Filling is then represented by

meta-syntactic application. A consequence of this is that holes in this approach have a fixed arity, and the composition of contexts in non-trivial, and non-uniform. We examine, in more detail, the relation between Sands approach and the one we adopt shortly. Abadi, Cardelli, Curien, and Levy [1] have developed (de Bruijn) calculi for reasoning about explicit substitutions. These calculi propagate substitutions generated from beta-reductions downwards until reaching a hole. These substitutions accumulate at holes, but do not necessarily have a unique syntactic form, since they depend on the order in which they reached the hole. Consequently for these systems confluence is an important question, and one that is the subject of many papers. Hasimoto and Ohori [11] propose a calculus extending the λ -calculus which includes first class contexts. The calculus incorporates mechanisms for context formation (context abstraction) and hole filling (context application). They also incorporate a type system which they use to prove confluence. Their system is similar to ours in that they annotate holes by substitutions. Sato, Sakurai, and Burstall [24] introduce a simply typed λ -calculus with first class environments, as well as the ability to evaluate terms in a particular environment. They establish confluence, subject reduction, conservativity over the simply typed λ -calculus, and strong normalizability. Dami [6] develops a λ -calculus for dynamic binding that allows the representation of contexts. Dami uses his calculus to model various forms of dynamic binding that occur in the modern programming languages, as well as to represent contexts in the traditional calculus. He also establishes that his encoding of contexts within his calculus commutes with the operation of filling a context with another term (which also may be a context). Lee and Friedman [14] propose an enrichment of the λ -calculus with a notion of contexts. In their approach contexts are thought of as *source code*, while the terms of the λ -calculus are thought of as *target code*. Hole filling is then represented as a compilation process. They use their calculus to represent program modules and their linking. They also simulate other approaches, such as Dami's above.

1.2. Our Approach

The purpose of this note is *not* to present another calculus of contexts akin to the latter five approaches above, but like Sands and Talcott, provide a representation of contexts within the framework of the λ -calculus, and show how, using such a representation, one can compute with such contexts. Thus our approach is to show how contexts can be represented and manipulated at the meta-level using such a representation. One can define contexts in such a way as to enable them to be manipulated, in particular α -converted and β -reduced. In particular we generalize the set of lambda expressions to include holes, and on these generalized entities define β -reduction (i.e. substitution) and filling.

We begin by describing the most obvious of problems created by considering contexts. Typically a hole in the λ -calculus is written as $[\]$. Thus perhaps the simplest, non-trivial context, is the term C , where C is the term $\lambda x. [\]$. Filling this context – i.e replacing the hole by a term e – is written $C[e]$, and in this case is the term $\lambda x. e$, where free occurrences of the variable x in e are now bound by the outermost abstraction λx .

Immediately, α -conversion becomes problematic, at least in the sense that either it does not commute with filling, or substitution becomes partial. Since, either $C = \lambda x. [\]$ is α -congruent to $C' = \lambda z. [\]$, but $C[x] = \lambda x. x$ which is clearly distinct from $C'[x] = \lambda z. x$, or

else we cannot α -convert variables which have a hole in their scope, and are thus prevented from substituting into such terms. For example, consider the term/context C denoting $\lambda z.z(y([\]))$. Suppose further that we wished to substitute z for the free occurrence of y in C , i.e. form $C[y := z]$, to do this we must prevent capture resulting in a term, C'' , such as $\lambda w.w(z([\]))$. But by the previous example our choice of w is plainly visible since $C'''[w]$ is $\lambda w.w(z(w))$, while for any other variable x , if we fill C'' with x the result is $\lambda w.w(z(x))$. These terms are clearly distinct.

Computing – by which we mean β -conversion – is equally problematic. Suppose we apply our term/context C above to z , forming the term/context $(\lambda x.([\]))(z)$. If we follow the usual notion of β -conversion we may reduce $(\lambda x.([\]))(z)$ to $[[x := z]]$. If this term is to be simplified (staying within the λ -calculus), then we really have only one sensible alternative, $[\]$. Thus β -conversion and filling are at odds, since $(\lambda x.([\]))(z)[x]$ reduces to z , whereas $[[x]$ is simply x . Thus the two paths in the diagram:

$$\begin{array}{ccc} (\lambda x.([\]))(z)[x] & \xrightarrow{[\]} & (\lambda x.x)(z) \\ \downarrow \beta & & \downarrow \beta \\ [[x] & \xrightarrow{[\]} & x \neq z \end{array}$$

do not result in the same term. Our approach is to enrich contexts so that other alternatives are available.

Before we describe our approach, it is instructive to examine the one proposed by Sands (which he attributes to Andy Pitts [21], but in fact can be traced back to the earlier work of Aczel [2] and Klop [13]). In his approach a hole is represented by a (meta) application of some function variable ξ to a vector of variables. Each function variable has a specific arity, which determines the length of these vectors of variables. So for example the classical context $\lambda x.([\])$ would be represented by the term $\lambda x.(\xi(x))$. Hole filling is then represented by substituting a meta-abstraction for ξ . Thus filling $\lambda x.([\])$ with the variable x would be represented by the substitution $\lambda x.(\xi(x))[\xi := \mathbf{\Lambda}z.z]$, here $\mathbf{\Lambda}z.z$ represents the (meta-syntactic) identity function. Thus:

$$\begin{aligned} (\lambda x.([\])[x] &\equiv \lambda x.(\xi(x))[\xi := \mathbf{\Lambda}z.z] \\ &\equiv \lambda x.(\mathbf{\Lambda}z.z)(x) \\ &\equiv \lambda x.x \end{aligned}$$

Note that in this approach it is possible to identify contexts upto α -equivalence. Also note the trappings at a hole ξ are made explicit at the hole – in the vector of variables, rather than implicit in the surrounding expression. Sands goes on to show that hole filling, in this representation, commutes with many other relations other than just α -equivalence, in particular substitution. Note that this is a somewhat *static* approach. Holes have fixed arities, consequently filling one context with another does not correspond to the classical case. To see this consider the two contexts $C_0 = \lambda x.([\])$ and $C_1 = \lambda y.([\])$, the composition being: $C_1[C_0] = \lambda y.\lambda x.([\])$. In Sands' representation the corresponding contexts are: $C_0 = \lambda x.\xi(x)$, $C_1 = \lambda y.\xi(y)$, and $C_1[C_0] = \lambda y.\lambda x.\xi(x, y)$. but to obtain the composition one must substitute $\mathbf{\Lambda}z.\lambda x.\xi(x, z)$, which although related to the representation of C_0 , remains

distinct.

$$\begin{aligned} C_1[C_0] &\equiv (\lambda y. \xi(y))[\xi := \mathbf{\Lambda}z. \lambda x. \xi(x, z)] \\ &\equiv (\lambda y. (\mathbf{\Lambda}z. \lambda x. \xi(x, z)))(y) \\ &\equiv \lambda y. \lambda x. \xi(y, z) \end{aligned}$$

Our solution is similar to Sands' but somewhat richer and more dynamic (we will provide a more comprehensive comparison later). We generalize the notion of a hole, so that (occurrences of) holes are decorated with substitutions. This is to enable one to substitute into expressions containing holes. In this paper our holes will be black, \bullet , and we use σ to denote substitutions (finite maps from variables to expressions). The generalized holes are holes decorated by (generalized) substitutions. These syntactic entities are written thus:

$$\bullet^\sigma.$$

We also modify the way one handles trappings at holes. Rather than make the trappings at holes implicit in the surrounding expression, we make them explicit at the hole itself. To elaborate a little on this, consider the obvious analogy between our generalized holes, and classical contexts, suppose that $\text{Dom}(\sigma) = \bar{x}$, then

$$\bullet^\sigma \text{ is analogous to } (\lambda x_0, \dots, x_n. []) (\sigma(x_0), \dots, \sigma(x_n)).$$

Where this analogy breaks down is when these two terms are embedded in larger expressions. In our approach it is only the \bar{x} that will be captured in filling, whereas in the classical case no such distinction is made. Consider the following simple example, where σ is the substitution $\{x := w\}$, and consider the analogous terms:

$$\lambda z. \bullet^{\{x:=w\}} \text{ versus } \lambda z. ((\lambda x. [])w)$$

Filling each term with the expression $x(z)$ will give rise to the distinct terms:

$$\lambda y. w(z) \text{ as opposed to } \lambda z. w(z).$$

This is what we mean when we say that we make the trappings at a hole explicit at the hole, rather than implicit in the surrounding expression. Following this line of reasoning provides a simple comparison with Sands' approach. Since these *innermost* λ s are to be treated differently, let us make this apparent by using the *meta-lambda* notation.

$$\bullet^\sigma \text{ is analogous to } (\mathbf{\Lambda}x_0, \dots, x_n. []) (\sigma(x_0), \dots, \sigma(x_n)).$$

Now to make this analogy accurate we have to define filling, by an expression e , of the right hand side term in such a way that only *meta*-bound variables become trapped. In other words we treat filling in the same fashion we would if we simply first replaced $(\mathbf{\Lambda}x_0, \dots, x_n. [])$ by ξ in the right hand side term, resulting in the term

$$\xi(\sigma(x_0), \dots, \sigma(x_n)),$$

then formed the *meta-abstraction* $\mathbf{\Lambda}x_0, \dots, x_n. e$ (trapping free occurrences of the \bar{x} in e) and then carried out the substitution $\{\xi := \mathbf{\Lambda}x_0, \dots, x_n. e\}$, avoiding free variable capture.

Thus Sands' approach can be seen as a special, and somewhat restricted, case of our general approach.

We investigate two notions of substitution and hole filling in this generalized setting which (for want of better terms) we shall call *weak* and *strong* substitution and hole filling, respectively. The two notions of substitution and filling differ only in what takes place at holes. $e[x := e_0]_w$ is the result of *weakly* substituting e_0 for the free occurrences of the variable x in e , taking care not to trap free variables of e_0 . $e[x := e_0]_s$ is the result of *strongly* substituting e_0 for the free occurrences of the variable x in e , again taking care not to trap free variables of e_0 . They differ in that $\bullet^\sigma[x := e_0]_w$ will not alter the domain of the substitution, σ , decorating the hole. Whereas $\bullet^\sigma[x := e_0]_s$ may possibly add x to its domain, if it is not already included. This dichotomy carries over into the fine details of the definition of hole filling. Given an expression e that may contain holes, we let $e[\llbracket \bullet \rrbracket]_w$ be the result of *weakly* filling each occurrence of a hole by the expression e_0 , and $e[\llbracket \bullet \rrbracket]_s$ be the result of *strongly* filling each occurrence of a hole by the expression e_0 . Both these operations are different from substitution in that the trapping of free variables can occur. To see where the dichotomy arises in the case of filling, consider filling the expression \mathcal{C} by some expression e . The result, intuitively should be obtained from e by carrying out the substitutions described by σ (possibly modified to reflect the filling of holes in the range of σ). Depending upon what type of substitution we understand this to be, i.e. strong or weak, we obtain the corresponding notion of filling.

Our results can be summarized as follows. We use weak substitution to define a notion of α -congruence, $\overset{\alpha}{\equiv}$, for contexts that ensures that both forms of filling and substitution preserve α -congruence. Substitution and filling are defined in such a way to ensure that strong filling commutes with strong substitution. β -reduction, if it is to be functional with respect to α -congruence, must then be defined via the weak form of substitution, and it is shown that β -reduction and either form of hole filling commute. The composition of contexts in the classical sense corresponds to the strong version of filling in our approach. Note that in Sands' approach each hole has a fixed arity. Thus, a priori, his form of substitution and filling correspond to the weak versions. It is for this reason that we state that his approach is static, whereas (the strong versions of) ours are dynamic.

Our approach adopted here is based on the more abstract de Bruijn theory presented in [25, 26]. However we have found it necessary to deviate somewhat from her approach. Firstly, Talcott makes a distinction between free and bound variables. Her free variables are just like ours, her bound variables however do not have names, rather they are identified by the place, suitably described, at which they are bound. To explain β -conversion she defines an operation of *unbinding* a bound variable. By using a de Bruijn notation for variables Talcott avoids the problems associated with α -conversion. In particular both substitution and hole filling are of the strong variety in our sense.

