Yale University
Department of Computer Science

The ∇-Calculus. Functional Programming with Higher-order Encodings.

Carsten Schürmann, Adam Poswolsky, Jeffrey Sarnat

YALEU/DCS/TR-1272
November 2, 2004, v1.0

This research is funded by NFS under grants CCR-0133502 and CCR-0325808.

Abstract

Higher-order encodings use functions provided by one language to represent variable binders of another. They lead to concise and elegant representations, which historically have been difficult to analyze and manipulate.

In this paper we present the ∇-calculus, a calculus for defining general recursive functions over higher-order encodings. To avoid problems commonly associated with using the same function space for representations and computations, we separate one from the other. The simply-typed λ-calculus plays the role of the representation-level. The computation-level contains not only the usual computational primitives but also an embedding of the representation-level. It distinguishes itself from similar systems by allowing recursion under representation-level λ-binders while permitting a natural style of programming which we believe scales to other logical frameworks. Sample programs include bracket abstraction, parallel reduction, and an evaluator for a simple language with first-class continuations.

1 Introduction

Higher-order abstract syntax refers to the technique of using a meta-language, or logical framework, to encode an object language in such a way that variables of the object language are represented by the variables of the logical framework. This deceptively simple idea has far reaching consequences for the design of languages that aim to manipulate these encodings. On one hand, higher-order encodings are often very concise and elegant since they take advantage of common concepts and operations automatically provided by the logical framework, including variable renaming, capture avoiding substitutions, and hypothetical judgments. On the other hand, higher-order encodings are not inductive in the usual sense, which means that they are difficult to analyze and manipulate.

Many attempts have been made to integrate advanced encoding techniques into functional programming languages. FreshML [GP99] supports implicit variable renaming for first-order encodings. The modal λ-calculus supports primitive recursion over higher-order encodings via an iterator. However, function definition via iteration is naturally limited [SDP01].

In this paper, we present the ∇-calculus, a step towards integrating logical frameworks into functional programming. It supports *general* recursive functions over higher-order encodings without burdening the representational expressiveness of the logical framework. The ∇-calculus distinguishes itself from similar systems by allowing recursion under representation-level λ-binders while permitting a natural style of programming, which we believe scales to other logical frameworks.

To avoid problems commonly associated with using the same function space for representations and computations, we separate one from the other. The simply-typed λ-calculus plays the role of the representation-level and provides a function space enabling higher-order encodings. A second simply-typed language plays the role of the computation-level. It provides embeddings of the higher-order encodings, function definition by cases, and insurances for safe returns from computation under representation-level λ-binders.

The resulting system allows us, for example, to write computation-level functions that recurse over the usual higher-order encoding of the untyped λ-calculus (see Example 4). It is general enough to permit case analysis over any representation-level object of any representation-level type. A prototype implementation [PS04] of the ∇-calculus, including a type-checker, an interactive runtime-system, and a collection of examples is available from the website `http://www.cs.yale.edu/~delphin`.

This paper is organized as follows. We explain the use of the simply-typed λ-calculus as a logical framework in Section 2. We introduce the ∇-calculus in Section 3. It is divided into several subsections describing the conventional features of the ∇-calculus and those constructs that facilitate programming with higher-order encodings. The static and operational semantics of the ∇-calculus are given in Section 4, while the meta-theoretic properties of the calculus are discussed and analyzed in Section 5. We assess results and discuss related and future work in Section 6.

2 The Simply-Typed Logical Framework

We choose the simply-typed λ-calculus as our logical framework. It is not as expressive as dependently-typed frameworks, such as LF [HHP93], but is expressive enough to permit interesting higher-order encodings.

$$\begin{array}{lll} \text{Types:} & A, B & ::= a \mid A \rightarrow B \\ \text{Objects:} & M, N & ::= x \mid c \mid \lambda x : A.\, M \mid M\ N \\[6pt] \text{Signatures:} & \Sigma & ::= \cdot \mid \Sigma, a : \mathit{type} \mid \Sigma, c : A \\ \text{Contexts:} & \Gamma & ::= \cdot \mid \Gamma, x : A \end{array}$$

We use a for type constants, c for object constants, and x for variables. We assume that constants and variables are declared at most once in a signature and context, respectively. To maintain this invariant, we tacitly rename bound variables and use capture-avoiding substitutions. The typing judgments for objects and signatures are standard. Type-level and term-level constants must be declared in the signature.

Definition 1 (Typing judgment). *$\Gamma \vdash M : A$ and $\Gamma \vdash A : \mathit{type}$ is defined by the following rules:*

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \mathsf{ofvar} \qquad \frac{\Sigma(c) = A}{\Gamma \vdash c : A}\ \mathsf{ofconst}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \rightarrow B}\ \mathsf{oflam} \qquad \frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \mathsf{ofapp}$$

$$\frac{\Sigma(a) = \mathit{type}}{\Gamma \vdash a : \mathit{type}}\ \mathsf{tpconst} \qquad \frac{\Gamma \vdash A : \mathit{type} \qquad \Gamma \vdash B : \mathit{type}}{\Gamma \vdash A \rightarrow B : \mathit{type}}\ \mathsf{tpArrow}$$

Our notion of definitional equality is obtained by taking the reflexive, transitive, and symmetric closure of β- and η-conversion [Coq91]. We write $\Gamma \vdash M \equiv N : A$ if and only if M is $\beta\eta$-equivalent to N and both have type A. For every well-typed object M of type A, there exists a unique β-normal, η-long term M' such that $\Gamma \vdash M \equiv M' : A$ [Pfe92]. We refer to M' as being canonical, which we denote as $\Gamma \vdash M' \Uparrow A$.

Throughout this paper, our examples will use encodings of natural numbers, first-order logic, and the untyped λ-calculus. An encoding consists of a signature and a representation function, which maps elements from our domain of discourse into canonical forms in our logical framework. We say that an encoding is *adequate* if the representation function is an isomorphism.

In all of the examples below, the signatures for our encoding are listed in italics and our translation functions $\ulcorner - \urcorner$ are defined by the given sets of equations.

Example 1 (Booleans). Boolean values $b ::= \mathbf{true} \mid \mathbf{false}$ can be represented as objects of type *bool* over the signature which includes the following declaration:

$$\begin{array}{rl} \ulcorner\mathbf{true}\urcorner = \mathit{true} & \mathit{true} : \mathit{bool} \\ \ulcorner\mathbf{false}\urcorner = \mathit{false} & \mathit{false} : \mathit{bool} \end{array}$$

□

Example 2 (Natural numbers).

$$\begin{array}{rl} & \mathit{nat} : \mathit{type} \\ \ulcorner 0\urcorner = z & z \;\; : \mathit{nat} \\ \ulcorner n+1\urcorner = s\,\ulcorner n\urcorner & s \;\; : \mathit{nat} \to \mathit{nat} \end{array}$$

□

Examples 2 and 1 are first-order encodings because none of the constants take arguments of functional types.

Example 3 (First order logic with equality). Terms $t ::= x$ and first order formulas $F ::= \forall x.\, F \mid F_1 \supset F_2 \mid \neg F \mid t_1 = t_2$ are represented as objects of type i and type o, respectively, in a signature that also includes the following declarations:

$$\begin{array}{rl} \ulcorner\forall x.\, F\urcorner = \mathit{forall}\,(\lambda x : i.\ulcorner F\urcorner) & \mathit{forall} : (i \to o) \to o \\ \ulcorner\neg F\urcorner = \mathit{neg}\,\ulcorner F\urcorner & \mathit{neg} \;\; : o \to o \\ \ulcorner F_1 \supset F_2\urcorner = \mathit{impl}\,\ulcorner F_1\urcorner\,\ulcorner F_2\urcorner & \mathit{impl} \;\; : o \to o \to o \\ \ulcorner t_1 = t_2\urcorner = \mathit{eq}\,\ulcorner t_1\urcorner\,\ulcorner t_2\urcorner & \mathit{eq} \;\;\; : i \to i \to o \\ \ulcorner x\urcorner = x & \end{array}$$

Example 4 (Untyped λ-expressions). Untyped λ-expressions $e ::= x \mid \mathbf{lam}\ x.\, e \mid e_1\ e_2$ are encoded as follows:

$$\begin{array}{rl} & \mathit{exp} \;\, : \mathit{type} \\ \ulcorner\mathbf{lam}\ x.\, e\urcorner = \mathit{lam}\,(\lambda x : \mathit{exp}.\ulcorner e\urcorner) & \mathit{lam} : (\mathit{exp} \to \mathit{exp}) \to \mathit{exp} \\ \ulcorner e_1\ e_2\urcorner = \mathit{app}\,\ulcorner e_1\urcorner\,\ulcorner e_2\urcorner & \mathit{app} : \mathit{exp} \to \mathit{exp} \to \mathit{exp} \\ \ulcorner x\urcorner = x & \end{array}$$

The encodings of first-order formulas and λ-calculus expressions illustrate the use of higher-order abstract syntax since object-language variable-binders use logical-framework functions. Because little meaningful analysis can be done on variables in our logical framework, the only interesting operation that can be performed on a variable is substitution. Thus, it is most helpful to think of a term of type $A \to B$ not as representing a computation, but as representing a term of type B that has a hole of type A.

