# Semantics of the *reFL^ect* Language*

Sava Krstić
Strategic CAD Labs, Intel Corporation
sava.krstic@intel.com

John Matthews
OGI School of Science & Engineering at OHSU
johnm@cse.ogi.edu

## ABSTRACT

*reFL^ect* is a new functional language, developed at Intel for use in hardware design and verification. It contains features intended to facilitate the construction, analysis, and manipulation of the language's own programs. It is also intended to be the executable subset of the term language of a theorem prover based on higher order logic.

In this paper, we consider core *reFL^ect*—a language that extends a polymorphically typed $\lambda$-calculus with a datatype for programs and with constructs for splicing programs into programs and for defining functions that inspect and modify programs. We prove that the reduction semantics for this language is strongly normalizing and confluent. We also give a set-theoretical denotational semantics for the language and prove that the reduction semantics is sound with respect to the denotational semantics. These results provide the basis for developing the semantics of *reFL^ect*'s extension of higher order logic and proving its soundness.

## Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Denotational semantics, Operational semantics; D.3.3 [**Language Constructs and Features**]: Patterns

## General Terms

Languages

## 1. INTRODUCTION

This paper presents several basic technical results concerning the operational and denotational semantics of the reflective functional programming language *reFL^ect*, currently being developed at Intel [8]. *reFL^ect* is a higher order, statically typed, lazy functional programming language. It is both the specification and implementation language for Intel's next-generation hardware formal verification environment.

*reFL^ect* is distinguished from widely-known higher order functional languages such as Haskell and ML in its support for *reflection*, that is, constructing and pattern-matching on *reFL^ect* syntactic expressions themselves as first-class data values. Reflection allows circuit specifications, circuit transformations, and formal verification algorithms to be specified uniformly and declaratively.

*reFL^ect* is being used to implement an integrated theorem prover and model checker based on classical higher order logic. The theorem prover is mainly used to manually decompose large circuit and protocol verification problems into a series of smaller problems, each of which is then automatically verified using existing model checking algorithms (which are also written in *reFL^ect*).

The term language of this verification tool is *reFL^ect* itself. Thus it is essential that the language semantics is logically consistent when the reduction rules are considered as inference rules of classical higher order logic. In particular, the denotational semantics of *reFL^ect* functions is set-theoretic, not domain-theoretic. This distinguishes *reFL^ect* from existing higher order metaprogramming languages such as Scheme [1], MetaML [15], and Template Haskell [14].

Intel expects practicing hardware designers to be using and even writing generic circuit transformations. This desideratum lead *reFL^ect* developers to devise an interesting, though somewhat weak type system. For example, it is possible in *reFL^ect* to concisely write a term-rewriting algorithm that takes as input a list of rewrite rules, the left-hand sides of which are expressions of different types. In more strongly typed metaprogramming languages like MetaML, such heterogeneous lists may not be typable. On the other hand, some form of dependent typing might be able to accept generic code transformation algorithms like the mentioned rewriting example. However, such systems require substantial expertise in type theory to use effectively. The price *reFL^ect* pays for using a simpler type system is that some type errors in reflective code will only be caught at runtime. This same tradeoff is made by Template Haskell, and seems to work well in practice.

Some of *reFL^ect*'s design decisions have been motivated by the notable success of the ACL2 theorem prover [11] in verifying realistic hardware and software specifications. ACL2 uses a first order applicative subset of Common Lisp as its logical and programming language semantics. This allows ACL2 specifications to be efficiently validated by executing

*The research reported in this paper was supported by a grant from Intel Corporation and was carried out while Sava Krstić was affiliated with OGI School of Science and Engineering.

them on concrete test cases. ACL2 also has constructs for first class metaprogramming and logical reflection. A key goal of *reFLect* is to combine the expressiveness and strong typing of classical higher order logic with the efficient execution and reflection abilities of ACL2.

The original *reFLect* paper [8] defines the operational semantics of the language, but does not establish its desirable fundamental properties. This is accomplished in the current paper; it solidifies the theoretical basis of *reFLect* and opens prospects for deeper investigations of the language and its logic. We prove that the operational semantics of *reFLect* satisfies subject reduction and that it is confluent and strongly normalizing. (Cf. [16, 5] for proofs of these properties for two other languages that generalize the lambda calculus by allowing more general pattern-matching constructs.) We also prove that *reFLect* is sound with respect to a simple set-theoretic denotational semantics.

The language considered in this paper is core *reFLect*, as described in Sections 1–7 of [8]. The full language has three additional constructs that facilitate reflection ([8], Section 8), but these are left out for future research.

We use standard proof techniques, with appropriate modifications to account for the reflective nature of the language. For example, even though the language definition begins with a simple context-free description of raw expressions, the actual well-formed *reFLect* expressions exhibit a distinctive recursive structure that is more subtle. Proofs by structural induction refer to this structure rather than the one induced by the grammar.

Basic notions like free variable occurrences, $\alpha$-equivalence, and substitution need to be overhauled too because they fundamentally bear upon *reFLect*'s own concepts of patterns, quotations and antiquotations. For example, it takes some effort to verify the simple but vital fact that substitution is a well-defined operation on $\alpha$-equivalence classes.

This paper is self-contained. However, the reader is strongly encouraged to consult [8], where motivation for various design decisions can be found, as well as more extensive examples. We have strived to produce complete proofs of the main theorems and fully develop all necessary concepts and auxiliary results. To stay within page limits of a conference paper, we relegate proofs of many of these results to the Appendix that can be found on the authors' web pages. Missing proofs can also be found in our technical report [12] The following paragraphs describe briefly the contents of individual sections of this paper.

In Section 2, slightly varying the original definition, we introduce well-formed *reFLect* expressions and several pertinent notions. Lemma 1 is the basis for proofs by structural induction. Also important is Lemma 4, showing preservation of well-formedness by replacement.

Section 3 gives a rather complete account of $\alpha$-equivalence and substitution, concluding with the definition of pattern matching.

In Section 4, we define the *reFLect* reduction system, and also a simpler but larger auxiliary non-deterministic system. It is not obvious that the reduction is well-defined as a relation between $\alpha$-equivalence classes, but once that has been checked, the Subject Reduction Theorem comes almost for free.

Confluence is established in Section 5, as a consequence of local confluence and strong normalization. Strong normalization of *reFLect* is a non-trivial generalization of the same result for typed $\lambda$-calculus. We prove it by partitioning the auxiliary reduction system mentioned above into two subsystems that quasi-commute and are strongly normalizing themselves.

Finally, in Section 6, we give a naive set-theoretic denotational semantics for *reFLect* expressions and prove soundness of *reFLect* reduction with respect to this semantics.

## 2. SYNTAX

*reFLect* is an ML-like language, with a built-in type term for quoted expressions and several constructs to facilitate constructing and analyzing quoted expressions. For logical soundness reasons explained in [8], the language restricts the ML-style polymorphism by disallowing the "polymorphic *let*". Without its main features, *reFLect* boils down to the polymorhically typed $\lambda$-calculus $\lambda\!\rightarrow$ as described in [10].

The language supports an arbitrary polymorphic signature (consisting of additional type operators and constant expressions), but for the reasons of simplicity and without essential loss of generality, we will restrict ourselves to the case when the signature is empty.

### 2.1 Expressions

The set of *reFLect* types is freely generated by a set of type variables and a set of type operators in the usual manner. In the minimal version of the language that will be considered in this paper, we require only the binary function space operator $\Rightarrow$ and a primitive type (nullary operator) term, as shown in the simple grammar in the first line of Figure 1.

$$\begin{array}{rcl} \sigma & ::= & \alpha \;\mid\; \mathsf{term} \mid \sigma \Rightarrow \sigma \\ \\ M & ::= & v^{\sigma} \;\mid\; M \cdot M \;\mid\; \lambda M.\, M \\ & & \mid\; M + M \;\mid\; \langle\!\langle M \rangle\!\rangle \;\mid\; {}^{\wedge}M^{\sigma} \end{array}$$

**Figure 1: Syntax for types and expressions.**

Figure 1 also shows a grammar for raw expressions. We will use the terminology VARIABLE, APPLICATION, LAMBDA ABSTRACTION, ALTERNATION, QUOTATION, ANTIQUOTATION to classify the expressions according to the top symbol in them. Note that each variable and each antiquotation are explicitly typed[1] and that $\lambda$-abstractions can bind arbitrary pattern expressions, not just variables.

Not all expressions generated by the grammar are meaningful. The quotation and antiqoutation symbols serve to generate expressions of type term and their use is restricted (governed by the typing rules given in Section 2.3) so that, for example, no top-level expression of the form ${}^{\wedge}M^{\sigma}$ is legitimate. The basic idea is that an expression of the form $\langle\!\langle M \rangle\!\rangle$ is legitimate if $M$ can be obtained by replacing in some expression $M'$ several (possibly zero) subexpressions $M'_1, \ldots, M'_k$ of respective types $\sigma_1, \ldots, \sigma_k$ with ${}^{\wedge}M_1{}^{\sigma_1}$, $\ldots$, ${}^{\wedge}M_k{}^{\sigma_k}$, where each $M_i$ is of type term.[2] For example,

[1]We will be careful with superscripts and use them only for this purpose of annotating variables and antiquotations with their types.