Since this world is more complex than the world without holes we shall take care to define such notions as the *free*, *bound*, *trapped* and *captured* variables of an expression. For example, consider the following skeleton of a term e :

$$\lambda z \dots \lambda x \dots \bullet^{\{z := \dots x \dots z \dots x \dots\}} \dots$$

for the sake of argument let us assume that neither x nor z occur elsewhere in e . In this expression all occurrences of x are bound, as is the rightmost occurrence of z . The oc-

currence of z in the domain of the substitution (second from the right) is *neither* free nor bound; rather it is what we shall call a *trapped* variable. The occurrences of x and z in the range of the substitution are called *captured* variables. Note that the terms *trapped* and *captured* have different meanings. α -congruence is defined in such a way that:

$$\lambda z \dots \lambda x \dots \bullet \{z := \dots x \dots z \dots x \dots\} \dots \quad \equiv \quad \lambda w \dots \lambda y \dots \bullet \{z := \dots y \dots w \dots y \dots\} \dots$$

and filling is defined so that filling either with z yields (upto α -congruence)

$$\lambda w \dots \lambda y \dots \bullet \dots y \dots w \dots y \dots$$

In this paper we shall treat contexts simply as special types of expressions. Namely those which contain these decorated holes.

1.3. Overview

The paper is organized as follows. In §2.1 we introduce the syntax and structure of the terms of our language. Notions such as the rank, the free, bound, trapped and captured variables of a term are explained. In §2.2 we introduce weak substitution and use it to clarify the concept of α -equivalence. We also establish the basic properties of these two notions. §2.3 concentrates on the notions of trapping and the strong version of hole filling, we also establish their basic properties. §2.4 is dedicated to the stronger notion of substitution. §2.5 investigates analogous notion of weak filling. §2.6 examines the relationships between our contexts, traditional contexts, and those of Sands. §3 provides applications of the theory. In particular we define β -conversion and present five variations on the λ calculus. §4 contains a discussion of related work, and suggestions for future work.

1.4. The Proofs of Claims

All the proofs are by induction on the rank of the expressions involved. The proofs of §2.2 are somewhat delicate and so we outline their form in some detail. Of the remaining proofs we include only very brief synopses. Without exception the application case is never interesting. The λ case is also almost always formulaic. The base cases are either obviously true, or a consequence of unwinding the definitions. It is usually only what takes place at holes which warrants any real attention. Indeed if one looks back at the examples (in particular the counterexamples) one sees that if something goes wrong it invariably goes wrong at holes. However even what takes place at holes can be entirely routine. Full proofs may be found in [17].

1.5. Notation

We use the usual notation for set membership and function application. $[X \rightarrow Y]$ is the set of functions from X to Y . $[X \xrightarrow{\omega} Y]$ is the set of functions f whose domain, $\text{Dom}(f)$, is a finite subset of X and whose range, $\text{Rng}(f)$, is contained in Y . $\mathbf{P}_\omega(X)$ is the collection of all finite subsets of X . For any function f , $f\{y := y'\}$ is the function f' such that

$\text{Dom}(f') = \text{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \text{Dom}(f)$. $\mathbb{N} = \{0, 1, 2, \dots\}$ is the natural numbers and i, j, n, n_0, \dots range over \mathbb{N} . We use \mathbf{A} as a *meta-lambda* to define functions; it is to be distinguished from any object lambdas that may occur.

2. The Syntax and Structure of Terms

2.1. The Syntactic Domains

We begin with a set, \mathbb{A} , of atoms, together with a countably infinite set of variables, \mathbb{X} , well-ordered by \leq . We let a, a_0, \dots range over \mathbb{A} and x, x_0, \dots, y, \dots range over \mathbb{X} . From the given sets we define λ -abstractions, \mathbb{L} , expressions, \mathbb{E} , and substitutions, \mathbb{S} , by induction as the least sets satisfying the following equations:

$$\mathbb{E} = \bullet^{\mathbb{S}} \cup \mathbb{A} \cup \mathbb{X} \cup \mathbb{L} \cup \mathbf{app}(\mathbb{E}, \mathbb{E}) \quad \mathbb{L} = \lambda \mathbb{X}. \mathbb{E} \quad \mathbb{S} = \mathbb{X} \xrightarrow{\omega} \mathbb{E}$$

A substitution, σ , is a finite map from variables to expressions. We write $\{x_0 := e_0, \dots, x_n := e_n\}$ for the substitution σ with domain $\{x_0, \dots, x_n\}$ such that $\sigma(x_i) = e_i$ for $i \leq n$. Note that holes can appear anywhere an expression can. In particular holes can appear inside elements of the range of a substitution.

These syntactic domains are constructed in the usual way: defined inductively by a sequence of monotonically increasing sets $\mathbb{E}_i, \mathbb{L}_i$ and \mathbb{S}_i for $i \in \mathbb{N}$ such that

$$\begin{aligned} \mathbb{E} &= \bigcup_{i \in \mathbb{N}} \mathbb{E}_i \quad \text{where} \quad \mathbb{E}_0 = \mathbb{A} \cup \mathbb{X} \quad \text{and} \quad \mathbb{E}_{i+1} = \{\bullet^{\mathbb{S}_i}\} \cup \mathbb{L}_i \cup \mathbf{app}(\mathbb{E}_i, \mathbb{E}_i) \\ &\quad e, e_0, \dots \text{ range over } \mathbb{E} \\ \mathbb{L} &= \bigcup_{i \in \mathbb{N}} \mathbb{L}_i \quad \text{where} \quad \mathbb{L}_0 = \emptyset \quad \text{and} \quad \mathbb{L}_{i+1} = \lambda \mathbb{X}. \mathbb{E}_i \\ &\quad \lambda x. e, \lambda x_0. e_0, \dots \text{ range over } \mathbb{L} \\ \mathbb{S} &= \bigcup_{i \in \mathbb{N}} \mathbb{S}_i. \quad \text{where} \quad \mathbb{S}_0 = \emptyset \quad \text{and} \quad \mathbb{S}_{i+1} = \mathbb{X} \xrightarrow{\omega} \mathbb{E}_i \\ &\quad \sigma, \sigma_0, \dots \text{ range over } \mathbb{S} \end{aligned}$$

These sets give rise to the standard notion of rank of an expression, $\rho(e)$, (and substitution, $\rho(\sigma)$), being the least $j \in \mathbb{N}$ for which $e \in \mathbb{E}_j$ ($\sigma \in \mathbb{S}_j$ respectively). We begin by defining the free variables of an expression or substitution.

Definition ($\rho(e)$ $\rho(\sigma)$):

$$\rho(e) = \begin{cases} \rho(\sigma) + 1 & \text{if } e = \bullet^{\sigma} \\ 0 & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ 1 + \rho(e_0) & \text{if } e = \lambda z. e_0 \\ 1 + \mathbf{max}(\rho(e_0), \rho(e_1)) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\rho(\sigma) = 1 + \mathbf{max}_{x \in \text{Dom}(\sigma)} \rho(\sigma(x))$$

As usual λ is a binding operator. We now define the *free*, and *trapped* variables of an expression, the entire set of variables that occur in it, and describe those that are *captured*.

Definition (FV(e) FV(σ)):

$$\text{FV}(e) = \begin{cases} \text{FV}(\sigma) & \text{if } e = \bullet^\sigma \\ \{e\} \cap \mathbb{X} & \text{if } e \in \mathbb{X} \cup \mathbb{A} \\ \text{FV}(e_0) - \{z\} & \text{if } e = \lambda z.e_0 \\ \text{FV}(e_0) \cup \text{FV}(e_1) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\text{FV}(\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{FV}(\sigma(x))$$

Note that for any $\sigma \in \mathbb{S}$ and any $e \in \mathbb{E}$ we have that $\text{FV}(\sigma)$ and $\text{FV}(e)$ are defined in terms of syntactic entities of lesser rank, and so in particular this definition is well-founded. Also note that variables in the domain of a substitution decorating a hole in an expression are *not* considered free in that expression. Similarly we define the set of all variables occurring in either an expression or a substitution.

Definition (V(e) V(σ)):

$$\text{V}(e) = \begin{cases} \text{V}(\sigma) & \text{if } e = \bullet^\sigma \\ \{e\} \cap \mathbb{X} & \text{if } e \in \mathbb{X} \cup \mathbb{A} \\ \text{V}(e_0) \cup \{z\} & \text{if } e = \lambda z.e_0 \\ \text{V}(e_0) \cup \text{V}(e_1) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\text{V}(\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{V}(\sigma(x)) \cup \text{Dom}(\sigma)$$

We use this definition to define a function from finite sets of syntactic entities to variables called **fresh**:

$$\mathbf{fresh} : \mathbf{P}_\omega(\mathbb{E} \cup \mathbb{S}) \mapsto \mathbb{X}$$

$$\text{where } \mathbf{fresh}(S) = \text{the } \leq\text{-least } x \in \mathbb{X} \text{ such that } x \notin \bigcup_{s \in S} \text{V}(s)$$

The *trapped* variables of either an expression or a substitution are those that appear anywhere in the domain of a substitution that decorates a hole, or in the case of a substitution the domain of the substitution itself:

Definition (TV(e) TV(σ)):

$$\text{TV}(e) = \begin{cases} \text{TV}(\sigma) & \text{if } e = \bullet^\sigma \\ \emptyset & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ \text{TV}(e_0) & \text{if } e = \lambda z.e_0 \\ \text{TV}(e_0) \cup \text{TV}(e_1) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\text{TV}(\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{TV}(\sigma(x)) \cup \text{Dom}(\sigma)$$

The definition of a *captured* variable (or one that possibly can be captured) is more complex, but fortunately we can safely keep it informal: an occurrence of a variable is *captured* if it occurs free in the range of a substitution that decorates a hole, which in turn occurs in the scope of a λ binding that variable.

2.2. Weak Substitution and α -Equivalence

We will let $e[\sigma]_{\mathbb{W}}$ be the result of simultaneous *weak substitution* of free occurrences of the variables $x \in \text{Dom}(\sigma)$ in e by $\sigma(x)$, taking care not to capture variables. It defined as follows:

Definition ($e[\sigma]_{\mathbb{W}} \quad \sigma_1[\sigma_2]_{\mathbb{W}}$):

$$e[\sigma]_{\mathbb{W}} = \begin{cases} \bullet^{\sigma_1[\sigma]_{\mathbb{W}}} & \text{if } e = \bullet^{\sigma_1} \\ e & \text{if } e \in \mathbb{A} \cup (\mathbb{X} - \text{Dom}(\sigma)) \\ \sigma(e) & \text{if } e \in \text{Dom}(\sigma) \\ \lambda\nu.(e_0[z := \nu]_{\mathbb{W}}[\sigma]_{\mathbb{W}}) & \text{if } e = \lambda z.e_0, \text{ and } \nu = \mathbf{fresh}(\{e, \sigma\}) \\ \mathbf{app}(e_0[\sigma]_{\mathbb{W}}, e_1[\sigma]_{\mathbb{W}}) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\sigma_1[\sigma]_{\mathbb{W}} = \mathbf{A}x \in \text{Dom}(\sigma_1).(\sigma_1(x)[\sigma]_{\mathbb{W}})$$

In the definition we have made use of the function **fresh**, thus ensuring $e[\sigma]_{\mathbb{W}}$ is indeed a function, rather than a relation. Elsewhere in this paper the term *fresh* simply means *any* variable that does not occur in the expressions or substitutions involved (thus dropping the requirement that it be the \leq -least such element of \mathbb{X}). It is important to note that $\text{Dom}(\sigma_1[\sigma]_{\mathbb{W}}) = \text{Dom}(\sigma_1)$, this has the consequence that weak substitution does not compose:

Example (1): $e[\sigma_0]_{\mathbb{W}}[\sigma_1]_{\mathbb{W}} \neq e[\sigma_0[\sigma_1]_{\mathbb{W}}]_{\mathbb{W}}$

Proof of Example (1): Simply take

$$e = z \quad \sigma_0 = \{x := 3\} \quad \sigma_1 = \{z := 4\}$$

Then

$$e[\sigma_0]_{\mathbb{W}}[\sigma_1]_{\mathbb{W}} = z[\{x := 3\}]_{\mathbb{W}}[\{z := 4\}]_{\mathbb{W}} = z[\{z := 4\}]_{\mathbb{W}} = 4$$

$$e[\sigma_0[\sigma_1]_{\mathbb{W}}]_{\mathbb{W}} = z[\{x := 3\}[\{z := 4\}]_{\mathbb{W}}]_{\mathbb{W}} = z[\{x := 3[\{z := 4\}]_{\mathbb{W}}\}]_{\mathbb{W}} = z[\{x := 3\}]_{\mathbb{W}} = z$$

□

The point being that the domain of $\sigma_0[\sigma_1]_{\mathbb{W}}$ is the domain of σ_0 so the substitutions described by σ_1 on those variables in the domain of σ_1 but not in the domain of σ_0 are lost.

When $\sigma = \{x := e\}$ we often write $e_0[x := e]_{\mathbb{W}}$ rather than $e_0[\{x := e\}]_{\mathbb{W}}$, when there is no cause for confusion. Often we will use weak substitution to replace one variable by another, in this case the reader should note that the operation corresponds to a simple renaming (leaving the domains of substitutions) unchanged.

We now define α -congruence on expressions. It agrees with the usual definition on expressions not containing holes and is a simple extension of the usual definition given via weak substitution [5, 22].