We demonstrate the formulation of an adequacy theorem. Each case can be proven by a straightforward induction.

Theorem 1 (Adequacy of *exp*). *Adequacy holds for our representation of untyped λ-expressions.*

$$\begin{array}{ll}
\text{Types:} & \tau, \sigma ::= \langle A \rangle \mid \tau \Rightarrow \sigma \mid \tau \star \sigma \mid \Box\tau \\
\text{Expressions:} & e, f ::= u \mid \langle M \rangle \mid e_1 \mapsto_\tau e_2 \mid \epsilon x : A.\, e \mid \epsilon u \in \tau.\, e \\
& \quad\mid e_1 \cdot e_2 \mid (e_1 \mid e_2) \mid \mathrm{rec}\ u \in \tau.\, e \\
& \quad\mid (e_1, e_2) \mid \mathrm{fst}\ e \mid \mathrm{snd}\ e \mid \mathrm{empty} \\
& \quad\mid \nu x : A.\, e \mid \mathrm{pop}\ e \mid \nabla x : A.\, e
\end{array}$$

Fig. 1. Syntactic categories of the ∇-calculus.

1. *If* e *is an expression with free variables among* $x_1, \ldots, x_n$, *then* $x_1 : exp, \ldots, x_n : exp \vdash \ulcorner e \urcorner \Uparrow exp$.
2. *If* $x_1 : exp, \ldots, x_n : exp \vdash M \Uparrow exp$ *then* $M = \ulcorner e \urcorner$ *for some expression* e *with free variables among* $x_1, \ldots, x_n$.
3. $\ulcorner - \urcorner$ *is a bijection between expressions and canonical forms where* $\ulcorner [e'/x]e \urcorner = [\ulcorner e' \urcorner / x] \ulcorner e \urcorner$. $\Box$

3 The ∇-Calculus

The logical-framework type *exp* is not inductive because the constructor *lam* : $(exp \to exp) \to exp$ has a negative occurrence [PM93] of *exp*. This is not just a formal observation, since this property has deep consequences for the design of the ∇-calculus, which needs to provide a notion of computation general enough to handle higher-order datatypes of this kind. We offer the ability to recurse under λ-binders and consider cases over functions of type $exp \to exp$ while continuing to guarantee the adequacy of the encoding. Allowing for this, as well as general recursive computation, can be seen as the main contribution of this work.

In the ∇-calculus, expressions permit function definition by cases and alternations instead of providing explicit λ-binders on the computation-level. Computation-level expressions and types are summarized in Figure 1 and explained in the remainder of this section.

3.1 Function Definition by Cases and Recursion

In the ∇-calculus, we draw a separating line between the levels of representation and computation. Representation-level types, such as *nat* and *exp* are *injected* into computation-level types $\langle nat \rangle$ and $\langle exp \rangle$. Likewise, representation-level constants, such as $(s\ z)$ and $lam\ (\lambda x : exp.\, x)$, are injected into computation-level terms $\langle s\ z \rangle$ and $\langle lam\ (\lambda x : exp.\, x) \rangle$. There are no user defined datatypes on the computation-level; all type and constant declarations must be done at the representation-level.

Example 5 (Addition). We informally define the function *plus* over the representation of natural numbers from Example 2 in the following manner:

$$\begin{array}{ll}
plus\ z\ y & = y \\
plus\ (s\ x)\ y & = s\ (plus\ x\ y)
\end{array}$$

We represent this formally in the ∇-calculus as follows:

$$\begin{array}{l}\text{rec}\ \mathit{plus} \in \langle \text{nat} \rangle \Rightarrow \langle \text{nat} \rangle \Rightarrow \langle \text{nat} \rangle. \\ \quad \epsilon y : \text{nat}.\, \langle \text{z} \rangle \mapsto \langle y \rangle \mapsto \langle y \rangle \\ \quad \mid\ \ \epsilon x : \text{nat}.\, \langle \text{s}\ x \rangle \mapsto \epsilon y : \text{nat}.\, \langle y \rangle \mapsto \langle \text{s} \rangle \circ (\mathit{plus} \cdot \langle x \rangle \cdot \langle y \rangle)\end{array}$$

$\square$

The recursion operator is conventional. In later examples we will omit it for the sake of readability. Alternation, "|", separates cases that may be chosen for evaluation non-deterministically. It binds more tightly than the recursion operator rec $u \in \tau.\, e$, but not as tight as any of the other operators. Individual cases are of the form $e_1 \mapsto_\tau e_2$, where e_1 can be thought of as a guard. Only when such a case is applied to an object equivalent to e_1 (as defined in Section 4.3) is e_2 evaluated. In particular, if e_1 is a value of type $\langle A \rangle$, then our notion of equality is given by our logical framework's notion of definitional equality. We refer to e_1 as the *pattern* and e_2 as the *body* of the case. The index τ states the type of the pattern, but is usually omitted when the type of the pattern can be easily inferred. In conventional programming languages, variables that occur in patterns are implicitly declared, whereas in the ∇-calculus they must be declared explicitly by $\epsilon x : A.\, e$ for reasons explained in Section 3.2. A similar declaration for the computation-level $\epsilon u \in \tau.\, e$ permits higher-order functions and is discussed in detail in Section 3.5. Application in the ∇-calculus is written as $e_1 \cdot e_2$ in order to avoid confusion with representation-level application, which is expressed via juxtaposition. The notation $e_1 \circ e_2$ is syntactic sugar that lifts representation-level application to the computation-level.

$$e_1 \circ_{A,B} e_2 = \epsilon x : A \to B.\, \langle x \rangle \mapsto_{\langle A \to B \rangle} \epsilon y : A.\, \langle y \rangle \mapsto_{\langle A \rangle} \langle x\ y \rangle$$

We refer to $\circ$ without type annotations because they are easily inferable.

3.2 Traversal of λ-Binders

Next, we explain the operators ν and pop from Figure 1. Recall the encoding of first-order logic from Example 3.

As a running example, we consider Kolmogorov's double-negation interpretation, which transforms formulas from classical logic into intuitionistic logic in the following way:

$$\begin{array}{lll} \mathit{dneg}\,(\mathit{eq}\, t_1\ t_2) & = \mathit{neg}\,(\mathit{neg}\,(\mathit{eq}\, t_1\ t_2)) \\ \mathit{dneg}\,(\mathit{impl}\, F_1\ F_2) & = \mathit{neg}\,(\mathit{neg}\,(\mathit{impl}\,(\mathit{dneg}\, F_1)\ (\mathit{dneg}\, F_2))) \\ \mathit{dneg}\,(\mathit{neg}\, F) & = \mathit{neg}\,(\mathit{neg}\,(\mathit{neg}\,(\mathit{dneg}\, F))) \\ \mathit{dneg}\,(\mathit{forall}\, F) & = \mathit{neg}\,(\mathit{neg}\,(\mathit{forall}\, F')) \\ & \qquad \text{where } F'\ x = \mathit{dneg}\,(F\ x) \\ & \qquad \text{for some new parameter } x : i \end{array}$$

In the last case *dneg* must recurse on the body F of the *forall* term, which is a representation-level function of type $i \to o$. Since F is definitionally equivalent to a canonical term that starts with a λ-binder, we strip away the λ-binder by applying F to some new parameter x before invoking *dneg*. The result of the computation depends on x and is hence written as $F'\ x$, where F' is a representation-level function of type $i \to o$.