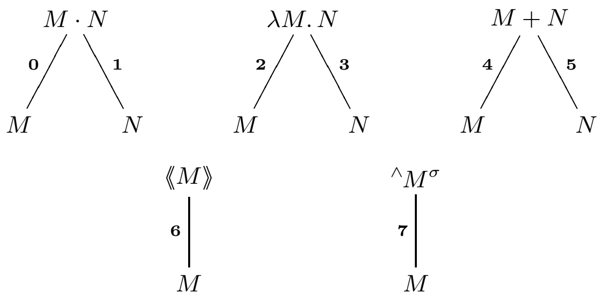
[2]Since it is irrelevant what the subexpressions $M'_i$ of $M'$ exactly are, the actual rule for constructing quotations will

$\langle\!\langle x^{\mathsf{nat}} + {}^\wedge\langle\!\langle 15\rangle\!\rangle^{\mathsf{nat}}\rangle\!\rangle$ and $\langle\!\langle x^{\mathsf{nat}} + {}^\wedge(y^{\mathsf{term}})^{\mathsf{nat}}\rangle\!\rangle$ are legitimate.[3]

There is also a restriction on the form of patterns in abstractions $\lambda L.\, M$. They can be either variables or quotations of a specific form, to be explained below. Quotation patterns make it possible to write functions that inspect and transform code. Two simple examples are $\lambda\langle\!\langle {}^\wedge(f^{\mathsf{term}})^{\alpha\Rightarrow\beta} \cdot {}^\wedge(x^{\mathsf{term}})^{\alpha}\rangle\!\rangle.\, x^{\mathsf{term}}$ and $\lambda\langle\!\langle {}^\wedge(x^{\mathsf{term}})^{\mathsf{nat}} + {}^\wedge(x^{\mathsf{term}})^{\mathsf{nat}}\rangle\!\rangle.\, \langle\!\langle 2 * {}^\wedge(x^{\mathsf{term}})^{\mathsf{nat}}\rangle\!\rangle$; the first extracts the argument of a function application, while the second can be used to optimize arithmetical code.

Since quotation patterns may fail to match, *reFLect* provides an alternation construct with the intended semantics that the evaluation of $(\lambda L.\, M + M')\cdot N$ proceeds as the evaluation of $(\lambda L.\, M)\cdot N$ if $L$ matches $N$, and as $M'\cdot N$ otherwise. Pattern matching rules are explained in Section 3.3.

For reasoning about *reFLect* expressions, it is convenient to view them as ordered trees with internal nodes labeled by symbols $\cdot$, $\lambda$, $+$, $\langle\!\langle\rangle\!\rangle$, ${}^\wedge()^\sigma$, and with leaves labeled by variables. The edges are labeled by numbers between $\mathbf{0}$ and $\mathbf{7}$, according to the symbol at the edges' source nodes, as follows:



Every node in the tree has its unique POSITION—the string over $\{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{7}\}$ describing the sequence of labels from the root to that node. The empty string $\varepsilon$ is the root position. Every expression $M$ thus has an associated prefix-closed set of positions $\mathsf{Pos}(M)$. We will write $\pi \leq \pi'$ to indicate that $\pi$ is a prefix of $\pi'$ and $\pi \wedge \pi'$ for the largest common prefix of $\pi$ and $\pi'$.

The subexpression of $M$ at position $\pi$ will be denoted $M|_\pi$. The result of replacing in $M$ this subexpression with another expression $N$ will be denoted $M[\pi \mapsto N]$.

The subexpression $M|_\pi$ is a variable if and only if $\pi$ is a maximal element of $\mathsf{Pos}(M)$ with respect to the prefix ordering. The set of variable positions will be denoted $\mathsf{VarPos}(M)$. Note that, due to the precise labeling of edges, $M$ is completely described by the set $\mathsf{Pos}(M)$ and by knowing $M|_\pi$ for all $\pi \in \mathsf{VarPos}(M)$.

CONTEXTS are defined using the same grammar for expressions as in Figure 1 extended with the hole production $M ::= \llcorner\!\lrcorner$. The definition of replacement trivially extends to contexts. Most of the time, our contexts will be denoted $\mathcal{C}$ and when we write $M = \mathcal{C}[M_1, \ldots, M_k]$, it will be implicitly assumed that $\mathcal{C}$ has $k$ holes at certain positions and that $M$ is the expression obtained by filling these holes in $\mathcal{C}$ with expressions $M_1, \ldots, M_k$.

use contexts in place of the expression $M'$ here.

[3]In small illustrative examples, we use expressions that mention the type $\mathsf{nat}$ and arithmetical operations, even though they do not figure in our syntax. See [8].

## 2.2 Level Consistency

We mentioned briefly that in a legitimate *reFLect* expression every occurring antiquote is introduced together with a corresponding enclosing quote. As a consequence, every subexpression is surrounded by no fewer quotes than antiquotes. We begin the formal treatment of defining well-formed expressions by making this fundamental property precise.

The LEVEL of a subexpression $M|_\pi$ in an expression $M$ is the number of the enclosing quotations minus the number of enclosing antiquotations. The level thus depends only on the position $\pi$ and is equal to

$$\#\{\text{occurrences of } \mathbf{6} \text{ in } \pi\} - \#\{\text{occurrences of } \mathbf{7} \text{ in } \pi\}.$$

LEVEL CONSISTENT EXPRESSIONS are those in which every position has a non-negative level. More generally, a LEVEL CONSISTENT CONTEXT is one in which all positions have non-negative level and all hole positions have level zero.

Level consistent expressions and contexts exhibit a block structure determined by the partition of the set of positions into those of level 0 (ACTIVE), and those of level greater than 0 (PASSIVE). This structure is depicted in Figure 2. Informally, starting at the root and walking down a level consistent expression's tree, one goes through active nodes for some time (the top block ① in the picture), until encountering the first quote node. This node is the entry point to the first layer of passive positions, grouped into blocks ②, ③, and ④ in the picture. One can pass through any number of quote or antiquote nodes while staying in a passive block, as long as the total number of visited quote nodes remains greater than the total number of visited antiquote nodes. Getting into an antiquote node that balances the two totals is the entry point into the new layer of active nodes. In the figure, that new layer cannot be reached from block ②, while from block ③ there are two antiquote "doors" to active blocks ⑤ and ⑥.

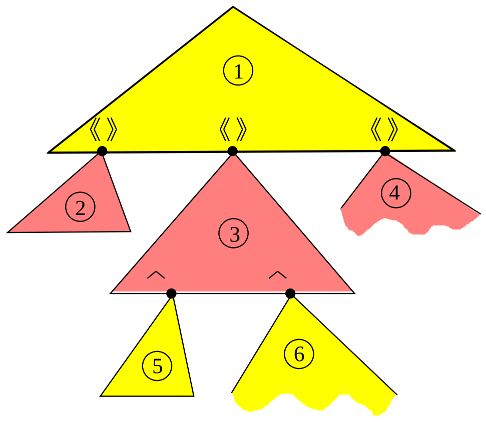

1
$\langle\!\langle\rangle\!\rangle$
$\langle\!\langle\rangle\!\rangle$
$\langle\!\langle\rangle\!\rangle$
2
3
4
$\wedge$
$\wedge$
5
6

**Figure 2: Structure of level consistent expressions.**

A convenient formalization of the inductive structure of the set of level consistent expressions is given by the following lemma, whose proof is straighforward and threrefore omitted. It is the basis for recursive definitions (of functions whose argument is a *reFLect* expression) and proofs by structural induction.

Lemma 1 (Structure). *If $M, N, M_1, \ldots, M_k$ are level consistent expressions and $\mathcal{C}$ is a level consistent context, then the expressions*

$$v^\sigma \quad M \cdot N \quad \lambda M.\, N \quad M + N \quad \langle\!\langle \mathcal{C}[{}^\wedge {M_1}^{\sigma_1}, \ldots, {}^\wedge {M_k}^{\sigma_k}] \rangle\!\rangle$$

*are level consistent. Moreover, every level consistent expression can be uniquely written in one of these five forms.* □

Note that the uniqueness part of the lemma says in particular that for every level consistent quotation $M$ there exist a unique level consistent context $\mathcal{C}$ (with, say, $k$ holes), unique types $\sigma_1, \ldots, \sigma_k$, and unique level consistent expressions $M_1, \ldots, M_k$ such that $M = \langle\!\langle \mathcal{C}[{}^\wedge {M_1}^{\sigma_1}, \ldots, {}^\wedge {M_k}^{\sigma_k}] \rangle\!\rangle$. The expressions $M_1, \ldots, M_k$ will be referred to as factors of $M$.

In the sequel we will also use the term active subexpression to mean a subexpression occurring at an active position.