Definition ($\overset{\alpha}{\equiv}$): The relation $\overset{\alpha}{\equiv}$ is defined (inductively) by the following rules:

- (0) $\frac{}{a \overset{\alpha}{\equiv} a} \quad a \in \mathbb{A}$
- (1) $\frac{}{x \overset{\alpha}{\equiv} x} \quad x \in \mathbb{X}$
- (2) $\frac{e_0 \overset{\alpha}{\equiv} e_1 \quad e'_0 \overset{\alpha}{\equiv} e'_1}{\mathbf{app}(e_0, e'_0) \overset{\alpha}{\equiv} \mathbf{app}(e_1, e'_1)}$
- (3) $\frac{\sigma_0(x) \overset{\alpha}{\equiv} \sigma_1(x) \quad \text{for each } x \in \text{Dom}(\sigma_i)}{\sigma_0 \overset{\alpha}{\equiv} \sigma_1} \quad \text{provided } \text{Dom}(\sigma_0) = \text{Dom}(\sigma_1)$
- (4) $\frac{\sigma_0 \overset{\alpha}{\equiv} \sigma_1}{\bullet^{\sigma_0} \overset{\alpha}{\equiv} \bullet^{\sigma_1}}$
- (5) $\frac{e_0[x_0 := \nu]_{\mathbb{W}} \overset{\alpha}{\equiv} e_1[x_1 := \nu]_{\mathbb{W}}}{\lambda x_0. e_0 \overset{\alpha}{\equiv} \lambda x_1. e_1} \quad \text{for } \nu \text{ fresh}$

Notice the simplicity of this proof system: each syntactic type has exactly one rule that will establish conclusions of that form, consequently by a simple application of the *inversion* principle we may conclude that if $e_0 \overset{\alpha}{\equiv} e_1$ is provable, then, modulo new variables introduced using rule (5), such a proof is unique. Another simple consequence of this definition is that $\overset{\alpha}{\equiv}$ is an equivalence relation, actually it is a congruence but this requires some proof (c.f. lemma ($\overset{\alpha}{\equiv}$.6) below). A somewhat more involved property is that weak substitution is functional modulo α -equivalence. This we will establish anon. Note that at holes the domain of the decorating substitution remains unchanged, for example:

Example (2): $\lambda z. \bullet^{\{z:=z\}} \overset{\alpha}{\equiv} \lambda \nu. \bullet^{\{z:=\nu\}}$.

Proof of Example (2): Choose ν_0 fresh. Then

- (1) $\nu_0 \overset{\alpha}{\equiv} \nu_0$ by rule ($\overset{\alpha}{\equiv}$.1)
- (2) $\{z := \nu_0\} \overset{\alpha}{\equiv} \{z := \nu_0\}$ from (1) by rule ($\overset{\alpha}{\equiv}$.3)
- (3) $\bullet^{\{z:=\nu_0\}} \overset{\alpha}{\equiv} \bullet^{\{z:=\nu_0\}}$ from (2) by rule ($\overset{\alpha}{\equiv}$.4)
- (4) $\bullet^{\{z:=z\}} [z := \nu_0]_{\mathbb{W}} \overset{\alpha}{\equiv} \bullet^{\{z:=\nu\}} [\nu := \nu_0]_{\mathbb{W}}$ from (3) by definition of $[\cdot]_{\mathbb{W}}$
- (5) $\lambda z. \bullet^{\{z:=z\}} \overset{\alpha}{\equiv} \lambda \nu. \bullet^{\{z:=\nu\}}$ from (4) by rule ($\overset{\alpha}{\equiv}$.5)

□

Two trivial properties of $\overset{\alpha}{\equiv}$ and weak substitution are: the renaming of a variable in an expression results in an expression of the same rank; and that α -equivalent expressions must be of the same rank. These are proved by extremely simple inductions on the ranks of the expressions involved and are omitted from this paper.

Lemma ($\overset{\alpha}{\equiv}$.0):

- (0) $\rho(e_0) = \rho(e_0[x := y]_{\mathfrak{w}})$
- (1) $e_0 \overset{\alpha}{\equiv} e_1 \Rightarrow \rho(e_0) = \rho(e_1)$

Perhaps the most basic and important property of the two definitions is that weak substitution preserves $\overset{\alpha}{\equiv}$ equivalence classes:

$$(e_0 \overset{\alpha}{\equiv} e_1 \wedge \sigma_0 \overset{\alpha}{\equiv} \sigma_1) \Rightarrow e_0[\sigma_0]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_1[\sigma_1]_{\mathfrak{w}}$$

Proving this is somewhat delicate and first requires us to establish several simpler properties:

Proposition ($\cdot[\cdot]_{\mathfrak{w}}$.1):

- (a) $(e'_0 \overset{\alpha}{\equiv} e'_1 \wedge e_0 \overset{\alpha}{\equiv} e_1) \Rightarrow e'_0[x := e_0]_{\mathfrak{w}} \overset{\alpha}{\equiv} e'_1[x := e_1]_{\mathfrak{w}}$
- (b) $(\nu_0 \notin \text{FV}(e) \wedge \nu_0 \neq x \neq \nu_1) \Rightarrow e_0[\nu_0 := \nu_1]_{\mathfrak{w}}[x := e]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_0[x := e]_{\mathfrak{w}}[\nu_0 := \nu_1]_{\mathfrak{w}}$
- (c) $e_0[x := \nu]_{\mathfrak{w}}[\nu := y]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_0[x := y]_{\mathfrak{w}} \quad \text{for } \nu \notin \text{FV}(e_0)$
- (d) $\lambda x.e_0 \overset{\alpha}{\equiv} \lambda \nu.e_0[x := \nu]_{\mathfrak{w}} \quad \nu \text{ fresh}$

The first three properties, (a), (b) and (c) are proved by simultaneous induction on the rank of expression e_0 . The next (d) is a simple consequence, since to establish (d) it suffices (due to rule ($\overset{\alpha}{\equiv}$.5)) to show that

$$e_0[z := \nu_0]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_0[z := \nu]_{\mathfrak{w}}[\nu := \nu_0]_{\mathfrak{w}} \quad \text{for } \nu \text{ and } \nu_0 \text{ fresh.}$$

This is immediate from (c). We may now prove a moderately more elaborate version of the desired fact that weak substitution preserves $\overset{\alpha}{\equiv}$ equivalence classes, namely:

Proposition ($\overset{\alpha}{\equiv}$.1):

- (a) $(e_0 \overset{\alpha}{\equiv} e_1 \wedge \sigma_0 \overset{\alpha}{\equiv} \sigma_1) \Rightarrow e_0[\sigma_0]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_1[\sigma_1]_{\mathfrak{w}}$
- (b) $(\nu_0 \notin \text{V}(\sigma) \wedge \nu_1 \notin \text{Dom}(\sigma)) \Rightarrow e_0[\nu_0 := \nu_1]_{\mathfrak{w}}[\sigma]_{\mathfrak{w}} \overset{\alpha}{\equiv} e_0[\sigma]_{\mathfrak{w}}[\nu_0 := \nu_1]_{\mathfrak{w}}$
- (c) $(\lambda \nu.e_0)[\sigma]_{\mathfrak{w}} \overset{\alpha}{\equiv} \lambda \nu.(e_0[\sigma]_{\mathfrak{w}}) \quad \text{if } \nu \notin \text{V}(\sigma)$

The first two properties, (a) and (b) are proved by simultaneous induction on the rank of expression e_0 . The next (c) is a simple consequence. Now that the most basic principles

have been established, we can be somewhat more freewheeling. For example we can show that α -equivalence is a congruence simply by establishing the following useful derived rule.

Lemma ($\stackrel{\alpha}{\equiv}$.6): The following rule is derivable:

$$(\stackrel{\alpha}{\equiv}.6) \quad \frac{e_0 \stackrel{\alpha}{\equiv} e_1}{\lambda x.e_0 \stackrel{\alpha}{\equiv} \lambda x.e_1}$$

Assume that $e_0 \stackrel{\alpha}{\equiv} e_1$. Then by part (a) of ($\stackrel{\alpha}{\equiv}$.1) we have that $e_0[x := \nu]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e_1[x := \nu]_{\mathfrak{w}}$. Thus by the rule ($\stackrel{\alpha}{\equiv}$.4) we may conclude $\lambda x.e_0 \stackrel{\alpha}{\equiv} \lambda x.e_1$. We can also strengthen part (b) of proposition ($[\cdot]_{\mathfrak{w}}$.1) to the following more useful principle.

Proposition ($[\cdot]_{\mathfrak{w}}$.2): If $(\text{Dom}(\sigma_1) - \text{Dom}(\sigma_0)) \cap \text{FV}(e) = \emptyset$, then

$$e[\sigma_0]_{\mathfrak{w}}[\sigma_1]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e[\sigma_0[\sigma_1]_{\mathfrak{w}}]_{\mathfrak{w}}$$

2.3. Trapping & Strong Filling

To define the process of strong hole filling requires a stronger notion of substitution. However from a technical (and perhaps conceptual) point of view it is simpler if we first introduce a restricted notion, that of *trapping*. This process is quite intuitive, given a finite set of variables X and a syntactic entity Φ (either an expression or a substitution), we define $\mathbf{Trap}(\Phi, X)$ to be the syntactic entity obtained by enlarging the domains of any substitution occurring in Φ (by mapping $x \in X$ to x itself, if x is not already in the domain of the substitution) so that they include X . Formally:

Definition ($\mathbf{Trap}(e, X)$ $\mathbf{Trap}(\sigma, X)$):

$$\mathbf{Trap}(e, X) = \begin{cases} \bullet \mathbf{Trap}(\sigma, X) & \text{if } e = \bullet^{\sigma} \\ e & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ \lambda \nu. \mathbf{Trap}(e_0[z := \nu]_{\mathfrak{w}}, X) & \text{if } e = \lambda z.e_0, \text{ and } \nu \text{ is fresh} \\ \mathbf{app}(\mathbf{Trap}(e_0, X), \mathbf{Trap}(e_1, X)) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\mathbf{Trap}(\sigma, X) = \Lambda y \in \text{Dom}(\sigma) \cup X. \begin{cases} \mathbf{Trap}(\sigma(y), X) & \text{if } y \in \text{Dom}(\sigma) \\ y & \text{if } y \in X - \text{Dom}(\sigma) \end{cases}$$

Observe that if an expression e contains no holes, then the process of trapping leaves the expression unchanged modulo α -congruence. The first fact we need to establish is that the process of trapping is functional with respect α -congruence.

Proposition ($\stackrel{\alpha}{\equiv}$.2):

$$e_0 \stackrel{\alpha}{\equiv} e_1 \Rightarrow \mathbf{Trap}(e_0, X) \stackrel{\alpha}{\equiv} \mathbf{Trap}(e_1, X)$$

The proof is a simple induction on $\rho(e_0) = \rho(e_1)$. The second fact establishes that trapping interacts simply with weak substitution and λ -abstraction.

Proposition (Trap(., .).1): If $\nu \notin X$, then

- (a) $\mathbf{Trap}(e, X)[\nu := y]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} \mathbf{Trap}(e[\nu := y]_{\mathfrak{w}}, X)$
- (b) $\mathbf{Trap}(\lambda\nu.e, X) \stackrel{\alpha}{\equiv} \lambda\nu.\mathbf{Trap}(e, X)$

Observe that (b) follows directly from (a). Since

$$\begin{aligned} \mathbf{Trap}(\lambda\nu.e, X) &= \lambda\nu'.\mathbf{Trap}(e[\nu := \nu']_{\mathfrak{w}}, X) && \text{by assumption} \\ &\stackrel{\alpha}{\equiv} \lambda\nu'.\mathbf{Trap}(e, X)[\nu := \nu']_{\mathfrak{w}} && \text{by (a) and rule } (\stackrel{\alpha}{\equiv}.6) \\ &\stackrel{\alpha}{\equiv} \lambda\nu.\mathbf{Trap}(e, X) && \text{by proposition } (.[.]_{\mathfrak{w}}.1) \end{aligned}$$

The proof of (a) is a simple induction on $\rho(e)$. The only delicate parts being the \bullet^σ and λ cases. We are now in a position to define strong hole filling:

Definition ($e[[e_0]]_s$ $\sigma[[e_0]]_s$):

$$e[[e_0]]_s = \begin{cases} \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma[[e_0]]_s]_{\mathfrak{w}} & \text{if } e = \bullet^\sigma \\ e & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ \lambda\nu.(e_1[z := \nu]_{\mathfrak{w}}[[e_0]]_s) & \text{if } e = \lambda z.e_1, \text{ and } \nu \text{ is fresh} \\ \mathbf{app}(e_1[[e_0]]_s, e_2[[e_0]]_s) & \text{if } e = \mathbf{app}(e_1, e_2) \end{cases}$$

$$\sigma[[e_0]]_s = \mathbf{\Lambda}x \in \text{Dom}(\sigma).(\sigma(x)[[e_0]]_s)$$

Before we delve in any further depth into this definition note that strong filling is defined so that capturing, if it is to occur, must be done by the substitution decorating the hole, rather than by the surrounding expression. Examples of this, as well as proof that weak substitution does not commute with strong filling are provided in the following example.