The first three cases of *dneg* can be implemented in the ∇-calculus with constructs we have already introduced. As for the *forall* case, we need to add new constructs to our language. We feel that there are several interesting possibilities worth considering. One possibility would be to introduce a computation-level operator $\hat{\lambda}$, which lifts representation-level abstraction to the computation-level in much the same way that the syntactic-sugar $\circ$ lifts representation-level application. In this case, we could write the *forall* case as

$$\epsilon F : \mathrm{i} \to \mathrm{o}.\, \langle \mathrm{forall}\ F\rangle \mapsto \langle \mathrm{neg}\rangle \circ (\langle \mathrm{neg}\rangle \circ (\langle \mathrm{forall}\rangle \circ (\hat{\lambda} x : \mathrm{i}.\, \mathit{dneg} \cdot \langle F\ x\rangle)))$$

where the subterm $(\hat{\lambda} x : \mathrm{i}.\, \mathit{dneg} \cdot \langle F\ x\rangle)$ has type $\langle i \to o\rangle$. In principle this is a possible solution. Adequacy is preserved because although the body of $\hat{\lambda}$ may diverge or get stuck, any value it computes must be of the form $\langle M\rangle$. However, $\hat{\lambda}$ is too limited for our purposes because it always returns a representation-level function, even if the expected result is of a base type (see Example 6). Meta-ML [TS00] employs a construct similar to $\hat{\lambda}$.

Another possibility is to add an explicit parameter introduction operator $\bar{\lambda}$

$$\begin{aligned}\epsilon F : \mathrm{i} \to \mathrm{o}.\, \langle \mathrm{forall}\ F\rangle \mapsto\ & \bar{\lambda} x : \mathrm{i}.\\ & \mathrm{case}\ \mathit{dneg} \cdot \langle F\ x\rangle\\ & \quad \mathrm{of}\ \epsilon F' : \mathrm{i} \to \mathrm{o}.\, \langle F'\ x\rangle \mapsto \langle \mathrm{neg}\ (\mathrm{neg}\ (\mathrm{forall}\ F'))\rangle\end{aligned}$$

where we write "case e_1 of e_2" as syntactic sugar for "$e_2 \cdot e_1$". In contrast to $\hat{\lambda}$, the type of the subterm starting with $\bar{\lambda}$ is $\langle o\rangle$. Since the recursive call results in a value of type $\langle o\rangle$, and *forall* requires a value of type $i \to o$, we need a way to turn the result into a value of type $\langle i \to o\rangle$. Furthermore, because this value escapes x's declaration, it should not contain any free occurrences of x. Ideally, higher-order pattern matching would yield F', which is the result of abstracting all occurrences of x from the result of the recursive call. But there is no guarantee that this will succeed, because F' is declared within the scope of x. For example, if $\mathit{dneg} \cdot \langle F\ x\rangle$ returns $\langle \mathrm{eq}\ x\ x\rangle$, then $F' = (\lambda y : i.\, \mathrm{eq}\ x\ x)$ and $F' = (\lambda y : i.\, \mathrm{eq}\ y\ y)$ are among the possible solutions to this matching problem. To remedy this, F' can be declared outside of the scope of x, and thus could not possibly be instantiated with a term containing x:

$$\begin{aligned}\epsilon F : \mathrm{i} \to \mathrm{o}.\, \langle \mathrm{forall}\ F\rangle \mapsto\ & \epsilon F' : \mathrm{i} \to \mathrm{o}.\\ & \bar{\lambda} x : \mathrm{i}.\mathrm{case}\ \mathit{dneg} \cdot \langle F\ x\rangle\ \mathrm{of}\ \langle F'\ x\rangle \mapsto \langle \mathrm{neg}\ (\mathrm{neg}\ (\mathrm{forall}\ F'))\rangle\end{aligned}$$

In this case, the only solution to the matching problem is $F' = (\lambda y : i.\, \mathrm{eq}\ y\ y)$, which illustrates the necessity of explicit ϵ-declarations. However, we do not include $\bar{\lambda}$ in the ∇-calculus since, as we have seen, it allows us to write functions that let parameters escape their scope.

Instead, we do include two operators and one new type constructor that can be found in Figure 1. The operator ν is similar to $\bar{\lambda}$ in that it introduces new parameters, but different because it statically requires that these parameters cannot extrude their scope. The operator "pop" provides such guarantees. These guarantees are communicated through the type $\Box\tau$, which pop introduces and ν eliminates. The complete function *dneg* is given below.

$$
\begin{aligned}
&\mathit{dneg} \;:\; \langle \mathrm{o} \rangle \Rightarrow \langle \mathrm{o} \rangle \\
&\quad = \epsilon t_1 : \mathrm{i}.\, \epsilon t_2 : \mathrm{i}.\, \langle \mathrm{eq}\; t_1\; t_2 \rangle \mapsto \langle \mathrm{neg}\; (\mathrm{neg}\; (\mathrm{eq}\; t_1\; t_2)) \rangle \\
&\qquad \mid\; \epsilon F_1 : \mathrm{o}.\, \epsilon F_2 : \mathrm{o}. \\
&\qquad\quad \langle \mathrm{imp}\; F_1\; F_2 \rangle \mapsto \langle \mathrm{neg} \rangle \circ (\langle \mathrm{neg} \rangle \circ (\langle \mathrm{imp} \rangle \circ (\mathit{dneg} \cdot \langle F_1 \rangle) \circ (\mathit{dneg} \cdot \langle F_2 \rangle))) \\
&\qquad \mid\; \epsilon F : \mathrm{o}.\, \langle \mathrm{neg}\; F \rangle \mapsto \langle \mathrm{neg} \rangle \circ (\langle \mathrm{neg} \rangle \circ (\langle \mathrm{neg} \rangle \circ (\mathit{dneg} \cdot \langle F \rangle))) \\
&\qquad \mid\; \epsilon F : \mathrm{i} \to \mathrm{o}.\, \langle \mathrm{forall}\; F \rangle \mapsto \epsilon F' : \mathrm{i} \to \mathrm{o}. \\
&\qquad\quad \nu x : \mathrm{i}.\, \mathrm{case}\; \mathit{dneg} \cdot \langle F\; x \rangle\; \mathrm{of}\; \langle F'\; x \rangle \mapsto \mathrm{pop}\; \langle \mathrm{neg}\; (\mathrm{neg}\; (\mathrm{forall}\; F')) \rangle
\end{aligned}
$$

The body of the ν is of type $\Box\langle o \rangle$; the $\Box$ ensures that whatever value this expression evaluates to does not contain x. The body of pop has type $\langle o \rangle$ only because it neither contains x nor any ϵ-quantified variable the may depend on x. Thus, the subexpression "pop $\langle \mathrm{forall}\; F' \rangle$" introduces type $\Box\langle o \rangle$. A precise type theoretic definition and analysis of the $\Box$ type will be given in Section 4.

3.3 Pattern-matching Parameters

Finally, we turn to the last unexplained operator from Figure 1, the ∇-operator, which is used to match parameters introduced by ν.

Example 6 (Counting variable occurrences). Consider a function that counts the number of occurrences of bound variables in an untyped λ-expression from Example 4.

$$
\begin{aligned}
&\mathit{cntvar}\; (x) &&= (s\; z) \text{ where } x : \mathit{exp} \text{ is a parameter} \\
&\mathit{cntvar}\; (\mathit{app}\; e_1\; e_2) &&= \mathit{plus}\; (\mathit{cntvar}\; e_1)\; (\mathit{cntvar}\; e_2) \\
&\mathit{cntvar}\; (\mathit{lam}\; e) &&= \mathit{cntvar}\; (e\; x) \text{ for some new parameter } x : \mathit{exp}
\end{aligned}
$$

The first of the three cases corresponds to the parameter case that matches *any* parameter of type *exp* regardless of where and when it was introduced. Formally, we use the ∇-operator to implement this case.

$$
\begin{aligned}
&\mathit{cntvar} \;:\; \langle \mathrm{exp} \rangle \Rightarrow \langle \mathrm{nat} \rangle \\
&\quad = \nabla x : \mathrm{exp}.\, \langle x \rangle \mapsto \langle \mathrm{s\; z} \rangle \\
&\qquad \mid\; \epsilon e_1 : \mathrm{exp}.\, \epsilon e_2 : \mathrm{exp}. \\
&\qquad\quad \langle \mathrm{app}\; e_1\; e_2 \rangle \mapsto \mathit{plus} \cdot (\mathit{cntvar} \cdot \langle e_1 \rangle) \cdot (\mathit{cntvar} \cdot \langle e_2 \rangle) \\
&\qquad \mid\; \epsilon e : \mathrm{exp} \to \mathrm{exp}. \\
&\qquad\quad \langle \mathrm{lam}\; e \rangle \mapsto \epsilon n : \mathrm{nat}. \\
&\qquad\qquad \nu x : \mathrm{exp}. \\
&\qquad\qquad\quad (\langle n \rangle \mapsto \mathrm{pop}\; \langle n \rangle) \cdot (\mathit{cntvar} \cdot \langle e\; x \rangle)
\end{aligned}
$$

$\Box$

Notice that, in the above example, if we were to replace the ∇ with ϵ, it would still be possible for *cntvar* to return correct answers, since $\epsilon x : exp$ can match any expression of type *exp* including parameters; however, it would also be possible for *cntvar* to always return $\langle s\, z\rangle$ for the same reason.