## 2.3 Typing

The set of well-typed expressions is defined by rules in Figure 3. Note that the last rule has a meaning even when $k = 0$. It says in this important special case that we can derive $\langle\!\langle M \rangle\!\rangle \colon \mathsf{term}$ whenever we have $M \colon \sigma$, for any $\sigma$. Such expressions (namely, well-typed quotations without factors) will be called pure quotations.

$$\frac{}{v^\sigma : \sigma} \qquad \frac{M \colon \sigma \Rightarrow \tau \qquad N \colon \sigma}{M \cdot N \colon \tau}$$

$$\frac{L \colon \sigma \qquad M \colon \tau}{\lambda L.\, M \colon \sigma \Rightarrow \tau} \qquad \frac{M \colon \sigma \qquad N \colon \sigma}{M + N \colon \sigma}$$

$$\frac{\begin{array}{c} M_i \colon \mathsf{term} \qquad \mathcal{C}[z_1^{\sigma_1}, \ldots, z_k^{\sigma_k}] \colon \sigma \\ (1 \le i \le k,\ \mathcal{C} \text{ level consistent},\ z_i \text{ fresh}) \end{array}}{\langle\!\langle \mathcal{C}[{}^\wedge {M_1}^{\sigma_1}, \ldots, {}^\wedge {M_k}^{\sigma_k}] \rangle\!\rangle \colon \mathsf{term}}$$

**Figure 3: Typing rules.**

The typing rules correspond to the five cases in Lemma 1 (Structure). The following lemma is easily proved by structural induction.

Lemma 2 (Unique Typing). *Every well-typed expression $M$ is level consistent and there is a unique type $\sigma$ such that $M \colon \sigma$.* □

Type inference in *reFLect* is briefly discussed in [8]. The reader may find it useful to work out the typing of the expression $(\lambda v.\, \langle\!\langle \lambda {}^\wedge v.\, {}^\wedge v + 1 \rangle\!\rangle) \cdot \langle\!\langle w \rangle\!\rangle$. The exercise is to show that a type derivation for $(\lambda v^\sigma.\, \langle\!\langle \lambda {}^\wedge (v^\sigma)^\tau.\, {}^\wedge (v^\sigma)^\tau + 1 \rangle\!\rangle) \cdot \langle\!\langle w^\omega \rangle\!\rangle$ exists if and only if $\sigma = \mathsf{term}$ and $\tau = \mathsf{nat}$ (while $\omega$ can be arbitrary).

## 2.4 Patterns and Well-formedness

By definition, a pattern is either a variable or a well-typed quotation all of whose factors are variables. Any quotation pattern thus has the form $\langle\!\langle \mathcal{C}[{}^\wedge (v_1^{\mathsf{term}})^{\sigma_1}, \ldots, {}^\wedge (v_k^{\mathsf{term}})^{\sigma_k}] \rangle\!\rangle$, where $\mathcal{C}$ is a level consistent context and the $v_i^{\mathsf{term}}$ are not necessarily distinct variables. Patterns can be characterized as well-typed expressions whose only active subexpressions are variables. Thus, for example, all pure quotations are patterns. The variables occurring as factors in quotation patterns are not required to be distinct; an example is the pattern $\langle\!\langle \lambda {}^\wedge v.\, {}^\wedge v + 1 \rangle\!\rangle$.[4]

If $L$ is a pattern, we will denote by $\mathsf{PatVar}(L)$ the set of variables occurring at active positions in $L$. Thus, $\mathsf{PatVar}(v^\sigma) = \{v^\sigma\}$, and $\mathsf{PatVar}(L) = \{v_1^{\mathsf{term}}, \ldots, v_k^{\mathsf{term}}\}$ when $L$ is a quotation pattern as above.

By definition, an expression is well-formed if it is well-typed and satisfies the following pattern condition: in every active subexpression of the form $\lambda L.\, M$, the expression $L$ is a pattern. Note that patterns themselves are all well-formed expressions; they satisfy the pattern condition vacuously.

In the sequel, by a pattern position in a well-formed expression we will understand an active position that ends in $\mathbf{2}$, and a binding position will mean an active position that contains a pattern position as a prefix. If $\mu\mathbf{2}$ is a pattern position, then clearly $E|_\mu$ is an active abstraction and $L = E|_{\mu\mathbf{2}}$ is a pattern. Binding positions that have $\mu\mathbf{2}$ as a prefix are of the form $\mu\mathbf{2}\pi$, where $\pi$ is an active position in $L$. If $L$ is a variable, the only possibility is $\pi = \varepsilon$ (the pattern position $\mu\mathbf{2}$ is at the same time a binding position); if $L = \mathcal{C}[{}^\wedge (v_1^{\mathsf{term}})^{\sigma_1}, \ldots, {}^\wedge (v_k^{\mathsf{term}})^{\sigma_k}]$, then $\pi = \pi'\mathbf{7}$, where $\pi'$ is one of the $k$ hole positions of $\mathcal{C}$.

The following lemmas state the basic properties of well-formed expressions that will be needed in subsequent sections. They are proved by structural induction and their proofs are given in [12]. The same results remain valid when "well-formed expression" is replaced everywhere with "well-typed expression".

Lemma 3. *All active subexpressions of a well-formed expression are well-formed.* □

Corollary 1. *The factors of well-formed quotations are well-formed.* □

Lemma 4 (Replacement). *Suppose $E$ is a well-formed expression of type $\sigma$ and $\pi$ is an active position in $E$, but not a pattern or a binding position. If $E|_\pi \colon \tau$ and $N$ is any well-formed expression of type $\tau$, then $E[\pi \mapsto N]$ is a well-formed expression of type $\sigma$.* □

## 2.5 Type Instantiation

A type instantiation is a map $\phi$ from type variables to types, with finite support set $\mathsf{supp}(\phi) = \{\alpha \mid \phi(\alpha) \neq \alpha\}$. Every type instantiation extends recursively to a map, also denoted by $\phi$, from types to types.

It is easy to prove, using the freeness of the algebra of types, that for any given types $\sigma_1, \ldots, \sigma_k$ and $\tau_1, \ldots, \tau_k$, there exists at most one type instantiation $\phi$ such that

(1) $\sigma_i \phi = \tau_i$ for all $i = 1, \ldots, k$;

(2) all type variables in $\mathsf{supp}(\phi)$ occur in the $\sigma_i$.

In cases like this, we will write $\phi = (\tau_1, \ldots, \tau_k)/(\sigma_1, \ldots, \sigma_k)$ or just $\phi = \vec{\tau}/\vec{\sigma}$. This notation will be used in specifying the *reFLect* reduction system in Section 4.

[4]For better readability, we sometimes omit typing annotations in the examples; both occurrences of ${}^\wedge v$ here should read ${}^\wedge (v^{\mathsf{term}})^{\mathsf{nat}}$.

Any type instantiation $\phi$ can be applied to any expression $M$ to produce a new expression, denoted $M\phi$. By definition, $M\phi$ is obtained from $M$ by replacing every active variable occurrence $v^\sigma$ with $v^{\phi(\sigma)}$. The same definition applies to contexts as well. All we need to know about this action of type instantiations on expressions and contexts is summarized in the following lemma and is easy to prove.

LEMMA 5 (INSTANTIATION). *Let $\phi$ be a type instantiation. If $E$ is a (well-formed) expression of type $\sigma$, then $E\phi$ is a (well-formed) expression of type $\phi(\sigma)$. Also, if $E = \mathcal{C}[M_1, \dots, M_k]$, where $\mathcal{C}$ is a level consistent context, then $E\phi = (\mathcal{C}\phi)[M_1\phi, \dots, M_k\phi]$.* $\square$

## 2.6 A Note on Alternations

The original *reFLect* paper [8] uses the combined abstraction/alternation production $M ::= (\lambda M.\, M) + M$ in place of our $M ::= M + M$, so the set of legitimate expressions in [8] is smaller than ours. These PROPER *reFLect* EXPRESSIONS can be characterized as expressions defined by the grammar in Figure 1 that satisfy an additional constraint that in every subexpression of the form $M+N$, the subexpression $M$ must be an abstraction. We have adopted the less restrictive syntax only for reasons of convenience; most results we prove in Sections 3–5 translate with trivial justification into the same results in the context of proper *reFLect* expressions. In particular, a proper *reFLect* expression is well-typed in our system if and only if it is well-typed in [8]. See also Remark 1 in Section 4. Only when we discuss denotational semantics in Section 6 will we have to restrict ourselves to proper *reFLect* expressions.

## 3. $\alpha$-EQUIVALENCE AND SUBSTITUTION

A SUBSTITUTION $\theta$ is a map from variables to expressions. For every $x$, the expression $\theta(x)$ must be well-formed and of the same type as $x$. Moreover, as for type instantiations, it is required that the support set $\mathsf{supp}(\theta) = \{x \mid \theta(x) \neq x\}$ be finite. The goal of this section is to define the *reFLect* notion of $\alpha$-equivalence and the capture-avoiding action of substitutions on $\alpha$-equivalence classes.