Example (4):

- (a) $\neg(e[[e_0]]_s[\sigma[[e_0]]_s]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e[\sigma]_{\mathfrak{w}}[[e_0]]_s)$
- (b) $(\lambda x.\bullet)[[x]]_s \stackrel{\alpha}{\equiv} \lambda y.x$
- (c) $(\lambda\nu.\bullet^{\{x:=\nu\}})[[x]]_s \stackrel{\alpha}{\equiv} \lambda x.x$

Proof of Example (4): Simply take

$$e = \bullet \quad e_0 = x \quad \sigma = \{x := 3\}$$

Then

$$\begin{aligned} e[[e_0]]_s[\sigma[[e_0]]_s]_{\mathfrak{w}} &= \bullet[[x]]_s[\{x := 3\}[[x]]_s]_{\mathfrak{w}} = x[x := 3]_{\mathfrak{w}} = 3 \\ e[\sigma]_{\mathfrak{w}}[[e_0]]_s &= \bullet[\{x := 3\}]_{\mathfrak{w}}[[x]]_s = \bullet[[x]]_s = x \end{aligned}$$

□₄

There is (hopefully) only one aspect of the definition that requires explanation. Namely, why is it that:

$$\bullet^\sigma \llbracket e_0 \rrbracket_s = \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma \llbracket e_0 \rrbracket_s]_{\mathbf{w}}$$

To anticipate matters slightly we should point out that *strong substitution* will be defined in such a way as to ensure that

$$\bullet^\sigma \llbracket e_0 \rrbracket_s = \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma \llbracket e_0 \rrbracket_s]_{\mathbf{w}} = e_0[\sigma \llbracket e_0 \rrbracket_s]_s.$$

In §2.5 we will investigate a weak notion of filling obtained by modifying this crucial clause to the obviously weaker version:

$$\bullet^\sigma \llbracket e_0 \rrbracket_s = e_0[\sigma \llbracket e_0 \rrbracket_s]_{\mathbf{w}}.$$

In §2.6 we show that this notion of strong filling correspond to composition of classical contexts.

Returning to the discussion at hand, consider the case when $e_0 = \bullet^{\sigma_0}$. To keep things simple we shall assume that neither σ nor σ_0 contain any holes. The idea is that \bullet^σ represents the situation where the substitutions described by σ have taken place at an initially simple hole \bullet . If we had started with \bullet^{σ_0} rather than the simple hole, we would be in a situation best described informally by $(\bullet^{\sigma_0})^\sigma$. By analysis we see that the result of the informal process is \bullet^{σ_1} where

$$\sigma_1 = \mathbf{A}y \in \text{Dom}(\sigma_0) \cup \text{Dom}(\sigma). \begin{cases} \sigma_0(y)[\sigma]_{\mathbf{w}} & \text{if } y \in \text{Dom}(\sigma_0) \\ \sigma(y) & \text{if } y \in \text{Dom}(\sigma) - \text{Dom}(\sigma_0) \end{cases}$$

and it is a simple computation to show that this is α -congruent to the right hand side of the equation under consideration. When there are holes present in either σ or σ_0 the process is slightly more complex. The underlying idea, however, remains the same. For example, consider the following instructive example.

Example (5): Let

$$\begin{aligned} \sigma_0 &= \{x := \bullet^{\{z:=y\}}\} \\ \sigma_1 &= \{w := \bullet^{\{v:=u\}}\} \\ \sigma_2 &= \{z := y, w := \bullet^{\{z:=y, v:=u\}}\} \\ \sigma_3 &= \{x := \bullet^{\sigma_2}, v := u\} \end{aligned}$$

Then

$$\bullet^{\sigma_0} \llbracket \bullet^{\sigma_1} \rrbracket_s = \bullet^{\{x := \bullet^{\sigma_2}, w := \bullet^{\sigma_3}\}}$$

The fully expanded equation is:

$$\begin{aligned} &\bullet^{\{x := \bullet^{\{z:=y\}}\}} \llbracket \bullet^{\{w := \bullet^{\{v:=u\}}\}} \rrbracket_s \\ &= \bullet^{\{x := \bullet^{\{z:=y, w := \bullet^{\{z:=y, v:=u\}}\}}, w := \bullet^{\{x := \bullet^{\{z:=y, w := \bullet^{\{z:=y, v:=u\}}\}}, v:=u\}} \} \end{aligned}$$

Proof of Example (5): Let us first work informally with the left hand side of this equation.

$$\begin{aligned}
& \bullet\{x := \bullet\{z := y\}\} \llbracket \bullet\{w := \bullet\{v := u\}\} \rrbracket_s \\
&= (\bullet\{w := \bullet\{v := u\}\}) \{x := (\bullet\{w := \bullet\{v := u\}\}) \{z := y\}\} \\
&= (\bullet\{w := \bullet\{v := u\}\}) \{x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}\} \\
&= \bullet\{x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}, w := \bullet\{x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}, v := u\}\}
\end{aligned}$$

Now we work with the actual definition.

$$\begin{aligned}
& \bullet\{x := \bullet\{z := y\}\} \llbracket \bullet\{w := \bullet\{v := u\}\} \rrbracket_s \\
&= \mathbf{Trap}(\bullet\{w := \bullet\{v := u\}\}, \{x\}) [x := \bullet\{z := y\} \llbracket \bullet\{w := \bullet\{v := u\}\} \rrbracket_s]_{\mathbb{W}} \\
&= \bullet\{x := x, w := \bullet\{x := x, v := u\}\} [x := \mathbf{Trap}(\bullet\{w := \bullet\{v := u\}\}, \{z\}) [z := y]_{\mathbb{W}}]_{\mathbb{W}} \\
&= \bullet\{x := x, w := \bullet\{x := x, v := u\}\} [x := \bullet\{z := z, w := \bullet\{z := z, v := u\}\} [z := y]_{\mathbb{W}}]_{\mathbb{W}} \\
&= \bullet\{x := x, w := \bullet\{x := x, v := u\}\} [x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}]_{\mathbb{W}} \\
&= \bullet\{x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}, w := \bullet\{x := \bullet\{z := y, w := \bullet\{z := y, v := u\}\}, v := u\}\}
\end{aligned}$$

□

Having convinced ourselves that the definition of strong filtering is correct, we proceed to establish some of its more basic properties. The first being its functionality with respect to α -congruence. As usual the proof is a simple induction on $\rho(e_0) = \rho(e_1)$.

Proposition ($\stackrel{\alpha}{\equiv}$.3):

$$(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge e'_0 \stackrel{\alpha}{\equiv} e'_1) \Rightarrow e_0 \llbracket e'_0 \rrbracket_s \stackrel{\alpha}{\equiv} e_1 \llbracket e'_1 \rrbracket_s$$

Strong filtering also interacts nicely with renaming and λ -abstraction under suitable hygiene conditions. These properties are used in the proofs of theorems (1) and (2) of §2.4.

Lemma ($\llbracket \cdot \rrbracket_s$.1):

- (a) $\nu \notin \text{FV}(e_0) \Rightarrow e[\nu := y]_{\mathbb{W}} \llbracket e_0 \rrbracket_s \stackrel{\alpha}{\equiv} e \llbracket e_0 \rrbracket_s [\nu := y]_{\mathbb{W}}$
- (b) $(\lambda\nu.e) \llbracket e_0 \rrbracket_s \stackrel{\alpha}{\equiv} \lambda\nu.(e \llbracket e_0 \rrbracket_s)$ if $\nu \notin \text{FV}(e_0)$

Proof of Lemma ($\llbracket \cdot \rrbracket_s$.1): First observe that (b) follows directly from (a). Since

$$\begin{aligned}
(\lambda\nu.e) \llbracket e_0 \rrbracket_s &= \lambda\nu'.(e[\nu := \nu']_{\mathbb{W}} \llbracket e_0 \rrbracket_s) \quad \text{by definition of } \llbracket \cdot \rrbracket_s \\
&\stackrel{\alpha}{\equiv} \lambda\nu'.(e \llbracket e_0 \rrbracket_s [\nu := \nu']_{\mathbb{W}}) \quad \text{by (a) and rule } (\stackrel{\alpha}{\equiv} .6) \\
&\stackrel{\alpha}{\equiv} \lambda\nu.(e \llbracket e_0 \rrbracket_s) \quad \text{by proposition } (\llbracket \cdot \rrbracket_s .1)
\end{aligned}$$

The argument for **(a)** is only interesting in the hole subcase of the induction step. Suppose that $e = \bullet^\sigma$. Then

$$\begin{aligned}
e[\nu := y]_{\mathfrak{w}} \llbracket e_0 \rrbracket_{\mathfrak{s}} &= \bullet^\sigma[\nu := y]_{\mathfrak{w}} \llbracket e_0 \rrbracket_{\mathfrak{s}} && \text{by assumption} \\
&= \bullet^{\sigma[\nu := y]_{\mathfrak{w}}} \llbracket e_0 \rrbracket_{\mathfrak{s}} && \text{by definition of } \llbracket \cdot \rrbracket_{\mathfrak{s}} \\
&= \mathbf{Trap}(e_0, \text{Dom}(\sigma[\nu := y]_{\mathfrak{w}}))[\sigma[\nu := y]_{\mathfrak{w}} \llbracket e_0 \rrbracket_{\mathfrak{s}}]_{\mathfrak{w}} && \text{by definition of } \llbracket \cdot \rrbracket_{\mathfrak{s}} \\
&= \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma[\nu := y]_{\mathfrak{w}} \llbracket e_0 \rrbracket_{\mathfrak{s}}]_{\mathfrak{w}} && \text{since } \text{Dom}(\sigma[\nu := y]_{\mathfrak{w}}) = \text{Dom}(\sigma)
\end{aligned}$$

On the other hand

$$\begin{aligned}
e \llbracket e_0 \rrbracket_{\mathfrak{s}}[\nu := y]_{\mathfrak{w}} &= \bullet^\sigma \llbracket e_0 \rrbracket_{\mathfrak{s}}[\nu := y]_{\mathfrak{w}} && \text{by assumption} \\
&= \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma \llbracket e_0 \rrbracket_{\mathfrak{s}}]_{\mathfrak{w}}[\nu := y]_{\mathfrak{w}} && \text{by definition of } \llbracket \cdot \rrbracket_{\mathfrak{s}}
\end{aligned}$$

The conditions of proposition $(\llbracket \cdot \rrbracket_{\mathfrak{w}} \mathbf{2})$ are met since $\nu \in \text{FV}(\mathbf{Trap}(e_0, \text{Dom}(\sigma)))$ iff $\nu \in \text{Dom}(\sigma)$ because by assumption $\nu \notin \text{FV}(e_0)$. By proposition $(\llbracket \cdot \rrbracket_{\mathfrak{w}} \mathbf{2})$

$$\mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma \llbracket e_0 \rrbracket_{\mathfrak{s}}]_{\mathfrak{w}}[\nu := y]_{\mathfrak{w}} = \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma \llbracket e_0 \rrbracket_{\mathfrak{s}}[\nu := y]_{\mathfrak{w}}]_{\mathfrak{w}}$$

So we are reduced to showing that

$$\sigma[\nu := y]_{\mathfrak{w}} \llbracket e_0 \rrbracket_{\mathfrak{s}} \stackrel{\alpha}{=} \sigma \llbracket e_0 \rrbracket_{\mathfrak{s}}[\nu := y]_{\mathfrak{w}}$$

and this again is a simple pointwise application of the induction hypothesis. \square

2.4. Strong Substitution

We are now in a position to define strong substitution.