Example 7 (Combinators). The combinators $c ::= \mathbf{S} \mid \mathbf{K} \mid \mathbf{MP}\ c_1\ c_2$ are represented as objects of type *comb* as follows:

$$\begin{array}{rlrl}
\ulcorner \mathbf{K} \urcorner &= K & K &: comb \\
\ulcorner \mathbf{S} \urcorner &= S & S &: comb \\
\ulcorner \mathbf{MP}\ c_1\ c_2 \urcorner &= MP \ulcorner c_1 \urcorner \ulcorner c_2 \urcorner & MP &: comb \to comb \to comb
\end{array}$$

Any simply-typed λ-expression from Example 4 can be converted into a combinator in a two-step algorithm. The first step is called bracket abstraction, or *ba*, which converts a parametric combinator (a representation-level function of type $comb \to comb$) into a combinator with one less parameter (of type *comb*). If M has type $comb \to comb$ and N has type *comb* then $\langle MP\rangle \circ (ba \cdot \langle M\rangle) \circ \langle N\rangle$ results in a term that is equivalent to $\langle M N\rangle$ in combinator logic.

$$\begin{array}{l}
ba\ (\lambda x : comb.\, x) = MP\ (MP\ S\ K)\ K \\
ba\ (\lambda x : comb.\, z) = MP\ K\ z \text{ where } z : comb \text{ is a parameter} \\
ba\ (\lambda x : comb.\, K) = MP\ K\ K \\
ba\ (\lambda x : comb.\, S) = MP\ K\ S \\
ba\ (\lambda x : comb.\, MP\ (c_1\ x)\ (c_2\ x)) = MP\ (MP\ S\ (ba\ c_1))\ (ba\ c_2)
\end{array}$$

$$\begin{array}{l}
ba\ :\ \langle \text{comb} \to \text{comb}\rangle \Rightarrow \langle \text{comb}\rangle \\
\quad = \langle \lambda x : \text{comb}.\, x\rangle \mapsto \langle \text{MP (MP S K) K}\rangle \\
\qquad \mid\ \nabla z : \text{comb}.\, \langle \lambda x : \text{comb}.\, z\rangle \mapsto \langle \text{MP K } z\rangle \\
\qquad \mid\ \langle \lambda x : \text{comb}.\, \text{K}\rangle \mapsto \langle \text{MP K K}\rangle \\
\qquad \mid\ \langle \lambda x : \text{comb}.\, \text{S}\rangle \mapsto \langle \text{MP K S}\rangle \\
\qquad \mid\ \epsilon c_1 : \text{comb} \to \text{comb}.\, \epsilon c_2 : \text{comb} \to \text{comb}. \\
\qquad\qquad \langle \lambda x : \text{comb}.\, \text{MP}\ (c_1\ x)\ (c_2\ x)\rangle \mapsto \\
\qquad\qquad\qquad \langle \text{MP}\rangle \circ (\langle \text{MP}\rangle \circ \langle \text{S}\rangle \circ (ba \cdot \langle c_1\rangle)) \circ (ba \cdot \langle c_2\rangle)
\end{array}$$

The first two cases of *ba* illustrate how to distinguish x, which is to be abstracted, from parameters that are introduced in the function *convert*, which we discuss next. The function *convert* traverses λ-expressions and uses *ba* to convert them into combinators.

$$\begin{array}{l}
convert\ (y\ z) = z \text{ where } y : comb \to exp \text{ and } z : comb \text{ are parameters} \\
convert\ (app\ e_1\ e_2) = MP\ (convert\ e_1)\ (convert\ e_2) \\
convert\ (lam\ e) = ba\ c \text{ where } c\ z = convert\ (e\ (y\ z)) \\
\qquad\qquad\qquad\qquad\quad \text{and } y : comb \to exp \\
\qquad\qquad\qquad\qquad\quad \text{and } z : comb \text{ are parameters}
\end{array}$$

The last case illustrates how a parameter of functional type may introduce information to be used when the parameter is matched. Rather than introduce a parameter x of type *exp*, we introduce a parameter of type $comb \rightarrow exp$ that carries a combinator as "payload." In our example, the payload is another parameter $z : comb$, the image of x under *convert*. This technique is applicable to a wide range of examples. We formalize *convert* below:

```
$convert\ :\ \langle \text{exp} \rangle \Rightarrow \langle \text{comb} \rangle$
          $= \nabla y : \text{comb} \rightarrow \text{exp}.\, \nabla z : \text{comb}.\, \langle y\ z \rangle \mapsto \langle z \rangle$
            $\mid\ \epsilon e_1 : \text{exp}.\, \epsilon e_2 : \text{exp}.$
                $\langle \text{app}\ e_1\ e_2 \rangle \mapsto \langle \text{MP} \rangle \circ (convert \cdot \langle e_1 \rangle) \circ (convert \cdot \langle e_2 \rangle)$
            $\mid\ \epsilon e : \text{exp} \rightarrow \text{exp}.\, \langle \text{lam}\ e \rangle \mapsto \epsilon c : \text{comb} \rightarrow \text{comb}.$
                $\nu y : \text{comb} \rightarrow \text{exp}.\, \nu z : \text{comb}.$
                  case $convert \cdot \langle e\ (y\ z) \rangle$ of $\langle c\ z \rangle \mapsto$ pop (pop $(ba \cdot \langle c \rangle)$)
```

□

We summarize a few of the most important properties of the ∇-operator. First, it is intuitively appealing to have one base case (the ∇-case) for each class of parameters, because what happens in these cases is uniquely defined in one place. Second, payload carrying parameters permit sophisticated base cases, which simplify the reading of a program because all information shared between the introduction and matching of parameters must be made explicit.

Example 8 (Counting abstractions). The function below counts the number of occurrences of λ-abstractions in an expression. It again provides one static case that matches any parameter of type *exp* and it has type $\langle exp \rangle \Rightarrow \langle nat \rangle$.

$$\begin{array}{ll} cntlam\ x & = z \\ & \qquad \text{where } x : exp \text{ is a parameter} \\ cntlam\ (app\ e_1\ e_2) & = plus\ (cntlam\ e_1)\ (cntlam\ e_2) \\ cntlam\ (lam\ e) & = s\ (cntlam\ (e\ x) \\ & \qquad \text{for some new parameter } x : exp) \end{array}$$

Its representation as an iteration follows the same ideas as in the example above.

```
$cntlam\ :\ \langle \text{exp} \rangle \Rightarrow \langle \text{nat} \rangle$
        $= \nabla x : \text{exp}.\, \langle x \rangle \mapsto \langle \text{z} \rangle$
          $\mid\ \epsilon e_1 : \text{exp}.$
              $\epsilon e_2 : \text{exp}.$
                $\langle \text{app}\ e_1\ e_2 \rangle \mapsto plus \cdot (cntlam \cdot \langle e_1 \rangle) \cdot (cntlam \cdot \langle e_2 \rangle)$
          $\mid\ \epsilon e : \text{exp} \rightarrow \text{exp}.$
              $\langle \text{lam}\ e \rangle \mapsto \epsilon n : \text{nat}.$
                $\nu p : \text{exp}.$
                  case $cntlam \cdot \langle e\ p \rangle$ of $\langle n \rangle \mapsto$ pop $\langle \text{s}\ n \rangle$
```

□

3.4 Pairs and Mutual Recursion

None of the examples presented so far are mutually recursive. Nevertheless, the ∇-calculus provides meta-level pairs and projections; this permits the formulation of mutually recursive functions by means of tupeling all mutually recursive parts into one recursive variable. Projections from this variable result then in the respective recursive calls.