## 3.1 Alpha-Equivalence

The standard notions of variable binding and $\alpha$-equivalence extend from the $\lambda$-calculus to *reFLect*, but there is a striking novelty in that passive variable positions are unaffected by binding. For example, there are no free or bound variables in the well-formed expression $\langle\!\langle \lambda x.\, x+y \rangle\!\rangle$ and this expression is not $\alpha$-equivalent to $\langle\!\langle \lambda x'.\, x'+y \rangle\!\rangle$ unless the variables $x$ and $x'$ are equal. In a *reFLect* expression, a variable occurrence can be either passive, binding, bound, or free. It behooves us to begin with a precise definition of this partition.

Suppose $E$ is a well-formed expression. Binding positions of $E$ have been already defined in Section 2.4; they are active variable positions that contain an active prefix of the form $\mu\mathbf{2}$. We say that $\pi$ is a BOUND POSITION if it is an active variable position which has an active prefix $\mu\mathbf{3}$ such that the variable $E|_\pi$ occurs in $\mathsf{PatVar}(E|_{\mu\mathbf{2}})$. FREE POSITIONS of $E$ are active variable positions that are neither binding nor bound. Thus, we have a partition

$$\mathsf{VarPos}(E) = \mathsf{FreePos}(E) \,\cup\, \mathsf{BdPos}(E) \,\cup\, \mathsf{PassPos}(E),$$

where FreePos and PassPos denote the sets of free and passive positions, and BdPos stands for the (disjoint) union of binding and bound positions. We will also write $\mathsf{FreePos}(E, x)$ for the set of all $\pi \in \mathsf{FreePos}(E)$ such that $E|_\pi = x$. Recall that in a pattern all active positions are variable positions and notice that they are all free.

Suppose $\mu$ is an active abstraction position in $E$, $E|_\mu = \lambda L.\, M$, and $x \in \mathsf{PatVar}(L)$. By definition, the CLUSTER corresponding to the pair $(\mu, x)$ is the set

$$\mu\mathbf{2} \cdot \mathsf{FreePos}(L, x) \,\cup\, \mu\mathbf{3} \cdot \mathsf{FreePos}(M, x).$$

Clearly, $E|_\pi = x$ for every position $\pi$ that belongs to this cluster. The type of $x$ will also be called the type of the cluster.

It can be proved now that the family of clusters forms a partition of the set $\mathsf{BdPos}(E)$. We will say that a position $\mu$ BINDS another position $\pi$ if $\mu$ is binding, $\pi$ is bound, and they belong to the same cluster.

Two expressions $E$ and $E'$ will be called $\alpha$-EQUIVALENT (written $E \sim_\alpha E'$) when

($\alpha_1$) $\mathsf{Pos}(E) = \mathsf{Pos}(E')$

($\alpha_2$) The clusters of $E$ and $E'$ are equal and of the same type

($\alpha_3$) $E|_\pi = E'|_\pi$ for all $\pi \in \mathsf{VarPos}(E) - \mathsf{BdPos}(E)$

The condition ($\alpha_1$) implies $\mathsf{PassPos}(E) = \mathsf{PassPos}(E')$ and ($\alpha_2$) implies $\mathsf{BdPos}(E) = \mathsf{BdPos}(E')$. Thus, a position in $E$ is free, binding, bound, or passive if and only if it is of the same kind in $E'$. Moreover, it follows from ($\alpha_3$) that $\mathsf{FreePos}(E, x) = \mathsf{FreePos}(E', x)$ for every $x$.

Clearly, $\sim_\alpha$ is an equivalence relation and $\alpha$-equivalent expressions have the same type. It is also easy to check that $\alpha$-equivalence satisfies the congruence properties:

$$L \cdot M \sim_\alpha L' \cdot M'$$
$$L + M \sim_\alpha L' + M'$$
$$\lambda L.\, M \sim_\alpha \lambda L.\, M'$$
$$\langle\!\langle \mathcal{C}[{}^\wedge M_1{}^{\sigma_1}, \dots, {}^\wedge M_k{}^{\sigma_k}] \rangle\!\rangle \sim_\alpha \langle\!\langle \mathcal{C}[{}^\wedge M_1'{}^{\sigma_1}, \dots, {}^\wedge M_k'{}^{\sigma_k}] \rangle\!\rangle$$

provided each of the subexpressions $L, M, M_i$ is $\alpha$-equivalent to its primed counterpart. (The right-hand side in the third formula could be $\lambda L'.\, M'$, but that would change nothing because $\alpha$-equivalent patterns must be equal.)

Suppose $E \sim_\alpha E'$ and $\mathcal{S}$ is the common set of clusters of these expressions. For every $C \in \mathcal{S}$ there is a position $\mu_C$ and two variables $x_C$ and $x'_C$ such that $C$ corresponds to the pair $(\mu_C, x_C)$ in $E$, and to the pair $(\mu_C, x'_C)$ in $E'$. The expressions $E$ and $E'$ can be obtained from each other by variable renaming at cluster positions:

$$E' = E[\pi \mapsto x'_C]^{\pi \in C \in \mathcal{S}}.$$

Of course, not every variable renaming at cluster positions will turn an expression into an $\alpha$-equivalent one. For a precise analysis of what can go wrong, suppose $C$ is a cluster in $E$ corresponding to the pair $(\mu, x)$ and $z$ is an arbitrary variable of the same type as $x$. The expression $E' = E[\pi \mapsto z]^{\pi \in C}$ will then have a cluster $C'$ corresponding to the pair $(\mu, z)$. This cluster will contain $C$ as a subset, but may also be strictly larger due to "variable capture", which can happen in two ways. First, as in the $\lambda$-calculus, if $z$ occurs free in $E|_\mu$ then $C' = C + \mu \cdot \mathsf{FreePos}(E|_\mu, z)$. The other way for $C' \neq C$ to occur arises from the possibility of patterns that contain multiple variables: if $z$ already exists as a pattern variable in $E|_{\mu\mathbf{2}}$, then we have $C' = C + C_1$, where $C_1$ is the cluster of $E$ corresponding to the pair $(\mu, z)$. It is easy to prove that $C' = C$ holds in all but the two situations just described, and this observation can be used to derive the following standard result.

Lemma 6. *For every expression $E$ and every finite set $X$ of variables, there exists an expression that is $\alpha$-equivalent to $E$ and contains no binding or bound occurrences of any variable in $X$.* $\square$

## 3.2 Substitution Action

If $\theta$ is a substitution and $E$ is a well-formed expression, the result of blind action (not capture-avoiding) of $\theta$ on $E$ is the expression obtained by replacing all occurrences of free variables with their $\theta$-values:

$$\theta(E) = E[\pi \mapsto \theta(E|_\pi)]^{\pi \in \mathsf{FreePos}(E)}$$

We can think of $\theta(E)$ as being obtained from $E$ through a sequence of steps each of which consists of replacing an occurrence of a variable with a well-formed expression of the same type. Thus, by Lemma 4 (Replacement), $\theta(E)$ is a well-formed expression of the same type as $E$.

The blind action does not preserve $\alpha$-equivalence in general. The following lemma (proved in [12]) gives a sufficient condition. It requires a definition: an expression $E$ is free for a substitution $\theta$ when no binding variable of $E$ occurs freely in any of the expressions $\theta(x)$, where $x \in \mathsf{supp}(\theta)$.

Lemma 7. *If $E \sim_\alpha E'$ and both $E$ and $E'$ are free for $\theta$, then $\theta(E) \sim_\alpha \theta(E')$.* $\square$

The capture-avoiding action of substitutions is defined on $\alpha$-equivalence classes. If $E$ is a well-formed expression and $[E]$ is its $\alpha$-equivalence class, we define $[E]\theta$ to be the class $[\theta(E')]$, where $E'$ is a representative of $[E]$ that is free for $\theta$. Existence of $E'$ follows from Lemma 6, and Lemma 7 shows correctness of our definition (independence of the chosen representative $E'$).

From now on, unless otherwise explicitly stated, we will consider $\alpha$-equivalent expressions as equal; that is, by "a well-formed expression" we will really mean "an $\alpha$-equivalence class". We will also suppress the brackets in $[E]$ indicating equivalence classes. When defining functions and relations that take expressions up to $\alpha$-equivalence as arguments, we typically need to check that the definition does not depend on the chosen representatives. The following simple properties give a test for $\alpha$-equivalence that is useful in such proofs: (1) $v^\sigma \sim_\alpha v'^{\sigma'}$ iff $v^\sigma = v'^{\sigma'}$; (2) $L \cdot M \sim_\alpha L' \cdot M'$ iff $L \sim_\alpha L'$ and $M \sim_\alpha M'$; the same statement holds for alternations as well; (3) $\lambda L.\, M \sim_\alpha \lambda L'.\, M'$ iff there exists a variable renaming[5] $\rho$ such that $L' = \rho(L)$ and $M' \sim_\alpha \rho(M)$; (4) $\langle\!\langle \mathcal{C}[{}^\wedge M_1{}^{\sigma_1}, \ldots, {}^\wedge M_k{}^{\sigma_k}] \rangle\!\rangle \sim_\alpha \langle\!\langle \mathcal{C}'[{}^\wedge M_1'{}^{\sigma_1'}, \ldots, {}^\wedge M_k'{}^{\sigma_k'}] \rangle\!\rangle$ iff $\mathcal{C} = \mathcal{C}'$ and (for every $i$) $\sigma_i' = \sigma_i$ and $M_i' \sim_\alpha M_i$.