Definition ($e[\sigma]_{\mathfrak{s}} \quad \sigma_1[\sigma_2]_{\mathfrak{s}}$):

$$e[\sigma]_{\mathfrak{s}} = \begin{cases} \bullet^{\sigma_1[\sigma]_{\mathfrak{s}}} & \text{if } e = \bullet^{\sigma_1} \\ e & \text{if } e \in \mathbb{A} \cup (\mathbb{X} - \text{Dom}(\sigma)) \\ \sigma(e) & \text{if } e \in \text{Dom}(\sigma) \\ \lambda\nu.(e_0[z := \nu]_{\mathfrak{w}}[\sigma]_{\mathfrak{s}}) & \text{if } e = \lambda z.e_0, \text{ and } \nu \text{ is fresh} \\ \mathbf{app}(e_0[\sigma]_{\mathfrak{s}}, e_1[\sigma]_{\mathfrak{s}}) & \text{if } e = \mathbf{app}(e_0, e_1) \end{cases}$$

$$\sigma_1[\sigma]_{\mathfrak{s}} = \mathbf{A}x \in \text{Dom}(\sigma_1) \cup \text{Dom}(\sigma). \begin{cases} \sigma_1(x)[\sigma]_{\mathfrak{s}} & \text{if } x \in \text{Dom}(\sigma_1), \\ \sigma(x) & \text{if } x \in \text{Dom}(\sigma) - \text{Dom}(\sigma_1) \end{cases}$$

A comparison between this definition and that of weak substitution reveals that the solitary difference between the two definitions lies in what takes place at holes. This manifests itself in the following observation:

Observation (1): If e does not contain any holes, then $e[\sigma]_{\mathfrak{s}} = e[\sigma]_{\mathfrak{w}}$.

It is important to note that $\text{Dom}(\sigma_1[\sigma]_{\mathfrak{s}}) = \text{Dom}(\sigma) \cup \text{Dom}(\sigma_1)$, in contrast with weak substitution. This has the following consequence:

Example (6): $\neg(e[x := \nu_0]_s[\nu_0 := \nu_1]_s \stackrel{\alpha}{\equiv} e[x := \nu_1]_s)$ for ν_i fresh.

Proof of Example (6): Simply take $e = \bullet$. Then

$$\begin{aligned} e[x := \nu_0]_s[\nu_0 := \nu_1]_s &= \bullet[x := \nu_0]_s[\nu_0 := \nu_1]_s \\ &= \bullet^{\{x := \nu_0\}}[\nu_0 := \nu_1]_s = \bullet^{\{x := \nu_1, \nu_0 := \nu_1\}} \\ e[x := \nu_1]_s &= \bullet[x := \nu_1]_s = \bullet^{\{x := \nu_1\}} \end{aligned}$$

and clearly $\neg(\{x := \nu_0, \nu_0 := \nu_1\} \stackrel{\alpha}{\equiv} \{x := \nu_1\})$ since they do not have the same domains. \square

An equivalent definition of strong substitution can be given in terms of trapping and weak substitution, as we mentioned in the previous section.

Lemma ($\cdot[\cdot]_s$.1):

$$e[\sigma]_s \stackrel{\alpha}{\equiv} \mathbf{Trap}(e, \text{Dom}(\sigma))[\sigma]_{\mathfrak{w}}$$

The proof is a simple induction on $\rho(e)$, the case of a decorated hole is again the only delicate point. To this end note that if $e = \bullet^{\sigma_0}$, then

$$\begin{aligned} e[\sigma]_s &= \bullet^{\sigma_0}[\sigma]_s = \bullet^{\sigma_0[\sigma]_s} \\ \mathbf{Trap}(e, \text{Dom}(\sigma))[\sigma]_{\mathfrak{w}} &= \bullet^{\mathbf{Trap}(\sigma_0, \text{Dom}(\sigma))}[\sigma]_{\mathfrak{w}} = \bullet^{\mathbf{Trap}(\sigma_0, \text{Dom}(\sigma))[\sigma]_{\mathfrak{w}}} \end{aligned}$$

So it suffices to show that

$$\sigma_0[\sigma]_s \stackrel{\alpha}{\equiv} \mathbf{Trap}(\sigma_0, \text{Dom}(\sigma))[\sigma]_{\mathfrak{w}}$$

This is relatively easy to establish pointwise. An immediate consequence of this is that $\bullet^{\sigma}[[e_0]]_s = e_0[\sigma[[e_0]]_s]_s$. It also follows that strong substitution is functional modulo α -congruence.

Corollary ($\stackrel{\alpha}{\equiv}$.4):

$$(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge \sigma_0 \stackrel{\alpha}{\equiv} \sigma_1) \Rightarrow e_0[\sigma_0]_s \stackrel{\alpha}{\equiv} e_1[\sigma_1]_s$$

Proof of Corollary ($\stackrel{\alpha}{\equiv}$.4): Assume the hypothesis. Then

$$\begin{aligned} e_0[\sigma_0]_s &\stackrel{\alpha}{\equiv} \mathbf{Trap}(e_0, \text{Dom}(\sigma_0))[\sigma_0]_{\mathfrak{w}} && \text{by the previous lemma} \\ &\stackrel{\alpha}{\equiv} \mathbf{Trap}(e_0, \text{Dom}(\sigma_0))[\sigma_1]_{\mathfrak{w}} && \text{by proposition ($\stackrel{\alpha}{\equiv}$.1) of §2.2 since } \sigma_0 \stackrel{\alpha}{\equiv} \sigma_1 \\ &= \mathbf{Trap}(e_0, \text{Dom}(\sigma_1))[\sigma_1]_{\mathfrak{w}} && \text{since } \sigma_0 \stackrel{\alpha}{\equiv} \sigma_1 \text{ implies } \text{Dom}(\sigma_0) = \text{Dom}(\sigma_1) \\ &\stackrel{\alpha}{\equiv} \mathbf{Trap}(e_1, \text{Dom}(\sigma_1))[\sigma_1]_{\mathfrak{w}} \\ &\text{by propositions ($\stackrel{\alpha}{\equiv}$.1) and ($\stackrel{\alpha}{\equiv}$.2) of §2.2 since } e_0 \stackrel{\alpha}{\equiv} e_1 \\ &\stackrel{\alpha}{\equiv} e_1[\sigma_1]_s && \text{again by the previous lemma} \end{aligned}$$

□

Example (7): Suppose that σ_0 contains no holes. Then

$$\bullet^{\sigma_0} \llbracket \bullet^{\sigma_1} \rrbracket_s \stackrel{\alpha}{\equiv} \bullet^{\sigma_1[\sigma_0]}_s$$

Proof of Example (7):

$$\begin{aligned} \bullet^{\sigma_0} \llbracket \bullet^{\sigma_1} \rrbracket_s &= \mathbf{Trap}(\bullet^{\sigma_1}, \text{Dom}(\sigma_0))[\sigma_0 \llbracket \bullet^{\sigma_1} \rrbracket_s]_{\mathbf{w}} && \text{by definition} \\ &= \mathbf{Trap}(\bullet^{\sigma_1}, \text{Dom}(\sigma_0))[\sigma_0]_{\mathbf{w}} && \text{since } \sigma_0 \text{ contains no holes} \\ &= \bullet^{\sigma_1}[\sigma_0]_s && \text{by lemma } (. [.]_s. \mathbf{1}) \\ &= \bullet^{\sigma_1[\sigma_0]}_s && \text{by definition} \end{aligned}$$

□

Lemma (. [.]_s. 2):

- (a) $(\nu_0 \notin (\text{FV}(\sigma) \cup \text{Dom}(\sigma)) \wedge \nu_1 \notin \text{Dom}(\sigma)) \Rightarrow e[\nu_0 := \nu_1]_{\mathbf{w}}[\sigma]_s \stackrel{\alpha}{\equiv} e[\sigma]_s[\nu_0 := \nu_1]_{\mathbf{w}}$
 (b) $(\lambda\nu.e)[\sigma]_s \stackrel{\alpha}{\equiv} \lambda\nu.(e[\sigma]_s) \quad \text{if } \nu \notin (\text{FV}(\sigma) \cup \text{Dom}(\sigma))$

As usual (b) follows directly from (a). To prove (a) observe that

$$\begin{aligned} e[\nu_0 := \nu_1]_{\mathbf{w}}[\sigma]_s &= \mathbf{Trap}(e[\nu_0 := \nu_1]_{\mathbf{w}}, \text{Dom}(\sigma))[\sigma]_{\mathbf{w}} && \text{by lemma } (. [.]_s. \mathbf{1}) \\ &\stackrel{\alpha}{\equiv} \mathbf{Trap}(e, \text{Dom}(\sigma))[\nu_0 := \nu_1]_{\mathbf{w}}[\sigma]_{\mathbf{w}} && \text{by } (\mathbf{Trap}(\cdot, \cdot). \mathbf{1}), \text{ as } \nu_0 \notin \text{Dom}(\sigma) \\ &\stackrel{\alpha}{\equiv} \mathbf{Trap}(e, \text{Dom}(\sigma))[\sigma]_{\mathbf{w}}[\nu_0 := \nu_1]_{\mathbf{w}} && \text{by proposition } (. [.]_{\mathbf{w}}. \mathbf{1}) \\ &= e[\sigma]_s[\nu_0 := \nu_1]_{\mathbf{w}} && \text{by lemma } (. [.]_s. \mathbf{1}) \end{aligned}$$

In contrast to weak substitution, strong substitution satisfies a composition principle.

Proposition (. [.]_s. 3):

$$e[\sigma_0]_s[\sigma_1]_s \stackrel{\alpha}{\equiv} e[\sigma_0[\sigma_1]_s]_s$$

Proposition (. [.]_s. 4):

$$\mathbf{Trap}(e, X) \stackrel{\alpha}{\equiv} e[\mathbf{A}x \in X.x]_s$$

We may collect all the main results of the previous sections into a single theorem:

Theorem (1):

- (1) $(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge \sigma_0 \stackrel{\alpha}{\equiv} \sigma_1) \Rightarrow e_0[\sigma_0]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e_1[\sigma_1]_{\mathfrak{w}}$
- (2) $e_0 \stackrel{\alpha}{\equiv} e_1 \Rightarrow \mathbf{Trap}(e_0, X) \stackrel{\alpha}{\equiv} \mathbf{Trap}(e_1, X)$
- (3) $(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge e'_0 \stackrel{\alpha}{\equiv} e'_1) \Rightarrow e_0[[e'_0]]_s \stackrel{\alpha}{\equiv} e_1[[e'_1]]_s$
- (4) $(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge \sigma_0 \stackrel{\alpha}{\equiv} \sigma_1) \Rightarrow e_0[\sigma_0]_s \stackrel{\alpha}{\equiv} e_1[\sigma_1]_s$

The main result of this section is that, unlike weak substitution, strong substitution commutes with strong filling.

Theorem (2):

$$e[\sigma]_s[[e']]_s \stackrel{\alpha}{\equiv} e[[e']]_s[\sigma[[e']]_s]_s$$

Corollary (2):

$$\mathbf{Trap}(e[[e_0]]_s, \text{Dom}(\sigma))[\sigma[[e_0]]_s]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} \mathbf{Trap}(e, \text{Dom}(\sigma))[\sigma]_{\mathfrak{w}}[[e_0]]_s$$

2.5. *Strong & Weak Filling*

The notion of hole filling forms a dichotomy, precisely in the same way as substitution, according to what takes place at holes. By lemma ($[\cdot]_s$.1) of §2.4 we may define the notion of hole filling studied in §2.3 using strong substitution, thus:

Definition ($e[[e_0]]_s$ $\sigma[[e_0]]_s$):

$$e[[e_0]]_s = \begin{cases} e_0[\sigma[[e_0]]_s]_s & \text{if } e = \bullet^\sigma \\ e & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ \lambda\nu.(e_1[z := \nu]_{\mathfrak{w}}[[e_0]]_s) & \text{if } e = \lambda z.e_1, \text{ and } \nu \text{ is fresh} \\ \mathbf{app}(e_1[[e_0]]_s, e_2[[e_0]]_s) & \text{if } e = \mathbf{app}(e_1, e_2) \end{cases}$$

$$\sigma[[e_0]]_s = \mathbf{\Lambda} x \in \text{Dom}(\sigma).(\sigma(x)[[e_0]]_s)$$

The weak version, called *weak hole filling* and denoted by $[\cdot]_{\mathfrak{w}}$, differs only in what takes place at holes:

Definition ($e[[e_0]]_{\mathfrak{w}}$ $\sigma[[e_0]]_{\mathfrak{w}}$):

$$e[[e_0]]_{\mathfrak{w}} = \begin{cases} e_0[\sigma[[e_0]]_{\mathfrak{w}}]_{\mathfrak{w}} & \text{if } e = \bullet^\sigma \\ e & \text{if } e \in \mathbb{A} \cup \mathbb{X} \\ \lambda\nu.(e_1[z := \nu]_{\mathfrak{w}}[[e_0]]_{\mathfrak{w}}) & \text{if } e = \lambda z.e_1, \text{ and } \nu \text{ is fresh} \\ \mathbf{app}(e_1[[e_0]]_{\mathfrak{w}}, e_2[[e_0]]_{\mathfrak{w}}) & \text{if } e = \mathbf{app}(e_1, e_2) \end{cases}$$

$$\sigma[[e_0]]_{\mathfrak{w}} = \mathbf{\Lambda} x \in \text{Dom}(\sigma).(\sigma(x)[[e_0]]_{\mathfrak{w}})$$

By theorem (1) of §2.4 and the above definitions we know that these operations are functional modulo α -congruence. We should also point out that as a consequence of observation (1) of §2.4 we also have the following fact:

Observation (2): If neither e_0 nor e_1 contain holes, then $e_1 \llbracket e_0 \rrbracket_{\mathbf{w}} = e_1 \llbracket e_0 \rrbracket_{\mathbf{s}}$.