Example 9 (Parallel reduction). Parallel reduction is here defined over expressions (from Example 4). We state the function first informally:

$$\begin{array}{ll}
\mathit{par}\, x & = x \\
& \quad \text{where } x : \mathit{exp} \text{ is a parameter} \\
\mathit{par}\,(\mathit{app}\, e_1\; e_2) & = \mathit{par}'\; e_1\; (\mathit{par}\, e_2) \\
\mathit{par}\,(\mathit{lam}\, e_1) & = \mathit{lam}\,(\lambda x : \mathit{exp}.\, \mathit{par}\,(e_1\; x) \\
& \quad \text{where } x : \mathit{exp} \text{ is a parameter}) \\
\mathit{par}'\; x\; e_2' & = \mathit{app}\; x\; e_2' \\
& \quad \text{where } x : \mathit{exp} \text{ is a parameter} \\
\mathit{par}'\; (\mathit{app}\, e_1\; e_2)\; e_2' & = \mathit{app}\; (\mathit{par}'\; e_1\; (\mathit{par}\, e_2))\; e_2' \\
\mathit{par}'\; (\mathit{lam}\, e_1)\; e_2' & = (\lambda x.\, \mathit{par}\,(e_1\; x) \\
& \quad \text{where } x : \mathit{exp} \text{ is a parameter})\; e_2'
\end{array}$$

The type of *par* is $\langle \mathit{exp}\rangle \Rightarrow \langle \mathit{exp}\rangle$; the auxiliary function par' has type $\langle \mathit{exp}\rangle \rightarrow \langle \mathit{exp}\rangle \Rightarrow \langle \mathit{exp}\rangle$. Mutually recursive functions are expressed in the ∇-calculus by meta-level pairing, and the individual parts as projections "fst *par*" and "snd *par*".

$$\begin{array}{l}
\mathit{par} \;:\; (\langle \text{exp}\rangle \Rightarrow \langle \text{exp}\rangle) \star (\langle \text{exp}\rangle \Rightarrow \langle \text{exp}\rangle \Rightarrow \langle \text{exp}\rangle) \\
\quad = \nabla x : \text{exp}.\, \langle x\rangle \mapsto \langle x\rangle \\
\qquad \mid\; \epsilon e_1 : \text{exp}. \\
\qquad\quad \epsilon e_2 : \text{exp}. \\
\qquad\qquad \langle \text{app } e_1\; e_2\rangle \mapsto (\text{snd } \mathit{par}) \cdot \langle e_1\rangle \cdot ((\text{fst } \mathit{par}) \cdot \langle e_2\rangle) \\
\qquad \mid\; \epsilon e_1 : \text{exp} \rightarrow \text{exp}. \\
\qquad\quad \langle \text{lam } e_1\rangle \mapsto \epsilon f_1 : \text{exp} \rightarrow \text{exp}. \\
\qquad\qquad \nu x : \text{exp}. \\
\qquad\qquad\quad \text{case } (\text{fst } \mathit{par}) \cdot \langle e_1\; x\rangle \text{ of } \langle f_1\; x\rangle \mapsto \text{pop } \langle \text{lam } f_1\rangle \\
\qquad +\; (\nabla x : \text{exp}.\, \epsilon e_2' : \text{exp}. \qquad\qquad\qquad\qquad\qquad\qquad) \\
\qquad\qquad \langle x\rangle \mapsto \langle e_2'\rangle \mapsto \langle \text{app } x\; e_2'\rangle \\
\qquad\quad \mid\; \epsilon e_1 : \text{exp}. \\
\qquad\qquad \epsilon e_2 : \text{exp}. \\
\qquad\qquad\quad \epsilon e_2' : \text{exp}. \\
\qquad\qquad\qquad \langle \text{app } e_1\; e_2\rangle \mapsto \langle e_2'\rangle \mapsto \langle \text{app}\rangle \circ ((\text{snd } \mathit{par}) \cdot \langle e_1\rangle \cdot ((\text{fst } \mathit{par}) \cdot \langle e_2\rangle)) \circ \langle e_2'\rangle \\
\qquad\quad \mid\; \epsilon e_1 : \text{exp} \rightarrow \text{exp}. \\
\qquad\qquad \epsilon e_2' : \text{exp}. \\
\qquad\qquad\quad \langle \text{lam } e_1\rangle \mapsto \langle e_2'\rangle \mapsto \epsilon f_1 : \text{exp} \rightarrow \text{exp}. \\
\qquad\qquad\qquad \nu x : \text{exp}. \\
\qquad\qquad\qquad\quad \text{case } (\text{fst } \mathit{par}) \cdot \langle e_1\; x\rangle \text{ of } \langle f_1\; x\rangle \mapsto \text{pop } \langle f_1\; e_2'\rangle
\end{array}$$

3.5 Higher-order Functions

Higher-order programming is also possible in the ∇-calculus. Consider an evaluator for our untyped λ-calculus extended by continuations as first-class values. We write κ for continuation parameters.

$$\text{Expressions } e ::= \ldots \mid \textbf{callcc}\ \kappa.\, e \mid \textbf{throw}\ \kappa\ e$$

Continuations parameters are represented as parameters of type *cont*.

$$\begin{array}{llll} \ulcorner \textbf{callcc}\ \kappa.\, e \urcorner = \mathit{callcc}\ (\lambda k : \mathit{cont}.\, \ulcorner e \urcorner) & & \mathit{callcc} & : (\mathit{cont} \to \mathit{exp}) \to \mathit{exp} \\ \ulcorner \textbf{throw}\ \kappa\ e \urcorner = \mathit{throw}\ \ulcorner e \urcorner \ulcorner \kappa \urcorner & & \mathit{app} & : \mathit{cont} \to (\mathit{exp} \to \mathit{exp}) \end{array}$$

Example 10. Let *ev* be a continuation passing evaluator for this language that we define as follows.

$$\begin{array}{ll} \mathit{ev}\ (\mathit{app}\ e_1\ e_2)\ K & = \mathit{ev}\ e_1\ (\lambda(\mathit{lam}\ (\lambda x : \mathit{exp}.\, e_1'\ x)).\ \mathit{ev}\ e_2\ (\lambda v_2 : \mathit{exp}.\ \mathit{ev}\ (e_1'\ v_2)\ K)) \\ \mathit{ev}\ (\mathit{lam}\ \lambda x : \mathit{exp}.\, e_1\ x)\ K & = K\ (\mathit{lam}\ \lambda x : \mathit{exp}.\, e_1\ x) \\ \mathit{ev}\ (\mathit{callcc}\ \lambda k : \mathit{cont}.\, e_1\ k)\ K & = \mathit{ev}\ (e_1\ k)\ K \\ & \quad \text{where } k : \mathit{cont} \text{ is a new continuation parameter} \\ & \quad \text{and } k \text{ is bound to } K \\ \mathit{ev}\ (\mathit{throw}\ k\ e_1)\ K & = K'\ e_1 \\ & \quad \text{if } k \text{ is bound to } K' \end{array}$$

The presentation of the *ev* in the ∇-calculus is not straightforward. A continuation is necessarily a meta-level function $\langle \mathit{exp} \rangle \Rightarrow \langle \mathit{exp} \rangle$, because continuations must consider cases of values. Consider for example the *app* case, where the outermost continuation on the lefthand-side of the equation pattern-matches against a λ-term, the result of evaluating e_1. Furthermore, substituting a meta-level continuation into an object-level parameter is impossible in the ∇-calculus because of the division into layers. Thus we introduce an environment $L \in \langle \mathit{cont} \rangle \Rightarrow (\langle \mathit{exp} \rangle \Rightarrow \langle \mathit{exp} \rangle)$ that maps continuation parameters to continuations. It is the first argument to *ev*, and initially empty. We write "*fail*", for a program that does not make progress, but gets stuck in the next step.

```
$ev\ :\ \langle\text{exp}\rangle \Rightarrow ((\langle\text{exp}\rangle \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle)) \Rightarrow \langle\text{exp}\rangle) \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle) \Rightarrow \langle\text{exp}\rangle$
    $= \epsilon e_1 : \text{exp}.$
        $\epsilon e_2 : \text{exp}.$
          $\langle\text{app}\ e_1\ e_2\rangle \mapsto \epsilon k : \langle\text{exp}\rangle \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle)) \Rightarrow \langle\text{exp}\rangle.\ k \mapsto ev \cdot \langle e_1\rangle \cdot \epsilon x_1 : \text{exp} \to \text{exp}.$
            $\langle\text{lam}\ x_1\rangle \mapsto ev \cdot \langle e_2\rangle \cdot \epsilon x_2 : \text{exp}.$
              $\langle x_2\rangle \mapsto ev \cdot \langle x_1\ x_2\rangle \cdot k$
      $\mid\ \epsilon e_1 : \text{exp} \to \text{exp}.$
          $\langle\text{lam}\ e_1\rangle \mapsto \epsilon k : \langle\text{exp}\rangle \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle)) \Rightarrow \langle\text{exp}\rangle.\ k \mapsto k \cdot \langle\text{lam}\ e_1\rangle$
      $\mid\ \epsilon e_1 : \text{cont} \to \text{exp}.$
          $\langle\text{callcc}\ e_1\rangle \mapsto \epsilon k : \langle\text{exp}\rangle \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle)) \Rightarrow \langle\text{exp}\rangle.\ k \mapsto$
            $\epsilon l : \langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle.\ l \mapsto \epsilon v : \langle\text{exp}\rangle.\ \nu k' : \text{cont}.$
              $\text{case}\ ev \cdot \langle e_1\ k'\rangle \cdot k \cdot (l \mid \langle k'\rangle \mapsto \epsilon x : \text{exp}.\ \langle x\rangle \mapsto k \cdot \langle x\rangle \cdot l)\ \text{of}\ v \mapsto \text{pop}\ v$
      $\mid\ \nabla k : \text{cont}.\ \epsilon e_1 : \text{exp}.$
          $\langle\text{throw}\ k\ e_1\rangle \mapsto \epsilon k' : \langle\text{exp}\rangle \Rightarrow (\langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle)) \Rightarrow \langle\text{exp}\rangle.\ k' \mapsto ev \cdot \langle e_1\rangle \cdot \epsilon x : \text{exp}.$
            $\langle x\rangle \mapsto \epsilon l : \langle\text{cont}\rangle \Rightarrow \langle\text{exp}\rangle \Rightarrow \langle\text{exp}\rangle.\ l \mapsto l \cdot \langle k\rangle \cdot \langle x\rangle$
```