We will write $E\theta$ for the capture-avoiding substitution action on (the equivalence class of) $E$. The reader should keep in mind that $E\theta$ is only an abbreviation for $[E]\theta$ and that $E\theta = [\theta(E)]$ when $E$ is free for $\theta$.

We have seen above that $\theta(E)$ must have the same type as $E$; now it trivially follows that the type of $E\theta$ is the same as that of $E$. This fact is crucial for the subject reduction property of $\beta$-reduction (Section 4).

Substitution action respects the structure of well-formed expressions, as follows.

Lemma 8. *The capture-avoiding substitution action satisfies (and is characterized by) the following properties:*

$$x\theta = \theta(x)$$
$$(L \cdot M)\theta = (L\theta) \cdot (M\theta)$$
$$(\lambda L.\, M)\theta = \lambda L.\, M\theta$$
$$(L + M)\theta = L\theta + M\theta$$
$$\langle\!\langle \mathcal{C}[{}^\wedge M_1{}^{\sigma_1}, \ldots, {}^\wedge M_k{}^{\sigma_k}] \rangle\!\rangle\theta = \langle\!\langle \mathcal{C}[{}^\wedge (M_1\theta)^{\sigma_1}, \ldots, {}^\wedge (M_k\theta)^{\sigma_k}] \rangle\!\rangle$$

*where, in the case of abstraction, the representative expression $\lambda L.\, M$ is assumed to be free for $\theta$.* $\square$

As usual, we write $[M_1/x_1, \ldots, M_k/x_k]$ for the substitution which maps each $x_i$ to $M_i$ and fixes all other variables. The following standard lemma will be used in the proof of strong normalization in Section 5. We omit a straighforward proof based on Lemma 8.

Lemma 9. *The equation $E[M/x]\theta = (E\theta)[M\theta/x]$ holds provided $x \notin \mathsf{supp}(\theta)$ and $x$ does not occur freely in any $\theta(y)$ for $y \in \mathsf{supp}(\theta)$.* $\square$

## 3.3 Pattern Matching

We will use the notation $\theta\colon L \geqslant M$ to say that the pattern $L$ matches the well-formed expression $M$ via the substitution $\theta$. This matching relation is defined by the following two rules.

$$\frac{M : \sigma \qquad \theta = [M/v^\sigma]}{\theta : v^\sigma \geqslant M}$$

$$\frac{\begin{array}{c}\theta = [\langle\!\langle M_1 \rangle\!\rangle / v_1^{\mathsf{term}}, \ldots, \langle\!\langle M_k \rangle\!\rangle / v_k^{\mathsf{term}}] \\ M_i : \tau_i \qquad \tau_i = \sigma_i\phi \qquad (i = 1, \ldots, k)\end{array}}{\theta\colon \langle\!\langle \mathcal{C}[{}^\wedge(v_1^{\mathsf{term}})^{\sigma_1}, \ldots, {}^\wedge(v_k^{\mathsf{term}})^{\sigma_k}] \rangle\!\rangle \geqslant \langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k] \rangle\!\rangle}$$

**Figure 4: Pattern matching.**

It is easy to check that the definition of pattern matching is correct (independent of the $\alpha$-representatives chosen) and that the substitution $\theta$ in $\theta\colon L \geqslant M$ is uniquely (up to $\alpha$-equivalence) determined by $L$ and $M$.

There are also two things to note when $\theta\colon L \geqslant M$ and $L$ is a quotation pattern: (1) $M$ is a pure quotation; (2) $\theta$ maps every variable in its support set to a pure quotation.

## 4. OPERATIONAL SEMANTICS AND SUBJECT REDUCTION

In this section we prove that the original small step reduction semantics of *reFL$^{ect}$* [8] has the subject reduction property. We actually prove that the subject reduction property is satisfied by a larger but simpler auxiliary reduction system that will also be used in the following section in the proof of strong normalization.

[5]A variable renaming is a substitution that maps variables to variables and is injective on its support set.

Figure 5 defines four kinds of redex-contractum pairs that will define reduction in *reFLect*. In each of them there is a condition (given in brackets) that must be satisfied by the redex. The auxiliary related system that will also be considered is given in Figure 6.

$$
\begin{array}{ll}
(\beta) & (\lambda L.\, M) \cdot N \to M\theta \qquad [\theta \colon L \geqslant N] \\[1em]
(\gamma) & ((\lambda L.\, M) + L') \cdot N \to M\theta \qquad [\theta \colon L \geqslant N] \\[1em]
(\xi) & \begin{array}{l} ((\lambda L.\, M) + L') \cdot N \to L' \cdot N \\ [N \text{ is a pure quotation and } L \text{ does not match } N] \end{array} \\[1.5em]
(\psi) & \begin{array}{l} \langle\!\langle \mathcal{C}[{}^\wedge\langle\!\langle M_1 \rangle\!\rangle^{\sigma_1}, \ldots, {}^\wedge\langle\!\langle M_k \rangle\!\rangle^{\sigma_k}] \rangle\!\rangle \to \langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k] \rangle\!\rangle \\ [M_1 \colon \tau_1, \ldots, M_k \colon \tau_k;\ \mathcal{C} \text{ is level consistent};\ \phi = \vec{\tau}/\vec{\sigma}] \end{array}
\end{array}
$$

**Figure 5: Elementary reductions for $\to_R$.**

$$
\begin{array}{ll}
(\beta') & (\lambda x.\, M) \cdot N \to M[N/x] \\[1em]
(\zeta) & M + N \to M \qquad\qquad (\zeta') \quad M + N \to N \\[1em]
\hline \\
(\beta'') & \begin{array}{l} (\lambda L.\, M) \cdot N \to M\theta \\ [\theta \colon L \geqslant N \text{ and } L \text{ is a quotation pattern}] \end{array} \\[1.5em]
(\psi) & \begin{array}{l} \langle\!\langle \mathcal{C}[{}^\wedge\langle\!\langle M_1 \rangle\!\rangle^{\sigma_1}, \ldots, {}^\wedge\langle\!\langle M_k \rangle\!\rangle^{\sigma_k}] \rangle\!\rangle \to \langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k] \rangle\!\rangle \\ [M_1 \colon \tau_1, \ldots, M_k \colon \tau_k;\ \mathcal{C} \text{ is level consistent};\ \phi = \vec{\tau}/\vec{\sigma}] \end{array}
\end{array}
$$

**Figure 6: Elementary reductions for the systems $\to_A$ (top), $\to_B$ (bottom), and their union $\to_S$.**

LEMMA 10. *Suppose $E \to E'$ is any redex-contractum pair occurring in Figure 5 or Figure 6. If $E$ is well-formed, then $E'$ is well-formed and of the same type as $E$.*

PROOF. Suppose the type of $E$ is $\sigma$ and consider first the rule $(\beta)$, which also subsumes $(\beta')$ and $(\beta'')$. Being an active subexpression of a well-formed expression $E = (\lambda L.\, M) \cdot N$, the expression $M$ is well-formed (Lemma 3) and of type $\sigma$ (by typing rules). It follows that $E' = M\theta$ is well-formed and of type $\sigma$, as noted in Section 3.2. Even though $M$ is not uniquely determined as an $\alpha$-equivalence class by (the $\alpha$-equivalence class of) $E$, the expression $E'$ is uniquely determined by it. This can be easily proved using the definition of pattern matching and the test for $\alpha$-equivalence given in Section 3.2.

For redex-contractum pairs of the form $(\zeta)$ or $(\zeta')$, the lemma follows immediately from the typing rule for alternation and the test for $\alpha$-equivalence.

The case $(\xi)$ is subsumed by $(\zeta')$, and the case $(\gamma)$ follows from the cases $(\zeta)$ and $(\beta'')$ since every $\gamma$-redex decomposes into a $\zeta$-redex followed by a $\beta''$-redex:

$$(\lambda L.\, M + L') \cdot N \to (\lambda L.\, M) \cdot N \to M\theta.$$

The only remaining case is $(\psi)$, where we need to prove

$$\langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k] \rangle\!\rangle \colon \mathsf{term} \tag{1}$$

assuming

$$
\begin{array}{l}
M_i \colon \tau_i \ \text{ for } \ i = 1, \ldots, k \\
\phi = (\tau_1, \ldots, \tau_k)/(\sigma_1, \ldots, \sigma_k) \\
\langle\!\langle \mathcal{C}[{}^\wedge\langle\!\langle M_1 \rangle\!\rangle^{\sigma_1}, \ldots, {}^\wedge\langle\!\langle M_k \rangle\!\rangle^{\sigma_k}] \rangle\!\rangle \colon \mathsf{term}
\end{array}
$$

and assuming also that the expression in the last line is well-formed.