This can be strengthened a little to only requiring that e_0 does not contain any holes, and there are no holes occurring in the range of any substitution that decorates a hole occurring in e_1 . In what follows let $a, b \in \{\mathbf{w}, \mathbf{s}\}$. The first obvious question to ask is for which a and b do we have that:

ab -Commutation

$$e[\sigma]_a \llbracket e' \rrbracket_b \stackrel{\alpha}{\equiv} e \llbracket e' \rrbracket_b [\sigma \llbracket e' \rrbracket_b]_a$$

Theorem (3): ab -Commutation is true iff $a = b = \mathbf{s}$.

Proof of Theorem (3): We need only show that \mathbf{ww} - and \mathbf{sw} -commutation are false, since \mathbf{ss} -commutation is a restatement of theorem (2) of §2.4, while the falsity of \mathbf{ws} -commutation is a restatement of example (4(a)) of §2.3. In both cases let $e = \bullet^{\sigma_0}$ and assume that neither σ_0 nor σ contain any holes. Then in the case of \mathbf{ww} -commutation the l.h.s $e[\sigma]_{\mathbf{w}} \llbracket e' \rrbracket_{\mathbf{w}} \stackrel{\alpha}{\equiv} e'[\sigma_0[\sigma]_{\mathbf{w}}]_{\mathbf{w}}$ while the r.h.s $e \llbracket e' \rrbracket_{\mathbf{w}} [\sigma \llbracket e' \rrbracket_{\mathbf{w}}]_{\mathbf{w}} \stackrel{\alpha}{\equiv} e'[\sigma_0]_{\mathbf{w}}[\sigma]_{\mathbf{w}}$. These are different by example (1) of §2.2. In the case of \mathbf{sw} -commutation the l.h.s $e[\sigma]_{\mathbf{s}} \llbracket e' \rrbracket_{\mathbf{w}} \stackrel{\alpha}{\equiv} e'[\sigma_0[\sigma]_{\mathbf{s}}]_{\mathbf{w}}$ while the r.h.s $e \llbracket e' \rrbracket_{\mathbf{w}} [\sigma \llbracket e' \rrbracket_{\mathbf{w}}]_{\mathbf{s}} \stackrel{\alpha}{\equiv} e'[\sigma_0]_{\mathbf{w}}[\sigma]_{\mathbf{s}}$. The inequality of these two expressions is made obvious by choosing $e' = \bullet$ and σ_0 to be the empty substitution. The r.h.s becomes \bullet^{σ} while the l.h.s remains as \bullet . \square

2.6. Representing Contexts

We make some simple comparisons with our approach and other approaches. In particular we show how to represent traditional contexts, as well as those of Sands.

2.6.1. Representing Classical Contexts We make some simple observations about *classical contexts* and their representation within this framework. Traditionally contexts in the λ calculus are generated from atoms, variables, and the hole $[]$ via the usual operations of abstraction and application. Thus we may define the set of traditional contexts, \mathbb{C} , by the equation:

$$\mathbb{C} = [] \cup \mathbb{A} \cup \mathbb{X} \cup \lambda \mathbb{X}. \mathbb{C} \cup \text{app}(\mathbb{C}, \mathbb{C})$$

In this setting hole filling is simply textual replacement. Our notion of context is richer than the traditional notion, but it is not difficult to define a translation of traditional contexts into our generalized contexts. To this end we define the translation τ from \mathbb{C} into \mathbb{E} as follows:

Definition ($\tau : \mathbb{C} \rightarrow \mathbb{E} \quad \mathbb{E}_{\mathbb{C}}$):

$$\tau(C) = \begin{cases} \bullet & \text{if } C \text{ is } [] \\ C & \text{if } C \in \mathbb{A} \cup \mathbb{X} \\ \lambda x. \mathbf{Trap}(\tau(C_0), \{x\}) & \text{if } C \text{ is } \lambda x. C_0 \\ \mathbf{app}(\tau(C_0), \tau(C_1)) & \text{if } C \text{ is } \mathbf{app}(C_0, C_1) \end{cases}$$

We let $\mathbb{E}_{\mathbb{C}}$ denote the range of τ as a subset of \mathbb{E} . We make a couple of simple observations concerning τ . Firstly, it is well defined. Secondly, on expressions that contain no holes (i.e. terms of the classical calculus) the translation is, up to α -congruence, the identity. Of more interest are the following two facts.

Proposition ($\tau.1$): If $C \in \mathbb{C}$ is a traditional context and $e \in \mathbb{E}$ contains no holes, then

$$\tau(C[e]) \stackrel{\alpha}{\equiv} \tau(C)[\tau(e)]_s \stackrel{\alpha}{\equiv} \tau(C)[e]_s \stackrel{\alpha}{\equiv} \tau(C)[\tau(e)]_w \stackrel{\alpha}{\equiv} \tau(C)[\tau(e)]_w$$

Proposition ($\tau.2$): If $C_0, C_1 \in \mathbb{C}$ are traditional contexts, then

$$\tau(C_0[C_1]) \stackrel{\alpha}{\equiv} \tau(C_0)[\tau(C_1)]_s$$

Finally, consider the following example:

Example (8):

$$C_0 = \lambda x. [] \quad \text{thus} \quad \tau(C_0) = \lambda x. \bullet^{\{x:=x\}}$$

$$C_1 = \lambda y. [] \quad \text{thus} \quad \tau(C_1) = \lambda y. \bullet^{\{y:=y\}}$$

$$C_0[C_1] = \lambda x. \lambda y. [] \quad \text{thus} \quad \tau(C_0[C_1]) = \lambda x. \lambda y. \bullet^{\{x:=x, y:=y\}}$$

$$\tau(C_0)[\tau(C_1)]_w = \lambda x. \lambda y. \bullet^{\{y:=y\}} \quad \text{which is not } \stackrel{\alpha}{\equiv} \text{ to } \tau(C_0[C_1])$$

2.6.2. Representing Sands' Contexts Sands' approach to contexts is couched in a type-theory-style abstract syntax for terms. We will not present that approach here, rather we will present a simple instance of his approach that corresponds to our particular choice of syntax. He also considers a countably infinite number of holes. We will be content to treat one, since their multitude presents no obstacles. So assume that ξ is a hole variable of fixed arity N . Relative to this hole we can define the class of *second order representations* of contexts as:

$$\mathbb{C}_{\xi} = \xi \underbrace{(\mathbb{C}_{\xi}) \dots (\mathbb{C}_{\xi})}_N \cup \mathbb{A} \cup \mathbb{X} \cup \lambda \mathbb{X}. \mathbb{C}_{\xi} \cup \mathbf{app}(\mathbb{C}_{\xi}, \mathbb{C}_{\xi})$$

In this approach filling is restricted to terms of the form $\mathbf{\Lambda}x_1, \dots, x_N. C'_{\xi}$. We use the notation $C_{\xi}[\xi := \mathbf{\Lambda}x_1, \dots, x_N. C'_{\xi}]$ to denote such an operation. It is defined exactly as (capture avoiding) substitution, except for the crucial case:

$$\begin{aligned} & (\xi(C_{\xi}^1) \dots (C_{\xi}^N))[\xi := \mathbf{\Lambda}x_1, \dots, x_N. C'_{\xi}] \\ & = C'_{\xi}[\{x_i := (C_{\xi}^i[\xi := \mathbf{\Lambda}x_1, \dots, x_N. C'_{\xi}]) \mid 1 \leq i \leq n\}] \end{aligned}$$

It is important to note that the *meta-application* $\xi(C_\xi^1) \dots (C_\xi^N)$, is not treated in the same way as other applications within the language. In particular this definition is somewhat freewheeling with respect to the order of evaluation of the C_ξ^i .

As in the previous example, we define a translation τ from \mathbb{C}_ξ into \mathbb{E} . To do this we first fix a collection of distinct variables $\{z_i \mid i \in \mathbb{N}\}$ with which to define the translation.

Definition ($\tau : \mathbb{C}_\xi \rightarrow \mathbb{E}$):

$$\tau(C_\xi) = \begin{cases} \bullet \{z_i := \tau(C_\xi^i) \mid 1 \leq i \leq N\} & \text{if } C \text{ is } \xi(C_\xi^1) \dots (C_\xi^N) \\ C & \text{if } C_\xi \in \mathbb{A} \cup \mathbb{X} \\ \lambda x. \tau(C_\xi^0) & \text{if } C_\xi \text{ is } \lambda x. C_\xi^0 \\ \mathbf{app}(\tau(C_\xi^0), \tau(C_\xi^1)) & \text{if } C_\xi \text{ is } \mathbf{app}(C_\xi^0, C_\xi^1) \end{cases}$$

As before τ , so defined, it is functional up to α -congruence, and on expressions that contain no holes (i.e. terms of the classical calculus) the translation is, up to α -congruence, the identity. The correspondence is then established via the following fact.

Proposition ($\tau.3$): If $C_\xi^0, C_\xi^1 \in \mathbb{C}_\xi$

$$\tau(C_\xi^0[\xi := \mathbf{\Lambda} z_1, \dots, z_N. C_\xi^1]) \stackrel{\alpha}{\equiv} \tau(C_\xi^0) \llbracket \tau(C_\xi^1) \rrbracket_{\mathbf{w}}$$

3. Notions of Reduction

3.1. Classical β -Reduction

The first problem that must be faced is: *In this more general setting is β -reduction defined via strong or weak substitution?* This turns out to be quite easy to answer, although the answer may be surprising. Suppose that β is defined via strong substitution:

$$(\beta.s) \quad \mathbf{app}(\lambda x. e_0, e_1) \xrightarrow{\beta} e_0[x := e_1]_s$$

Then by theorem (2) of §2.4 one would expect β reduction to commute with filling:

$$\begin{array}{ccc} \mathbf{app}(\lambda x. e, e_1) \llbracket e_0 \rrbracket_s & \xrightarrow{\llbracket \cdot \rrbracket_s} & \mathbf{app}((\lambda x. e) \llbracket e_0 \rrbracket_s, e_1 \llbracket e_0 \rrbracket_s) \\ \downarrow \beta.s & & \downarrow \beta.s \\ (e[x := e_1]_s) \llbracket e_0 \rrbracket_s & \xrightarrow{\llbracket \cdot \rrbracket_s} & (e \llbracket e_0 \rrbracket_s)[x := e_1 \llbracket e_0 \rrbracket_s]_s \end{array}$$

So it is somewhat surprising that this is not the case. However before we provide a counterexample we point out an even worse feature: it is not functional modulo α -conversion: Suppose for sake of argument that β -reduction (so defined) does preserve equivalence. Observe by proposition ($\llbracket \cdot \rrbracket_{\mathbf{w}}.1.d$) of §2.2 that for ν fresh

$$\lambda x. e_0 \stackrel{\alpha}{\equiv} \lambda \nu. e_0[x := \nu]_{\mathbf{w}}$$

consequently β reducing the application of either to e_1 yields

$$e_0[x := e_1]_s \stackrel{\alpha}{\equiv} e_0[x := \nu]_{\mathfrak{w}}[\nu := e_1]_s$$

by considering the case when e_0 is the undecorated hole \bullet we are forced into the unpleasant situation of concluding that

$$\bullet\{x := e_1\} \stackrel{\alpha}{\equiv} \bullet\{\nu := e_1\} .$$

On the other hand, suppose that β is defined via weak substitution:

$$(\beta.\mathfrak{w}) \quad \text{app}(\lambda x.e, e_1) \xrightarrow{\beta} e[x := e_1]_{\mathfrak{w}}$$

Then by virtue of proposition ($\stackrel{\alpha}{\equiv}$.1) of §2.2 it is functional modulo α -conversion, but it is not at all not obvious that this form β reduction commutes with filling. Indeed, the fact that weak substitution does *not* commute with filling suggests that the contrary is true. Happily this is an illusion:

Theorem (4):

$$\begin{array}{ccc} \text{app}(\lambda x.e, e_1)[[e_0]]_s & \xrightarrow{\mathbb{I}_s} & \text{app}((\lambda x.e)[[e_0]]_s, e_1[[e_0]]_s) \\ \downarrow \beta.\mathfrak{w} & & \downarrow \beta.\mathfrak{w} \\ (e[x := e_1]_{\mathfrak{w}})[[e_0]]_s & \xrightarrow{\mathbb{I}_s} & (e[[e_0]]_s)[x := e_1[[e_0]]_s]_{\mathfrak{w}} \end{array}$$

This theorem is established by proving the following *weak* form of commutation between weak substitution and filling:

Lemma (3): If ν is fresh, then

$$(e[x := e_1]_{\mathfrak{w}})[[e_0]]_s \stackrel{\alpha}{\equiv} (e[x := \nu]_{\mathfrak{w}})[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}}$$

Proof of Lemma (3): The proof is a simple induction on $\rho(e)$. The only case of any interest is the hole subcase of the induction step.