□

This technique of handling and computing with meta-level functions has a second interesting application. As second example, consider the usual higher-order encoding of the untyped λ-calculus from Example 4, an arbitrary domain type D, and two functions $g : D \to D \to D$, and $h : (D \to D) \to D$.

Example 11 (Iteration). In the ∇-calculus we define iteration over higher-order encodings that replaces all constants *app* by g and all constants *lam* by g before reducing the term via a helper function f'. f' takes two arguments, an environment k of type $\langle exp \rangle \Rightarrow D$ and e of type $\langle exp \rangle$. Let $i = \epsilon x : exp. \langle x \rangle \mapsto \text{empty}$ be the initial environment.

$$
\begin{array}{l}
f' \;:\; (\langle \text{exp} \rangle \Rightarrow D) \Rightarrow \langle \text{exp} \rangle \Rightarrow \langle D \rangle \\
\quad = \epsilon k \in (\text{exp} \Rightarrow D).\, k \mapsto \\
\qquad\quad (\nabla x : \text{exp}.\, \langle x \rangle \mapsto k \cdot \langle x \rangle \\
\qquad\quad \mid\; \epsilon e_1 : \text{exp}. \\
\qquad\qquad \epsilon e_2 : \text{exp}. \\
\qquad\qquad\quad \langle \text{app } e_1\ e_2 \rangle \mapsto g \cdot (f' \cdot k \cdot \langle e_1 \rangle) \cdot (f' \cdot k \cdot \langle e_2 \rangle) \\
\qquad\quad \mid\; \epsilon e : \text{exp} \to \text{exp}. \\
\qquad\qquad \langle \text{lam } e \rangle \mapsto h \cdot (\epsilon u \in D.\, u \mapsto \epsilon v \in D. \\
\qquad\qquad\qquad\qquad\qquad \nu x : \text{exp}. \\
\qquad\qquad\qquad\qquad\qquad\quad (\langle v \rangle \mapsto \text{pop}\langle v \rangle) \cdot (f' \cdot (k \mid \langle x \rangle \mapsto u) \cdot \langle e\ x \rangle))) \\
f \;\;:\; \langle \text{exp} \rangle \Rightarrow D \\
\quad = \epsilon e : \text{exp}.\, \langle e \rangle \mapsto f' \cdot i \cdot \langle e \rangle
\end{array}
$$

3.6 More Examples

Below we present more examples. We start with an implementation of the conjunction function on booleans, and the "identity test" on parametric λ-calculus expressions, both of which are used in subsequent examples.

Example 12 (Conjunction). Computing the conjunction of two Booleans value illustrates the use of variables in patterns. Informally we implement the Boolean operation *and* as follows.

$$
\begin{array}{l}
\textit{and true } b\ = b \\
\textit{and false } b = \textit{false}
\end{array}
$$

A formal representation of *and* in the ∇-calculus is then as follows:

$$
\begin{array}{l}
\textit{and} \;:\; \langle \text{bool} \rangle \Rightarrow \langle \text{bool} \rangle \Rightarrow \langle \text{bool} \rangle \\
\qquad = \epsilon b : \text{bool}. \\
\qquad\qquad \langle \text{true} \rangle \mapsto \langle b \rangle \mapsto \langle b \rangle \\
\qquad\quad \mid\; \epsilon b : \text{bool}. \\
\qquad\qquad \langle \text{false} \rangle \mapsto \langle b \rangle \mapsto \langle \text{false} \rangle
\end{array}
$$

□

Example 13 (Identity test). Below is a function which decides if a parametric function mapping *exp* to *exp* is the identity function or not. The function has type $\langle exp \to exp \rangle \Rightarrow \langle bool \rangle$. This example, along with Example 15 will be used by Example 16

$$\begin{array}{l} idtest\,(\lambda x : exp.\, app\ (E_1\ x)\ (E_2\ x)) = false \\ idtest\,(\lambda x : exp.\, lam\ \lambda y : exp.\, E\ x\ y) = false \\ idtest\,(\lambda x : exp.\, x) = true \end{array}$$

The identity test function is hence represented in the ∇-calculus as follows.

$$\begin{array}{rl} idtest & : \langle \mathrm{exp} \to \mathrm{exp} \rangle \Rightarrow \langle \mathrm{bool} \rangle \\ & = \langle \lambda x : \mathrm{exp}.\, x \rangle \mapsto \langle \mathrm{true} \rangle \\ & \quad \mid\ \epsilon e_1 : \mathrm{exp} \to \mathrm{exp}. \\ & \qquad\quad \epsilon e_2 : \mathrm{exp} \to \mathrm{exp}. \\ & \qquad\qquad \langle \lambda x : \mathrm{exp}.\, \mathrm{app}\ (e_1\ x)\ (e_2\ x) \rangle \mapsto \langle \mathrm{false} \rangle \\ & \quad \mid\ \epsilon e : \mathrm{exp} \to \mathrm{exp} \to \mathrm{exp}. \\ & \qquad\quad \langle \lambda x : \mathrm{exp}.\, \mathrm{lam}\ \lambda y : \mathrm{exp}.\, e\ x\ y \rangle \mapsto \langle \mathrm{false} \rangle \end{array}$$

□

Functions in the ∇-calculus may be nested. As example, consider a decision procedure that tests if an untyped λ-expression is a β-redex. To remind the reader β-reduction are defined as follows.

$$\beta\text{-reduction: } (\lambda x.\, E_1)\ E_2 \rightsquigarrow [E_2/x](E_1)$$

$(\lambda x.\, E_1)\ E_2$ is called a β redex.

Example 14 (β-redex test). The β-redex test function has type $\langle exp \rangle \Rightarrow \langle bool \rangle$ and can informally be defined as follows.

$$\begin{array}{l} betatest\ F = \text{case } f \text{ of } (lam\ f) \mapsto false \\ \qquad\qquad\qquad \mid (app\ E_1\ E_2) \mapsto\ (\text{case } E_1 \text{ of } (lam\ E') \mapsto true \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid (app\ E_1'\ E_2') \mapsto false) \end{array}$$

Its representation in our calculus is:

$$\begin{array}{rl} betatest & : \langle \mathrm{exp} \rangle \Rightarrow \langle \mathrm{bool} \rangle \\ & = \epsilon f : \mathrm{exp} \to \mathrm{exp}. \\ & \qquad \langle \mathrm{lam}\ f \rangle \mapsto \langle \mathrm{false} \rangle \\ & \quad \mid\ \epsilon e_1 : \mathrm{exp}. \\ & \qquad\quad \epsilon e_2 : \mathrm{exp}. \\ & \qquad\qquad \langle \mathrm{app}\ e_1\ e_2 \rangle \mapsto \mathrm{case}\ \langle e_1 \rangle\ \mathrm{of}\ \epsilon f : \mathrm{exp} \to \mathrm{exp}. \\ & \qquad\qquad\qquad\qquad\qquad\qquad \langle \mathrm{lam}\ f \rangle \mapsto \langle \mathrm{true} \rangle \\ & \qquad\qquad\qquad\qquad\qquad \mid\ \epsilon e_3 : \mathrm{exp}. \\ & \qquad\qquad\qquad\qquad\qquad\qquad \epsilon e_4 : \mathrm{exp}. \\ & \qquad\qquad\qquad\qquad\qquad\qquad\quad \langle \mathrm{app}\ e_3\ e_4 \rangle \mapsto \langle \mathrm{false} \rangle \end{array}$$

□

The following example will be useful later on.