The last assumption and the typing rule for quotations imply

$$\mathcal{C}[z_1^{\sigma_1}, \ldots, z_k^{\sigma_k}] \colon \sigma$$

for some $\sigma$, where the $z_i$ are fresh variables. By Lemma 5 (Instantiation),

$$(\mathcal{C}\phi)[z_1^{\tau_1}, \ldots, z_k^{\tau_k}] \colon \sigma\phi$$

and then by Lemma 4 (Replacement),

$$(\mathcal{C}\phi)[M_1, \ldots, M_k] \colon \sigma\phi$$

Now the claim (1) follows by the typing rule for quotations (the pure quotation case). Independence of representatives is trivial in this case, because both redex and contractum are pure quotations and so unique elements of their respective $\alpha$-equivalence classes. $\square$

If $E$ is a well-formed expression, we define $E \to_\beta E'$ to mean

$(\beta_1)$ $E' = E[\pi \mapsto R]$,

$(\beta_2)$ $\pi$ is an active position in $E$,

$(\beta_3)$ $E|_\pi \to R$ is a $\beta$-redex-contractum pair.

In the same fashion, we define $\to_\gamma$, $\to_\xi$ and $\to_\psi$. The *reFLect* small step reduction semantics is given as the union rewriting system

$$\to_R \;=\; \to_\beta \cup \to_\gamma \cup \to_\xi \cup \to_\psi \;.$$

We also define $\to_S$ as the reduction system defined by rules in Figure 6:

$$\to_S \;=\; \to_{\beta'} \cup \to_{\beta''} \cup \to_\zeta \cup \to_{\zeta'} \cup \to_\psi \;.$$

By Lemma 10, both $\to_R$ and $\to_S$ are well-defined reduction systems on the set of $\alpha$-equivalence classes (of well-formed expressions). Lemma 10 and Lemma 4 (Replacement) also immediately imply our main result:

THEOREM 1 (SUBJECT REDUCTION). *If $E \to_R E'$ or $E \to_S E'$, then $E$ and $E'$ have the same type.* $\square$

*Remark 1.* Clearly, if $E \to_R E'$ and $E$ is a proper *reFLect* expression (Section 2.6), then so is $E'$. The reduction system $\to_R$ restricted to the set of proper *reFLect* expressions coincides with the reduction system defined in [8]. Thus, subject reduction holds for the original *reFLect* system. For the same reason, confluence, once established for $\to_R$, will immediately follow for the original system.

The power of ($\beta$) together with ($\gamma$) and ($\xi$) is, by observations made in the proof of Lemma 10, matched by the power of ($\beta$) together with ($\zeta$) and ($\zeta'$); the last two rules are simpler and symmetric, which makes $\to_S$ a more convenient system to reason about.

Lemma 11. *If $M \to_R^* N$, then $M \to_S^* N$.* $\square$

The reasons for splitting ($\beta$) into two rules ($\beta'$) and ($\beta''$) and for the partition of $\to_S$ into $\to_A$ and $\to_B$ will be clear in the proof of strong normalization (Section 5). For reference there, we also include the following easily proved lemma.

Lemma 12. *Let $\omega$ be any of the reduction symbols $\beta, \gamma, \xi, \psi$.*

*(a) If $E \to_\omega E'$, then $E[M/x] \to_\omega E'[M/x]$.*

*(b) If $M \to_\omega M'$, then $E[M/x] \to_\omega^* E[M'/x]$.* $\square$

## 5. STRONG NORMALIZATION AND CONFLUENCE

Confluence (or the Church-Rosser property) of $\to_R$ asserts that every equivalence class of the equivalence relation generated by $\to_R$ contains exactly one $\to_R$-irreducible expression.

Theorem 2 (Confluence). *The reduction relation $\to_R$ on the set of well-formed reFLect expressions is confluent.*

We will derive Theorem 2 from the local confluence and strong normalization (termination) of $\to_R$. This is a standard way of proving confluence, based on Newman's Lemma (see, for example, [3]).

Lemma 13 (Local confluence). *If $E \to_R E_1$ and $E \to_R E_2$, then there exists $E'$ such that $E_1 \to_R^* E'$ and $E_2 \to_R^* E'$.*

Theorem 3 (Strong normalization). *In either of the reduction systems $\to_R$ and $\to_S$, every reduction sequence terminates.*

In *reFLect* as in the $\lambda$-calculus, local confluence is largely a consequence of coherence of reduction and substitution expressed by Lemma 12. There are a few additional cases that are easily disposed of. A complete proof of Lemma 13 is given in [12].

As for strong normalization, it also can be proved by extending the methods used for proving the same property of simply typed $\lambda$-calculus. This generalization, however, requires a substantial additional effort. First, by Lemma 11, it suffices to prove strong normalization for $\to_S$, and this is the main reason we introduced the system $\to_S$. The strong normalization of this system can be obtained in a modular fashion. Our proof uses the partition of $\to_S$ into into subsystems $\to_A = \to_{\beta'} \cup \to_\zeta \cup \to_{\zeta'}$ and $\to_B = \to_{\beta''} \cup \to_\psi$ (see Section 4). A lemma of Bachmair and Dershowitz [4] now reduces the proof of Theorem 3 to the following tasks:

(1) $\to_A$ and $\to_B$ are strongly normalizing;

(2) $\to_A \circ \to_B \subseteq \to_B \circ (\to_A \cup \to_B)^*$.

For the property (2), which is known as *quasi-commutation*, it will suffice to show that each of the reduction systems $\to_{\beta'}$, $\to_\zeta$, $\to_{\zeta'}$ quasi-commutes with each of $\to_{\beta''}$, $\to_\psi$. The proof is given in [12].

Strong normalization of $\to_B$ is a simple matter. To prove it, we measure the complexity of an expression $E$ by the pair $(m_1, m_2)$, where $m_1$ is the number of active application positions in $E$, and $m_2$ is the size (total number of positions) of $E$. Every application of $\to_{\beta''}$ decreases $m_1$ because the substitution of some variables with pure quotations does not create any new active positions. On the other hand, any application of $\to_\psi$ preserves $m_1$, but reduces the total expression size. Thus, the measure goes down in each step of any $\to_B$ reduction chain, so any such chain must be finite.

It only remains to prove the strong normalization of $\to_A$, which is the most complicated and the most interesting part. To appreciate the problem, suppose we restrict the system $\to_A$ to the subset of expressions that do not use any quotations and abstractions with non-variable patterns. The resulting system, call it $\to_{A'}$, can be seen as the extension of the (polymorphically) typed lambda calculus with the choice operator $+$ constrained by its typing rule as in Figure 3 and reduction rules $\zeta, \zeta'$. If we could prove $\to_{A'}$ is strongly normalizing, we could perhaps with little effort derive strong normalization of $\to_A$. (Quotations and abstractions with non-variable patterns would be treated as constant operators whose arguments are maximal active subexpressions.)

There are several versions of nondeterministic lambda calculi in the literature (cf. [6, 7, 13] and the references there), but it seems that the strong normalization problem for them is being considered only in [6]. Unfortunately, the calculus described in [6] does not exactly match $\to_{A'}$, and it is not clear that the strong normalization for $\to_{A'}$ can be derived from the results of [6]. Therefore, we decide to give a direct proof of strong normalization of $\to_A$.

We prove the strong normalization of $\to_A$ by generalizing the standardization technique for the typed $\lambda$-calculus, as described in Section 2.2 of [2], which we try to follow closely. The generalization is not entirely obvious because we have to deal with the increased complexity caused by the presence of alternation rules. The full proof is given in [12].

## 6. DENOTATIONAL SEMANTICS

In this section we show that the standard set-theoretical denotational semantics of simply typed $\lambda$-calculus extends to *reFLect*. The most interesting aspect of this extension is the use of confluence of the operational semantics. We will write $E^{\mathbf{nf}}$ for the normal form of a well-formed expression $E$ (which is well-defined as a consequence of confluence) and will use the set $\mathbb{Q}$ of all normal forms of type term as the semantical interpretation of that type. The main result of the section is that the operational semantics as defined in Section 4, restricted to the set of proper *reFLect* expressions (see Section 2.6 and Remark 1) is sound with respect to the denotational semantics.

For simplicity, we will define the denotational semantics and prove the soundness theorem only for the fragment of *reFLect* that does not use type variables. In Section 6.3 we explain that there is no serious loss of generality in this.