(Induction Step subcase 2. $e = \bullet^\sigma$) In this case we have by computation

$$\begin{aligned} e[x := e_1]_{\mathfrak{w}}[[e_0]]_s &= (\bullet^\sigma)[x := e_1]_{\mathfrak{w}}[[e_0]]_s \\ &= (\bullet^{\sigma[x := e_1]_{\mathfrak{w}}})[[e_0]]_s \\ &= \mathbf{Trap}(e_0, \text{Dom}(\sigma[x := e_1]_{\mathfrak{w}}))[\sigma[x := e_1]_{\mathfrak{w}}[[e_0]]_s]_{\mathfrak{w}} \\ &= \mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma[x := e_1]_{\mathfrak{w}}[[e_0]]_s]_{\mathfrak{w}} \end{aligned}$$

Similarly:

$$\begin{aligned} e[x := \nu]_{\mathfrak{w}}[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} &= (\bullet^\sigma)[x := \nu]_{\mathfrak{w}}[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \\ &= (\bullet^{\sigma[x := \nu]_{\mathfrak{w}}})[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \\ &= (\mathbf{Trap}(e_0, \text{Dom}(\sigma[x := \nu]_{\mathfrak{w}}))[\sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s]_{\mathfrak{w}})[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \\ &= (\mathbf{Trap}(e_0, \text{Dom}(\sigma))[\sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s]_{\mathfrak{w}})[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \end{aligned}$$

So letting $e' = \mathbf{Trap}(e_0, \text{Dom}(\sigma))$ it suffices to show that

$$e'[\sigma[x := e_1]_{\mathfrak{w}}[[e_0]]_s]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e'[\sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s][\nu := e_1[[e_0]]_s]_{\mathfrak{w}}$$

Notice that by definition $\nu \notin \text{FV}(e')$. Thus by lemma $(\cdot)_{\mathfrak{w}}\mathbf{.2}$ we have

$$e'[\sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s][\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \stackrel{\alpha}{\equiv} e'[\sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s][\nu := e_1[[e_0]]_s]_{\mathfrak{w}}$$

Consequently we need only show that

$$\sigma[x := e_1]_{\mathfrak{w}}[[e_0]]_s \stackrel{\alpha}{\equiv} \sigma[x := \nu]_{\mathfrak{w}}[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}}$$

So choose $z \in \text{Dom}(\sigma)$ arbitrarily. Then

$$\begin{aligned} \sigma[x := e_1]_{\mathfrak{w}}[[e_0]]_s(z) &= \sigma(z)[x := e_1]_{\mathfrak{w}}[[e_0]]_s \\ &\stackrel{\alpha}{\equiv} \sigma(z)[x := \nu]_{\mathfrak{w}}[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} && \text{by the induction hypothesis} \\ &= \sigma(z)[x := \nu]_{\mathfrak{w}}[[e_0]]_s[\nu := e_1[[e_0]]_s]_{\mathfrak{w}} \end{aligned}$$

□

Just as we were in §2.5, we can be somewhat more systematic about the crucial property needed for filling to commute with β reduction.

β -ab-Commutation If ν is fresh, then

$$(e[x := e_1]_a)[[e_0]]_b \stackrel{\alpha}{\equiv} (e[x := \nu]_{\mathfrak{w}})[[e_0]]_b[\nu := e_1[[e_0]]_b]_a$$

To see the significance of β -ab-commutation, assume that we have defined β -reduction as follows:

$$(\beta_a) \quad \mathbf{app}(\lambda x.e_0, e_1) \xrightarrow{\beta} e_0[x := e_1]_a$$

Then there are two obvious methods for computing $\mathbf{app}(\lambda x.e, e_1)[[e_0]]_b$. We can either do the β -reduction first:

$$\mathbf{app}(\lambda x.e, e_1)[[e_0]]_b \xrightarrow{\beta} (e_0[x := e_1]_a)[[e_0]]_b$$

or else do the filling first and then the β -reduction:

$$\begin{aligned} \mathbf{app}(\lambda x.e, e_1)[[e_0]]_b &= \mathbf{app}((\lambda x.e)[[e_0]]_b, e_1[[e_0]]_b) \\ &\stackrel{\alpha}{\equiv} \mathbf{app}((\lambda \nu.e[x := \nu]_{\mathfrak{w}})[[e_0]]_b, e_1[[e_0]]_b) && \text{for } \nu \text{ fresh} \\ &\stackrel{\alpha}{\equiv} \mathbf{app}(\lambda \nu.(e[x := \nu]_{\mathfrak{w}}[[e_0]]_b), e_1[[e_0]]_b) && \text{by lemma } (\cdot)_{\mathfrak{w}}\mathbf{.1b} \text{ of } \S 2.3 \\ &\xrightarrow{\beta} e[x := \nu]_{\mathfrak{w}}[[e_0]]_b[\nu := e_1[[e_0]]_b]_a \end{aligned}$$

that these two methods give the same result is precisely the import of β -ab-commutation. With this in mind the following theorem is of interest.

Theorem (5): β -ab-commutation is true iff $a = \mathfrak{w}$.

Proof of Theorem (5):

β -ww-commutation: By lemma (4) below.

β -sw-commutation: Take $e = \bullet$, $e_0 = \bullet$, and $e_1 = y$. Then the l.h.s reduces thus:

$$(e[x := e_1]_a)[[e_0]]_b = (\bullet[x := y]_s)[[x]]_{\mathbf{w}} = (\bullet^{\{x:=y\}})[[x]]_{\mathbf{w}} = y,$$

while the r.h.s reduces as follows:

$$\begin{aligned} (e[x := \nu]_{\mathbf{w}})[[e_0]]_b[\nu := e_1[[e_0]]_b]_a &= (\bullet[x := \nu]_{\mathbf{w}})[[x]]_{\mathbf{w}}[\nu := y[[x]]_{\mathbf{w}}]_s \\ &= (\bullet)[[x]]_{\mathbf{w}}[\nu := y]_s = x. \end{aligned}$$

Clearly x can be chosen distinct from y .

β -ss-commutation: Take $e = \bullet$, $e_0 = x$, and $e_1 = y$. Then the l.h.s reduces thus:

$$(e[x := e_1]_a)[[e_0]]_b = (\bullet[x := y]_s)[[\bullet]]_s = \bullet^{\{x:=y\}}[[\bullet]]_s = (\bullet[x := y]_s) = \bullet^{\{x:=y\}}.$$

On the other hand the r.h.s. reduces as follows:

$$\begin{aligned} (e[x := \nu]_{\mathbf{w}})[[e_0]]_b[\nu := e_1[[e_0]]_b]_a &= (\bullet[x := \nu]_{\mathbf{w}})[[\bullet]]_s[\nu := y[[\bullet]]_s]_s \\ &= \bullet^{\{x:=\nu\}}[\nu := y]_s = \bullet^{\{x:=y, \nu:=y\}} \end{aligned}$$

β -ws-commutation: By lemma (3) above.

□

Lemma (4): If ν is fresh, then

$$(e[x := e_1]_{\mathbf{w}})[[e_0]]_{\mathbf{w}} \stackrel{\alpha}{=} (e[x := \nu]_{\mathbf{w}})[[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}}$$

Proof of Lemma (4): The proof is a simple induction on $\rho(e)$, and as usual only the case of the holes warrants inclusion.

(Induction Step subcase 2. $e = \bullet^{\sigma_0}$) By computation:

$$\begin{aligned} (e[x := e_1]_{\mathbf{w}})[[e_0]]_{\mathbf{w}} &= (\bullet^{\sigma_0}[x := e_1]_{\mathbf{w}})[[e_0]]_{\mathbf{w}} \\ &= \bullet^{\sigma_0[x:=e_1]}_{\mathbf{w}}[[e_0]]_{\mathbf{w}} \\ &= [e_0 := \sigma_0[x := e_1]_{\mathbf{w}}][[e_0]]_{\mathbf{w}} \end{aligned}$$

While

$$\begin{aligned} (e[x := \nu]_{\mathbf{w}})[[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}} &= (\bullet^{\sigma_0}[x := \nu]_{\mathbf{w}})[[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}} \\ &= \bullet^{\sigma_0[x:=\nu]}_{\mathbf{w}}[[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}} \\ &= e_0[\sigma_0[x := \nu]_{\mathbf{w}}][[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}} \\ &= e_0[\sigma_0[x := \nu]_{\mathbf{w}}][[e_0]]_{\mathbf{w}}[\nu := e_1[[e_0]]_{\mathbf{w}}]_{\mathbf{w}} \quad \text{by proposition } ([.]_{\mathbf{w}}.2) \text{ of §2.2} \end{aligned}$$

□

As a result of these observations we make the following definition.

Definition (\mapsto^β):

$$(\beta) \quad \mathbf{app}(\lambda x.e_0, e_1) \mapsto^\beta e_0[x := e_1]_{\mathfrak{w}}$$

We define unrestricted multiple step reduction \mapsto^* to be the smallest reflexive transitive relation generated by:

$$(\beta\kappa) \quad e[\mathbf{app}(\lambda x.e_0, e_1)]_{\mathfrak{s}} \mapsto^* e[e_0[x := e_1]_{\mathfrak{w}}]_{\mathfrak{s}}$$

Many (but not all, e.g. the λ -I calculus) variations of the λ -calculus can be obtained by restricting, in various ways, the $\beta\kappa$ reductions and perhaps adding new forms of reductions. We give several examples to illustrate this idea. In what follows we will use the operation $[\cdot]_{\mathfrak{s}}$ to denote either the strong or weak version of hole filling, since by the previous results either has the properties we require.

3.2. Call-by-Value λ -calculus

To describe the call-by-value λ -calculus we must first define the set of value expressions, \mathbb{V} , or more simply the set of values: $\mathbb{V} = \mathbb{A} \cup \mathbb{X} \cup \mathbb{L}$. The call-by-value β -reduction is defined by:

$$(\beta_v) \quad \mathbf{app}(\lambda x.e_0, e_1) \mapsto^\beta e_0[x := e_1]_{\mathfrak{w}} \quad \text{provided } e_1 \in \mathbb{V}$$

We define unrestricted multiple step reduction \mapsto^* to be the smallest reflexive transitive relation generated by:

$$(\beta_v\kappa) \quad e[\mathbf{app}(\lambda x.e_0, e_1)] \mapsto^* e[e_0[x := e_1]_{\mathfrak{w}}] \quad \text{provided } e_1 \in \mathbb{V}$$

3.3. Left-first call-by-Value λ -calculus

To describe the left first call-by-value λ -calculus (ISWIM) we first define reduction contexts (a.k.a evaluation contexts) and use them to restrict the order of evaluation. Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of [22] and were first introduced in [8]. The set of reduction contexts, \mathbb{R} , is the subset of \mathbb{E} defined by

$$\mathbb{R} = \{\bullet\} + \mathbf{app}(\mathbb{R}, \mathbb{E}) + \mathbf{app}(\mathbb{V}, \mathbb{R})$$

We define left-first multiple step reduction to be the smallest reflexive transitive relation generated by:

$$(\beta_v\kappa.\text{lf}) \quad e[\mathbf{app}(\lambda x.e_0, e_1)] \mapsto^* e[e_0[x := e_1]_{\mathfrak{w}}] \quad \text{provided } e_1 \in \mathbb{V} \text{ and } e \in \mathbb{R}$$

3.4. λ_{cc} -calculus

Following [10] we present the λ_{cc} -calculus as the left-first call-by-value λ -calculus enriched with two control primitives **A** and **C** in \mathbb{A} . The reduction relation is the least transitive, reflexive relation generated by $(\beta_v \kappa. \mathbf{If})$ extended by the two delta rules $\delta. \mathbf{A}$ and $\delta. \mathbf{C}$

$$(\beta_v \kappa. \mathbf{If}) \quad e[\mathbf{app}(\lambda x. e_0, e_1)] \xrightarrow{*} e[[e_0[x := e_1]_{\mathbb{W}}]] \quad \text{provided } e_1 \in \mathbb{V} \text{ and } e \in \mathbb{R}$$

$$(\delta. \mathbf{A}) \quad e[\mathbf{app}(\mathbf{A}, e_0)] \xrightarrow{\delta} e_0 \quad \text{provided } e \in \mathbb{R}$$

$$(\delta. \mathbf{C}) \quad e[\mathbf{app}(\mathbf{C}, e_0)] \xrightarrow{\delta} \mathbf{app}(e_0, \lambda z. \mathbf{app}(\mathbf{A}, e[[z]])) \quad \text{provided } e \in \mathbb{R} \text{ and } z \text{ is fresh}$$