Example 15 (Constant test). Below we define a function which returns true if a given object of type $exp \to exp$ (see Example 4) is constant with respect to the first argument (e.g. the given expression-with-a-hole doesn't use its hole). This function is used along with Example 13 in Example 16.

$$\begin{aligned}
&\mathit{const}\ (\lambda x : \mathit{exp}.\, y) = \mathit{true}\\
&\qquad\qquad \text{where } y : \mathit{exp} \text{ is a parameter}\\
&\mathit{const}\ (\lambda x : \mathit{exp}.\, \mathit{app}\ (E_1\ x)\ (E_2\ x))\\
&\qquad = \mathit{and}\ (\mathit{const}\ (\lambda x : \mathit{exp}.\, E_1\ x))\ (\mathit{const}\ (\lambda x : \mathit{exp}.\, E_2\ x))\\
&\mathit{const}\ (\lambda x : \mathit{exp}.\, \mathit{lam}\ (\lambda y : \mathit{exp}.\, E\ x\ y))\\
&\qquad = \mathit{const}\ (\lambda x : \mathit{exp}.\, E\ x\ y)\\
&\qquad\qquad \text{for some new parameter } y : \mathit{exp}\\
&\mathit{const}\ (\lambda x : \mathit{exp}.\, x) = \mathit{false}
\end{aligned}$$

The representation of *const* has type $\langle \mathit{exp} \to \mathit{exp}\rangle \Rightarrow \langle \mathit{bool}\rangle$.

$$\begin{aligned}
&\mathit{const}\ :\ \langle \text{exp} \to \text{exp}\rangle \Rightarrow \langle \text{bool}\rangle\\
&\quad = \nabla y : \text{exp}.\ \langle \lambda x : \text{exp}.\, y\rangle \mapsto \langle \text{true}\rangle\\
&\qquad \mid\ \epsilon e_1 : \text{exp} \to \text{exp}.\\
&\qquad\quad \epsilon e_2 : \text{exp} \to \text{exp}.\\
&\qquad\qquad \langle \lambda x : \text{exp}.\, \text{app}\ (e_1\ x)\ (e_2\ x)\rangle \mapsto \mathit{and} \cdot (\mathit{const} \cdot \langle \lambda x : \text{exp}.\, e_1\ x\rangle) \cdot (\mathit{const} \cdot \langle \lambda x : \text{exp}.\, e_2\ x\rangle)\\
&\qquad \mid\ \epsilon e : \text{exp} \to \text{exp} \to \text{exp}.\\
&\qquad\quad \langle \lambda x : \text{exp}.\, \text{lam}\ \lambda y : \text{exp}.\, e\ x\ y\rangle \mapsto \epsilon b : \text{bool}.\\
&\qquad\qquad \nu y : \text{exp}.\\
&\qquad\qquad\quad \text{case } \mathit{const} \cdot \langle \lambda x : \text{exp}.\, e\ x\ y\rangle \text{ of } \langle b\rangle \mapsto \text{pop } \langle b\rangle\\
&\qquad \mid\ \langle \lambda x : \text{exp}.\, x\rangle \mapsto \langle \text{false}\rangle
\end{aligned}$$

□

In example 14 we demonstrated a function that tests whether a given expression in the untyped λ-calculus starts with a β redex. In the next example, we use examples 13 and 15 to help write a function that tests whether a given expression in the untyped λ-calculus starts with an η-redex. To remind the reader, η-reduction is defined as follows.

$$\eta\text{-reduction: } \lambda x.\, (E\ x) \rightsquigarrow E \quad \text{where } x \text{ does not occur free in } E$$

$\lambda x.\, (E\ x)$ is called an η-redex if x is not free in E.

Example 16 (η-redex test). The function that determines if a given expression is an η-redex is more difficult to define than the function that determines if a given expression is a β-redex. This is because the function must take into account not only the structure of the given expression, but a side condition as well. This can be accomplished using the functions *idtest* (from Example 13) and *const* (from Example 15) defined above.

$$
\begin{aligned}
&\mathit{etatest}\, F = \text{case } F \text{ of}\\
&\qquad (\mathit{lam}\ E) \mapsto\ \text{case } E \text{ of } \lambda x : \mathit{exp}.\,(\mathit{lam}\ \lambda y : \mathit{exp}.\, E'\ x\ y) \mapsto \mathit{false}\\
&\qquad\qquad\qquad \mid \lambda x : \mathit{exp}.\,(\mathit{app}\ (E'_1\, x)\ (E'_2\, x)) \mapsto\\
&\qquad\qquad\qquad\quad \mathit{and}\ (\mathit{const}\ E'_1)\ (\mathit{idtest}\ E'_2)\\
&\qquad\qquad\qquad \mid \lambda x : \mathit{exp}.\, x \mapsto \mathit{false}\\
&\qquad \mid (\mathit{app}\ E_1\ E_2) \mapsto \mathit{false}
\end{aligned}
$$

Its representation in our calculus is:

$$
\begin{aligned}
&\mathit{etatest}\ :\ \langle \text{exp}\rangle \Rightarrow \langle\text{bool}\rangle\\
&\quad = \epsilon e : \text{exp} \to \text{exp}.\\
&\qquad \langle \text{lam } e\rangle \mapsto \text{case } \langle e\rangle \text{ of } \epsilon e' : \text{exp} \to \text{exp} \to \text{exp}.\\
&\qquad\qquad\qquad \langle \lambda x : \text{exp}.\, \text{lam } \lambda y : \text{exp}.\, e'\ x\ y\rangle \mapsto \langle\text{false}\rangle\\
&\qquad\qquad\quad \mid\ \epsilon e'_1 : \text{exp} \to \text{exp}.\\
&\qquad\qquad\qquad \epsilon e'_2 : \text{exp} \to \text{exp}.\\
&\qquad\qquad\qquad\quad \langle \lambda x : \text{exp}.\, \text{app } (e'_1\ x)\ (e'_2\ x)\rangle \mapsto \mathit{and} \cdot (\mathit{const} \cdot \langle e'_1\rangle) \cdot (\mathit{idtest} \cdot \langle e'_2\rangle)\\
&\qquad\qquad\quad \mid\ \langle \lambda x : \text{exp}.\, x\rangle \mapsto \langle\text{false}\rangle\\
&\qquad \mid\ \epsilon e_1 : \text{exp}.\\
&\qquad\quad \epsilon e_2 : \text{exp}.\\
&\qquad\qquad \langle \text{app } e_1\ e_2\rangle \mapsto \langle\text{false}\rangle
\end{aligned}
$$

□

Example 17 (Translation to de Bruijn representation). Untyped λ-expressions in de Bruijn form $d ::= n \mid \mathbf{lam}\ d \mid d_1\ \mathbf{@}\ d_2$ are represented as canonical objects of type *db* over the signature which includes the natural numbers and the following declarations.

$$
\begin{array}{rcl@{\qquad}l}
\ulcorner n \urcorner &=& \mathit{var}\, \ulcorner n\urcorner & \mathit{var} : \mathit{nat} \to \mathit{db}\\
\ulcorner \mathbf{lam}\ d\urcorner &=& \mathit{lm}\, \ulcorner d\urcorner & \mathit{lm}\ \ : \mathit{db} \to \mathit{db}\\
\ulcorner d_1\ \mathbf{@}\ d_2\urcorner &=& \mathit{ap}\, \ulcorner d_1\urcorner\, \ulcorner d_2\urcorner & \mathit{ap}\ \ : \mathit{db} \to \mathit{db} \to \mathit{db}
\end{array}
$$

A translation from the higher-order representation to de Bruijn form has type $\langle \mathit{exp}\rangle \Rightarrow \langle \mathit{db}\rangle$ and is represented formally in terms of an auxiliary function *trans* of type $\langle \mathit{exp}\rangle \Rightarrow \langle \mathit{nat}\rangle \Rightarrow \langle \mathit{db}\rangle$, which keeps track of the number of *lambda*-abstractions the current subexpression is under. These functions can be written informally as follows:

$$
\begin{array}{ll}
\mathit{trans}\, (x\ m)\ n & = \mathit{var}\, (\mathit{minus}\ m\ n)\\
 & \qquad \text{where } x : \mathit{nat} \to \mathit{exp} \text{ is a parameter}\\
\mathit{trans}\, (\mathit{lam}\, e)\ n & = \mathit{lm}\, (\mathit{trans}\, (e\ (x\ n))(s\ n)\\
 & \qquad \text{for some new parameter } x : \mathit{nat} \to \mathit{exp})\\
\mathit{trans}\, (\mathit{app}\, e_1\ e_2)\ n & = \mathit{ap}\, (\mathit{trans}\, e_1 n)\ (\mathit{trans}\, e_2\ n)\\
\mathit{dbtrans}\, e & = \mathit{trans}\ e\ z
\end{array}
$$