### 6.1 Semantics of Types and Expressions

By suppressing polymorphism, we restrict ourselves to the set of types that can be generated using the type constant

$\mathsf{term}$ and the function space operator $\Rightarrow$. The INTERPRETATION OF A TYPE $\sigma$ is the set $[\![\sigma]\!]$ defined recursively by equations

$$[\![\mathsf{term}]\!] = \mathbb{Q} \qquad [\![\sigma \Rightarrow \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]}$$

where $[\![\tau]\!]^{[\![\sigma]\!]}$ is the set of all functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$.

We will use the "meta-lambda" notation $\boldsymbol{\lambda} s\colon X.\, A$ to denote the function that maps every element $s$ of $X$ to $A$, where $A$ is some mathematical expression depending on $s$.

An ENVIRONMENT is a finite map that associates to any variable $v^\sigma$ of its domain an element of $[\![\sigma]\!]$. We write $\rho\rho'$ for the environment $\rho$ updated with $\rho'$. More precisely, the domain of $\rho\rho'$ is the union of the domains of $\rho$ and $\rho'$, and $\rho\rho'$ is equal to $\rho'$ on $\mathsf{dom}(\rho')$, and equal to $\rho$ on $\mathsf{dom}(\rho) - \mathsf{dom}(\rho')$. The notation $[s/x]$ will be used for the environment that has only the variable $x$ in its domain, and maps it to $s$.

The INTERPRETATION OF A PROPER WELL-FORMED EXPRESSION $E$ is defined with respect to environments that contain all free variables of $E$. For every such environment $\rho$, the equations in Figure 7 define the interpretation $[\![E]\!]\rho$ as an element of $[\![\mathsf{type}(E)]\!]$. The definition is justified by the inductive structure of the set of well-formed expressions given in Lemma 1.

Figure 7 gives separate equations for abstractions with variable patterns and abstractions with quotation patterns. The notation $\mathsf{Arb}$ used in the second of these equations is for an arbitrary but fixed element of $[\![type(M)]\!]$. It is important to observe that the environment $\rho[Q/L]$ in the same equation is well-defined; even though $[Q/L]$ is defined as a substitution (namely, that $\theta$ for which the pattern matching relation $\theta\colon L \gtrdot Q$ holds), it maps all variables in its support set to pure quotations. Since pure quotations are in $\mathbb{Q}$, the substitution $[Q/L]$ can be regarded as an environment as well.

Note that we define interpretations of "proper" alternations only. Indeed, there is no reasonable way of giving a general definition of $[\![M + N]\!]\rho$ purely in terms of $[\![M]\!]\rho$ and $[\![N]\!]\rho$.

Interpretations depend only on the values the environment assigns to free variables; this is expressed by the following lemma and easily proved by induction.

LEMMA 14. *If the environments $\rho$ and $\rho'$ agree on the set of free variables of $E$, then $[\![E]\!]\rho = [\![E]\!]\rho'$.* □

We will refer several times to the following fact that immediately follows from the definitions.

LEMMA 15. *If $E$ is a pure quotation, then $[\![E]\!]\rho = E$ holds for every environment $\rho$.* □

## 6.2 Soundness Theorem

THEOREM 4 (SOUNDNESS). *Suppose $E$ is a proper well-formed expression and $\rho$ is an environment that contains all free variables of $E$. If $E \to_R E'$ then $[\![E]\!]\rho = [\![E']\!]\rho$.*

We begin the proof of Theorem 4 with an appropriate substitution lemma.

LEMMA 16. *Let $M$ and $N$ be well-formed expressions. Then $[\![M[N/x]]\!]\rho = [\![M]\!]\rho'$, where $\rho' = \rho\,[[\![N]\!]\rho/x]$, and $\rho$ is any environment whose domain contains all free variables of $M[N/x]$.*

PROOF. The cases when $M$ is a variable or application are easy.

Suppose $M$ is an abstraction with a variable pattern, $M = \lambda z.\, M_1$. We may asume that $z$ is not free in $N$ and we have

$$\begin{aligned} & [\![(\lambda z.\, M_1)[N/x]]\!]\rho = [\![\lambda z.\, M_1[N/x]]\!]\rho \\ = \;& \boldsymbol{\lambda} s.\, [\![M_1[N/x]]\!]\rho_1 = \boldsymbol{\lambda} s.\, [\![M_1]\!]\left(\rho_1\,[[\![N]\!]\rho_1/x]\right), \end{aligned}$$

where $\rho_1 = \rho[s/z]$. The first equation is a property of substitutions, the second follows from the first semantic eqaution for abstractions, and the third holds by the induction hypothesis.

On the other hand,

$$[\![\lambda z.\, M_1]\!]\rho' = \boldsymbol{\lambda} s.\, [\![M_1]\!](\rho'[s/z]) = \boldsymbol{\lambda} s.\, [\![M_1]\!](\rho_1[[\![N]\!]\rho/x]),$$

where we again use the first semantic equation for abstractions and commutativity of independent updates of environments.

To finish the proof in this case, we just need to check that $[\![N]\!]\rho = [\![N]\!]\rho_1$, but this is an instance of Lemma 14.

We move to the next case, where $M = \lambda L.\, M_1$, an abstraction with a quotation pattern. We may assume that the pattern variables of $L$ do not occur freely in $N$. Both functions $[\![(\lambda L.\, M_1)[N/x]]\!]\rho$ and $[\![\lambda L.\, M_1]\!]\rho'$ when applied to a quotation $Q$ that does not match $L$ produce $\mathsf{Arb}$ as a result. Thus, we may assume that $L$ matches $Q$. Similarly as above, we have

$$\begin{aligned} & ([\![(\lambda L.\, M_1)[N/x]]\!]\rho)(Q) = ([\![\lambda L.\, M_1[N/x]]\!]\rho)(Q) \\ = \;& [\![M_1[N/x]]\!]\rho_1 = [\![M_1]\!](\rho_1[[\![N]\!]\rho_1/x]), \end{aligned}$$

where $\rho_1 = \rho[Q/L]$. Also,

$$([\![\lambda L.\, M_1]\!]\rho')(Q) = [\![M_1]\!](\rho'[Q/L]) = [\![M_1]\!](\rho_1[[\![N]\!]\rho/x]),$$

and the proof follows from $[\![N]\!]\rho = [\![N]\!]\rho_1$, which is again true by Lemma 14. (Notice that the free variables of $N$ are not in the domain of $[Q/L]$).

For the case when $M$ is an alternation, we need to prove

$$[\![M_1[N/x] + M_2[N/x]]\!]\rho = [\![M_1 + M_2]\!]\rho', \tag{2}$$

since the left-hand side is clearly equal to $[\![(M_1{+}M_2)[N/x]]\!]\rho$. The expression $M_1$ is an abstraction; let $L$ be its pattern. Applying the two sides of (2) to an arbitrary value $Q \in [\![\mathsf{type}(L)]\!]$ produces $([\![M_i[N/x]]\!]\rho)(Q)$ and $([\![M_i]\!]\rho')(Q)$ respectively, where $i$ is 1 or 2 depending on whether $L$ matches $Q$ or not. In both cases, the two values are equal by induction hypothesis.

The last case to consider is when $M$ is a quotation. The proof is a straightforward chain of equalities, so we omit its details. □

COROLLARY 2. *Suppose $\pi$ is an active non-pattern and non-binding position in a well-formed expression $E$, $\rho$ is an environment for $E$, and $M$ is a well-formed expression such that $[\![E|_\pi]\!]\rho = [\![M]\!]\rho$. Then $[\![E]\!]\rho = [\![E[\pi \mapsto M]]\!]\rho$.*

PROOF. Let $E' = E[\pi \mapsto z]$, where $z$ is a fresh variable. The position $\pi$ is free in $E'$ and Lemma 16 implies

$$\begin{aligned} & [\![E]\!]\rho = [\![E'[E|_\pi/z]]\!]\rho = [\![E']\!]\rho\,[[\![E|_\pi]\!]\rho/z] \\ = \;& [\![E']\!]\rho\,[[\![M]\!]\rho/z] = [\![E'[M/z]]\!]\rho = [\![E[\pi \mapsto M]]\!]\rho. \end{aligned}$$

□

$$[\![v^\sigma]\!]\rho = \rho(v^\sigma)$$

$$[\![M \cdot N]\!]\rho = ([\![M]\!]\rho)([\![N]\!]\rho)$$

$$[\![\lambda v^\sigma .\, M]\!]\rho = \boldsymbol{\lambda} s\colon [\![\sigma]\!].\ [\![M]\!](\rho[s/v^\sigma])$$

$$[\![\lambda L .\, M]\!]\rho = \boldsymbol{\lambda} Q\colon \mathbb{Q}\,.\begin{cases} [\![M]\!](\rho[Q/L]) & \text{if } L \text{ is a quotation pattern and it matches } Q \\ \mathsf{Arb} & \text{otherwise}\end{cases}$$

$$[\![(\lambda L .\, M) + N]\!]\rho = \boldsymbol{\lambda} Q\colon [\![\mathit{type}(L)]\!]\,.\begin{cases} ([\![\lambda L.\, M]\!]\rho)(Q) & \text{if } L \text{ is a variable, or } L \text{ matches } Q \\ ([\![N]\!]\rho)(Q) & \text{otherwise}\end{cases}$$

$$[\![\langle\!\langle \mathcal{C}[{}^\wedge M_1{}^{\sigma_1}, \ldots, {}^\wedge M_k{}^{\sigma_k}]\rangle\!\rangle]\!]\rho = \langle\!\langle \mathcal{C}[{}^\wedge([\![M_1]\!]\rho)^{\sigma_1}, \ldots, {}^\wedge([\![M_k]\!]\rho)^{\sigma_k}]\rangle\!\rangle^{\mathbf{nf}}$$