3.5. $\lambda_{\mathbb{M}}$ -calculus

Following [12] we define the $\lambda_{\mathbb{M}}$ -calculus as left-first call-by-value λ -calculus enriched with the three memory primitives **M**, **G** and **S** $\in \mathbb{A}$ (called *make*, *get* and *set*) together an pairing operation **D**. The pairing operation separates the part of the expression that represents the memory, from the part that represents the current computation (see §2.2.1 of [12] for details). In what follows we will describe the actual terms as well as give more readable variations using the traditional **let** abbreviation for λ -application, function application $F(X)$ to represent $\mathbf{app}(F, X)$ (similarly $F(X_1, X_2)$ for binary application $\mathbf{app}(\mathbf{app}(F, X_1), X_2)$), **I** for the identity $\lambda y. y$, **infi** x ; for the sequencing construct, and following [27] $e_0 : e_1$ to abbreviate $\mathbf{app}(\mathbf{app}(\mathbf{D}, e_0), e_1)$. The set of memory contexts, \mathbb{M} , is defined inductively as the smallest subset of \mathbb{E} containing \bullet and closed under the following formation rules:

If $e \in \mathbb{M}$, and $x \in \mathbb{X}$ is fresh, then

$$\begin{aligned} & \mathbf{app}(\lambda x. (e[\bullet^{\{x:=x\}}]), \mathbf{app}(\mathbf{M}, \lambda y. y)) \in \mathbb{M} \\ \text{i.e.} \quad & \mathbf{let}\{x := \mathbf{M}(\mathbf{I})\}(e[\bullet^{\{x:=x\}}]) \in \mathbb{M} \end{aligned}$$

If $e \in \mathbb{M}$, $v \in \mathbb{V}$, and $x \in \text{TV}(e)$, then

$$\begin{aligned} & e[\mathbf{app}(\lambda z. \bullet, \mathbf{app}(\mathbf{app}(\mathbf{S}, x), v))] \in \mathbb{M} \\ \text{i.e.} \quad & e[\mathbf{S}(x, v); \bullet] \in \mathbb{M} \end{aligned}$$

Note that no hygiene conditions are needed concerning z . If $e \in \mathbb{M}$, then either e is a **let**-context (i.e. contains no occurrences of **S**) or upto α -congruence there exists unique $e_0, e_1 \in \mathbb{E}$ and $\sigma \in \mathbb{S}$ such that $e_1 \in \mathbb{M}$ has a unique hole \bullet^σ , σ is a bijection and its range is in \mathbb{X} , $e \stackrel{\alpha}{\equiv} e_1[[e_0; \bullet]]$ and $e_0 \stackrel{\alpha}{\equiv} \mathbf{S}(x, v)$ with $x \in \text{Dom}(\sigma)$. Using this decomposition we can define a simple rewrite system that looks up the value of z in a memory context.

Definition ($\overset{z}{\hookrightarrow}$): Assume that $e \in \mathbb{M}$ decomposes into $e_1[[\mathbf{S}(x, v); \bullet]]$. Then we define

$$e_1[[\mathbf{S}(x, v); \bullet]] \overset{z}{\hookrightarrow} \begin{cases} v & \text{if } x = z \\ e_1 & \text{otherwise} \end{cases}$$

Similarly we write $\xrightarrow{z^*}$ for the transitive closure of this relation. It is straight forward to check that if $e \xrightarrow{z} v$, then v is unique upto α -congruence. We are now in a position to define the reduction rules for the $\lambda_{\mathbb{M}}$ -calculus.

$$\begin{aligned}
(\mathbb{M}.\beta_v \kappa.\text{lf}) \quad & \text{app}(\text{app}(\mathbb{D}, e_2), e[\text{app}(\lambda x.e_0, e_1)]) \xrightarrow{*} \text{app}(\text{app}(\mathbb{D}, e_2), e[e_0[x := e_1]_{\mathbb{W}}]) \\
& \text{provided } e_2 \in \mathbb{M}, e_1 \in \mathbb{V} \text{ and } e \in \mathbb{R} \\
& \text{i.e. } e_2 : e[\text{app}(\lambda x.e_0, e_1)] \xrightarrow{*} e_2 : e[e_0[x := e_1]_{\mathbb{W}}] \\
(\mathbb{M}.\delta.\mathbb{M}) \quad & \text{app}(\text{app}(\mathbb{D}, e_2), e[\text{app}(\mathbb{M}, e_1)]) \xrightarrow{\delta} \\
& \text{app}(\text{app}(\mathbb{D}, \text{app}(\lambda x.(e_2[\text{app}(\lambda z.\bullet^{\{x:=x\}}, \text{app}(\text{app}(\mathbb{S}, x), e_1)])), \\
& \quad \text{app}(\mathbb{M}, \lambda y.y))), e[x]) \\
& \text{provided } e_2 \in \mathbb{M}, e_1 \in \mathbb{V}, e \in \mathbb{R} \text{ and } x \in \mathbb{X} \text{ is fresh} \\
& \text{i.e. } e_2 : e[\text{app}(\mathbb{M}, e_1)] \xrightarrow{\delta} \text{let}\{x := \mathbb{M}(\mathbb{I})\}(e_2[\mathbb{S}(x, e_1); \bullet^{\{x:=x\}}]) : e[x] \\
(\mathbb{M}.\delta.\mathbb{G}) \quad & \text{app}(\text{app}(\mathbb{D}, e_2), e[\text{app}(\mathbb{G}, e_1)]) \xrightarrow{\delta} \text{app}(\text{app}(\mathbb{D}, e_2), e[e_3]) \\
& \text{provided } e_2 \in \mathbb{M}, e_1 \in \mathbb{X} \cap \text{TV}(e_2), e \in \mathbb{R} \text{ and } e_2 \xrightarrow{e_1^*} e_3 \in \mathbb{V}. \\
& \text{i.e. } e_2 : e[\mathbb{G}(e_1)] \xrightarrow{\delta} e_2 : e[e_3] \\
(\mathbb{M}.\delta.\mathbb{S}) \quad & \text{app}(\text{app}(\mathbb{D}, e_2), e[\text{app}(\lambda z.\bullet, \text{app}(\text{app}(\mathbb{S}, x), e_1)])) \xrightarrow{\delta} \\
& \text{app}(\text{app}(\mathbb{D}, e_2[\text{app}(\text{app}(\mathbb{S}, x), e_1)]), e[\lambda y.y]) \\
& \text{provided } e_2 \in \mathbb{M}, e_1 \in \mathbb{V}, e \in \mathbb{R} \text{ and } x \in \text{TV}(e_2) \\
& \text{i.e. } e_2 : e[\mathbb{S}(x, e_1)] \xrightarrow{\delta} e_2[\mathbb{S}(x, e_1); \bullet] : e[\mathbb{I}]
\end{aligned}$$

Observe that

$$(\text{let}\{x := \mathbb{M}(\mathbb{I})\}(e[\bullet^{\{x:=x\}}]))[\mathbb{S}(x, v); \bullet] \stackrel{\alpha}{\equiv} \text{let}\{x := \mathbb{M}(\mathbb{I})\}(e[\mathbb{S}(x, v); \bullet^{\{x:=x\}}])$$

and so in the δ rule $(\mathbb{M}.\delta.\mathbb{M})$ the resulting *memory component* is indeed an element of \mathbb{M} . Thus the reduction system so defined, contains only one complex side condition. Other than the δ rule $(\mathbb{M}.\delta.\mathbb{G})$ all reductions rely on simple structural conditions. The side conditions on $(\mathbb{M}.\delta.\mathbb{G})$ are expressed via a simple reduction system.

4. Conclusions & Future Work

In this paper we have presented a *named variable* version of Talcott's binding structure theory [25, 26] and established several important results concerning it. In the general framework presented here notions of substitution and hole filling have two obvious versions: *weak* and *strong*. We demonstrated that it is the weak form of substitution that corresponds most appropriately to β -reduction. However when it came to hole filling the weak and strong versions seemed equally suited to their job. Using this theory we have

also shown that computing with contexts, via the various forms of reduction presented in §3, is not problematic.

We have also presented reduction systems for the lambda calculus enriched with control and imperative features that should be easily encoded into modern logical frameworks. Thus enabling the syntax and semantics of logics such as VTL_{oE} to be encoded [12]. This would allow for both proof theoretical and semantic reasoning to be carried out at the same time in the same framework. It will also allow the system to semantically verify the soundness of its own proof system. It would also allow for the dynamic enrichment (via a sort of *meta-rule*) of the proof theory by introducing new, semantically verified, principles. Thus the logic implemented would be truly dynamic. The actual choice of logical framework & the subsequent encoding is the subject of future work.

The underlying aim behind this work is enable a clean elegant encoding of various operational semantics and programming logics into the current generation of logical frameworks. In our previous work [9] we have described an encoding of the syntax and proof theory of a modern programming logic [16] into a generic theorem prover. Encoding the syntax and proof theory of the logic was a relatively painless procedure. Especially when compared with the contortions required for logics of the Hoare and Dynamic ilk [15, 4]. Since the semantics of the underlying programming language (λ_{H} -calculus) is operational, and the semantics of the logic is defined strictly in terms of syntactic entities, it seems not unreasonable to expect an implementation to be capable of encoding the semantics as well as the syntax and proof theory. This would allow for both proof theoretical and semantic reasoning to be carried out at the same time in the same context [28]. It will also allow the system to semantically verify its own proof system, an attractive idea. It would also allow for the dynamic enrichment of the proof theory by introducing new, semantically verified, principles. Thus the logic implemented would be truly extensible or dynamic. The only obstacle to successfully encoding the semantics is the problem of encoding lambda calculus style contexts and hole filling (i.e the corresponding notion of substitution with variable binding capture).

It is of obvious interest to compare our work with the other approaches mentioned in §1. Time (& space) considerations prevent us from doing this in the present version of this paper.

Acknowledgments

This work was done while the author was partially supported by ARC grant M0008202 (University of Tasmania), ARC grant IA131.84 and URG grant IB34.1 (University of New England). The author would also like to thank Carolyn Talcott, Laurent Dami, and the three anonymous referees for helpful comments.

References

1. Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.
2. Peter Aczel. A general Church-Rosser theorem, 1978. Unpublished manuscript.
3. Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

4. Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. *Journal of Automated Reasoning*, 9:309–354, 1992.
5. Haskell B. Curry and Robert Feys. *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1958.
6. Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192, 1998.
7. Matthias Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.
8. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ -calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
9. Jacob Frost and Ian A. Mason. An Operational Logic of Effects. In *Proceedings of the Australasian Theory Symposium, CATS '96*, pages 147–156, 1996.
10. Timothy G. Griffin. A formulae as types notion of control. In *Seventeenth Annual ACM Symposium on Principles of Programmings Languages*, pages 47–58, 1990.
11. Masatomo Hashimoto and Atsushi Ohori. A typed context calculus., 1996. Preprint RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, 1996. See: <http://www.kurims.kyoto-u.ac.jp/~ohori/>.
12. Furio Honsell, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, May 1995.
13. Jan. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Published as a Mathematical Centre Tract 129.
14. Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: Towards a theory of incremental program construction. In *Proceedings ACM Int. Conf. on Functional Programming, ACM SIGPLAN Notices*, 31(6):239-250, 1996.
15. Ian A. Mason. Hoare's Logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for foundations of computer science, University of Edinburgh, 1987.
16. Ian A. Mason. A First Order Logic of Effects. *Theoretical Computer Science*, 185:277 – 318, 1997.
17. Ian A. Mason. Parametric Computation. In James Harland, editor, *Proceedings of the Australasian Theory Symposium, CATS '97*, pages 103 – 112, 1997. Complete proofs of all claims are also available from <http://smcs.une.edu.au/~iam/Data/Papers/97cats-proofs.ps> as postscript.
18. Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
19. Robin Milner. Fully abstract models of typed λ -calculi. *Theoretical Computer Science*, 4:1–22, 1977.
20. James H. Morris. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
21. Andy M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs, 1994. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus.
22. Gordon Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
23. David Sands. Computing with contexts – a simple approach. In *Proceedings of Higher-Order Operational Techniques in Semantics, HOOTS II*, volume 10 of *Electronic Notes in Theoretical Computer Science*, 1998.
24. Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. In J.-Y Girard, editor, *Typed Lambda Calculi and Applications – 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, 1999.
25. Carolyn L. Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artifi cial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.
26. Carolyn L. Talcott. A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.
27. Carolyn L. Talcott. Reasoning about functions with effects. In Andrew Gordon and Andy Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998.
28. Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artifi cial Intelligence*, 13:133–170, 1980.