**Figure 7: Semantic equations.**

In view of Corollary 2, for the proof of Theorem 4 it remains now to check that $[\![E]\!]\rho = [\![E']\!]\rho$ holds for every redex/contractum pair $(E, E')$ and every applicable environment $\rho$.
*Case* $(\beta)$. When the pattern in the participating abstraction is a variable, we need to prove

$$[\![(\lambda x.\, M) \cdot N]\!]\rho = [\![M[N/x]]\!]\rho.$$

By semantic equations for applications and abstractions,

$$[\![(\lambda x.\, M) \cdot N]\!]\rho = ([\![\lambda x.\, M]\!]\rho)([\![N]\!]\rho) = [\![M]\!](\rho[[\![N]\!]\rho/x]),$$

and Lemma 16 finishes the proof.

For the case when the pattern is a quotation, we need to prove

$$[\![(\lambda L.\, M) \cdot N]\!]\rho = [\![M[N/L]]\!]\rho,$$

where $N$ is a pure quotation and the substitution $[N/L]$ can be written as

$$[N/L] = [M_1/x_1] \cdots [M_k/x_k],$$

where $M_1, \ldots, M_k$ are pure quotations. Using the semantic equation for applications and Lemma 15, we have

$$[\![(\lambda L.\, M) \cdot N]\!]\rho = ([\![\lambda L.\, M]\!]\rho)([\![N]\!]\rho) = ([\![\lambda L.\, M]\!]\rho)(N).$$

We know that $L$ matches $N$, so the second semantic equation for abstractions gives

$$\begin{aligned} & ([\![\lambda L.\, M]\!]\rho)(N) = [\![M]\!](\rho[N/L]) \\ = \; & [\![M]\!](\rho[M_1/x_1] \cdots [M_k/x_k]),\end{aligned}$$

and we need to prove this is equal to $[\![M[M_1/x_1] \cdots [M_k/x_k]]\!]\rho$. The proof follows by $k$ applications of Lemma 16 together with Lemma 15. (Notice that every $M_i$ is a pure quotation).
*Case* $(\gamma)$. We only need to check that

$$[\![(\lambda L.\, M + L') \cdot N]\!]\rho = [\![(\lambda L.\, M) \cdot N]\!]\rho$$

to reduce the proof of this case to Case $(\beta)$ above. By the semantic equation for applications, our goal can be rewritten as

$$([\![(\lambda L.\, M + L')]\!]\rho)([\![N]\!]\rho) = ([\![(\lambda L.\, M)]\!]\rho)([\![N]\!]\rho).$$

This follows from the semantic equation for alternations. Just note that if $L$ is a quotation, we have $[\![N]\!]\rho = N$ (Lemma 15), and $L$ matches $N$.
*Case* $(\xi)$. Our goal is now

$$[\![(\lambda L.\, M + L') \cdot N]\!]\rho = [\![L' \cdot N]\!]\rho.$$

Using the semantic equations and the assumption that $N$ is a pure quotation that does not match $L$, both sides reduce easily to $([\![L']\!]\rho)(N)$.
*Case* $(\psi)$. We need to prove

$$[\![\langle\!\langle \mathcal{C}[{}^\wedge\langle\!\langle M_1\rangle\!\rangle^{\sigma_1}, \ldots, {}^\wedge\langle\!\langle M_k\rangle\!\rangle^{\sigma_k}]\rangle\!\rangle]\!]\rho = [\![\langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k]\rangle\!\rangle]\!]\rho,$$

where $\mathcal{C}, M_i, \phi$ are as given in the conditions of the rule $(\psi)$. By Lemma 15, the right-hand side is just $\langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k]\rangle\!\rangle$. The left-hand side, by the semantic equation for quotations, is equal to $\langle\!\langle \mathcal{C}[{}^\wedge([\![\langle\!\langle M_1\rangle\!\rangle]\!]\rho)^{\sigma_1}, \ldots, {}^\wedge([\![\langle\!\langle M_k\rangle\!\rangle]\!]\rho)^{\sigma_k}]\rangle\!\rangle^{\mathbf{nf}}$. Since each $\langle\!\langle M_i\rangle\!\rangle$ is a pure quotation, by Lemma 15 the whole expression reduces to $\langle\!\langle \mathcal{C}[{}^\wedge\langle\!\langle M_1\rangle\!\rangle^{\sigma_1}, \ldots, {}^\wedge\langle\!\langle M_k\rangle\!\rangle^{\sigma_k}]\rangle\!\rangle^{\mathbf{nf}}$, and this normal form is indeed equal to $\langle\!\langle (\mathcal{C}\phi)[M_1, \ldots, M_k]\rangle\!\rangle$.

This finishes the proof of Theorem 4.

## 6.3 Handling Polymorphism

The denotational semantics of the monomorphic fragment of *reFL^ect* presented in Section 6.1 extends to the full language in the same way the frame semantics of the simply typed $\lambda$-calculus extends to a semantics of the polymorphically typed $\lambda$-calculus (system $\lambda\!\rightarrow$ of [10], or ML$_0$ of [9] without the polymorphic *let*).

For this extension, we need a type universe $\mathcal{U}$—a family of sets that contains $\mathbb{Q}$ and is closed under the set-theoretic function space operation. A TYPE ENVIRONMENT is an arbitrary finite map from type variables to $\mathcal{U}$. Types are interpreted in the context of type environments: for every type $\sigma$ and every type environment $\iota$ whose domain contains all type variables occurring in $\sigma$, the interpretation $[\![\sigma]\!]\iota$ is defined recursively by

$$[\![\mathsf{term}]\!]\iota = \mathbb{Q} \qquad [\![\alpha]\!]\iota = \iota(\alpha) \qquad [\![\sigma \Rightarrow \tau]\!]\iota = ([\![\tau]\!]\iota)^{[\![\sigma]\!]\iota}$$

For every well-formed expression $E$ of type $\sigma$, the interpretation $[\![E]\!]\iota\rho$ is an element of $[\![\sigma]\!]\iota$. It is defined for every type environment $\iota$ whose domain contains all type variables occurring in $E$ and for every $\iota$-environment $\rho$ whose domain contains all free variables of $E$. Here, when we say that $\rho$ is an $\iota$-environment, we mean that $\rho(v^\sigma) \in [\![\sigma]\!]\iota$, for every $v^\sigma \in \mathsf{dom}(\rho)$. The semantic equations that recursively define $[\![E]\!]\iota\rho$ are the same as those given in Figure 7; we only need to replace $[\![\ ]\!]\rho$ everywhere with $[\![\ ]\!]\iota\rho$ and to replace $\boldsymbol{\lambda}s\colon [\![\sigma]\!]$ in the third equation with $\boldsymbol{\lambda}s\colon [\![\sigma]\!]\iota$. All results and proofs generalize in a straightforward way.

## 7. ACKNOWLEDGMENTS

We are grateful to Jim Grundy, John O'Leary and Tom Melham for their help in understanding *reFLect*; and to Andy Moran, Sylvain Conchon and Alexandre Miquel for helping us track down references to non-deterministic lambda calculus. We would especially like to thank Jim Grundy for carefully reading and debugging various versions of our paper.

## 8. REFERENCES

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[2] R. M. Amadio and P.-L. Curien. *Domains and Lambda-Calculi*. Cambridge University Press, 1998.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction (CADE)*, volume 230 of *LNCS*, pages 5–20. Springer, 1986.

[5] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages (POPL 2003)*. ACM, 2003.

[6] P. de Groote. Strong normalization in a non-deterministic typed lambda-calculus. In *Logical Foundations of Computer Science*, pages 142–152, 1994.

[7] M. Dezani-Ciancaglini, U. de'Liguoro, and A. Piperno. A filter model for concurrent lambda-calculus. *SIAM J. Comput.*, 27(5):1376–1419, 1998.

[8] J. Grundy, J. O'Leary, and T. Melham. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Oxford University Computing Laboratory, 2003.

[9] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[10] B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.

[11] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[12] S. Krstić and J. Matthews. Subject reduction and confluence for the *reFLect* language. Technical Report CSE-03-014, OGI, 2003.

[13] C.-H. L. Ong. Non-determinism in a functional setting. In *8th Symposium on Logic in Computer Science (LICS)*, pages 275–286. IEEE Computer Society Press, 1993.

[14] T. Sheard and S. P. Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, 2002.

[15] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[16] V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.