The translation function is expressed in the ∇-calculus as follows.

$$\begin{aligned}
\mathit{trans} &: \langle \mathrm{exp}\rangle \Rightarrow \langle \mathrm{nat}\rangle \Rightarrow \langle \mathrm{db}\rangle\\
&= \nabla x : \mathrm{nat} \to \mathrm{exp}.\, \epsilon n : \mathrm{nat}.\\
&\qquad \langle x\ n\rangle \mapsto \epsilon m : \mathrm{nat}.\\
&\qquad\quad \langle m\rangle \mapsto \langle \mathrm{var}\rangle \circ (\mathit{minus} \cdot \langle m\rangle \cdot \langle n\rangle)\\
&\quad\ \mid\ \epsilon e_1 : \mathrm{exp}.\\
&\qquad\quad \epsilon e_2 : \mathrm{exp}.\\
&\qquad\qquad \langle \mathrm{app}\ e_1\ e_2\rangle \mapsto \epsilon n : \mathrm{nat}.\\
&\qquad\qquad\quad \langle n\rangle \mapsto \langle \mathrm{ap}\rangle \circ (\mathit{trans} \cdot \langle e_1\rangle \cdot \langle n\rangle) \circ (\mathit{trans} \cdot \langle e_2\rangle \cdot \langle n\rangle)\\
&\quad\ \mid\ \epsilon e : \mathrm{exp} \to \mathrm{exp}.\\
&\qquad\quad \langle \mathrm{lam}\ e\rangle \mapsto \epsilon n : \mathrm{nat}.\\
&\qquad\qquad \langle n\rangle \mapsto \epsilon d : \mathrm{db}.\\
&\qquad\qquad\quad \nu x : \mathrm{nat} \to \mathrm{exp}.\\
&\qquad\qquad\qquad \mathrm{case}\ \mathit{trans} \cdot \langle e\ (x\ n)\rangle \cdot \langle \mathrm{s}\ n\rangle\ \mathrm{of}\ \langle d\rangle \mapsto \mathrm{pop}\ \langle d\rangle
\end{aligned}$$

In order to define *dbtrans* we can simply instantiate *trans*'s second argument with z to obtain a function of type $\langle exp\rangle \Rightarrow \langle db\rangle$.

$$\begin{aligned}
\mathit{dbtrans} &: \langle \mathrm{exp}\rangle \Rightarrow \langle \mathrm{db}\rangle\\
&= \epsilon x : \mathrm{exp}.\\
&\qquad \langle x\rangle \mapsto \mathit{trans} \cdot \langle x\rangle \cdot \langle \mathrm{z}\rangle
\end{aligned}$$

□

4 Semantics

The operators ν and pop have guided the design of the static and operational semantics of the ∇-calculus. To reiterate, once a parameter is introduced by a ν, all other declarations that take place within its scope may depend on the new parameter. As we will see, pop statically ensures that an expression is valid outside ν's scope by discarding all declarations since the last parameter introduction in a manner reminiscent of popping elements off a stack. The ambient environment is therefore formally captured in form of *scope stacks*. A scope consists of two parts: The context Γ (defined in Section 2), which summarizes all object-level declarations $x : A$, and the context Φ, which summarizes all meta-level declarations $u \in \tau$.

$$\begin{aligned}
&\text{Meta Contexts: } \Phi ::= \cdot \mid \Phi, u \in \tau\\
&\text{Scope Stacks: } \quad \Omega ::= \cdot \mid \Omega, (\Gamma; \Phi)
\end{aligned}$$

We refer to the top and second-from-top elements of Ω as the *current* and *previous* scopes, respectively. The scope stack Ω grows monotonically, which means that the current scope always extends the previous scope.

$$\frac{\Gamma \vdash A : \mathit{type}}{\Omega, (\Gamma; \Phi) \vdash \langle A \rangle : \mathit{type}}\ \mathsf{wfinj} \qquad \frac{\Omega \vdash \tau : \mathit{type} \quad \Omega \vdash \sigma : \mathit{type}}{\Omega \vdash \tau \Rightarrow \sigma : \mathit{type}}\ \mathsf{wffun}$$

$$\frac{\Omega \vdash \tau : \mathit{type} \quad \Omega \vdash \sigma : \mathit{type}}{\Omega \vdash \tau \star \sigma : \mathit{type}}\ \mathsf{wfprod} \qquad \frac{\Omega \vdash \tau : \mathit{type}}{\Omega, (\Gamma; \Phi) \vdash \Box\tau : \mathit{type}}\ \mathsf{wfbox}$$

Fig. 2. Well-formed types in the ∇-calculus

4.1 Valid types

Validity of a type depends not only on its formation, but on the scope stack it is postulated to exist in. For example, $\langle A \rangle$ is a valid type in any scope because every type A in the simply-typed λ-calculus is well-formed. However, the type $\Box\tau$ is only well-formed in a scope when τ is valid in the preceding scope. For example, in $\cdot, (\cdot; \cdot), \Box\tau$ is not a valid type. We have summarized the rules defining the judgment $\Omega \vdash \tau : \mathit{type}$ in Figure 2.

4.2 Static semantics

We define the typing judgment $\Omega \vdash e \in \tau$ by the rules depicted in Figure 3. Many of the rules are self-explanatory. All rules except for tpnew and tppop touch only the current scope. For example, tpvar relates variables and types, whereas tpinj enforces that only representation-level objects valid in the current scope can be lifted to the computation-level. For functions, the pattern must be of the argument type, whereas the body be of the result type. Variables that may occur in patterns must be declared b y a preceding $\epsilon x : A$ or $\epsilon u \in \tau$ declaration, which will be recorded in the current scope by tptheobj and tpthemeta, respectively. The rules tpapp, tpalt, and tpfix are standard. The tppop rule is the introduction rule for $\Box\tau$. The expression pop e is valid if e is valid in the previous scope. The corresponding elimination rule is tpnew. The expression $\nu x : A. e$ has type τ when e is of type $\Box\tau$ in the properly extended scope stack.

4.3 Operational Semantics

Computation level function application in the ∇-calculus is more demanding than the usual substitution of an argument for a free variable. It relies on the proper instantiation of all ϵ- and ∇-bound variables that occur in the function's pattern. Perhaps not surprisingly, the behavior of our calculus depends on when these instantiations are committed. For example,

$$(\epsilon f \in \langle nat \rangle \rightarrow \langle nat \rangle. f \mapsto plus \cdot (f \cdot \langle z \rangle) \cdot (f \cdot \langle s\ z \rangle)) \cdot (\epsilon n : nat. \langle n \rangle \mapsto \langle n \rangle)$$

may either return $s\ z$ under a call-by-name semantics, or no solution at all under a call-by-value semantics because $n : nat$ may be instantiated either by z or

$$\frac{\Phi(u) = \tau}{\Omega, (\Gamma; \Phi) \vdash u \in \tau}\,\mathsf{tpvar} \qquad \frac{\Gamma \vdash M : A}{\Omega, (\Gamma; \Phi) \vdash \langle M \rangle \in \langle A \rangle}\,\mathsf{tpinj}$$

$$\frac{\Omega \vdash e_1 \in \tau \qquad \Omega \vdash e_2 \in \sigma}{\Omega \vdash e_1 \mapsto_\tau e_2 \in \tau \to \sigma}\,\mathsf{tpfun} \qquad \frac{\Omega, (\Gamma; \Phi, u \in \tau) \vdash e \in \tau}{\Omega, (\Gamma; \Phi) \vdash \mathrm{fix}\ u \in \tau.\, e \in \tau}\,\mathsf{tpfix}$$

$$\frac{\Omega, (\Gamma, x : A; \Phi) \vdash e \in \tau}{\Omega, (\Gamma; \Phi) \vdash \epsilon x : A.\, e \in \tau}\,\mathsf{tptheobj} \qquad \frac{\Omega, (\Gamma; \Phi, u \in \tau) \vdash e \in \tau}{\Omega, (\Gamma; \Phi) \vdash \epsilon u \in \tau.\, e \in \tau}\,\mathsf{tpthemeta}$$

$$\frac{\Omega \vdash e_1 \in \sigma \to \tau \qquad \Omega \vdash e_2 \in \sigma}{\Omega \vdash e_1 \cdot e_2 \in \tau}\,\mathsf{tpapp} \qquad \frac{\Omega \vdash e_1 \in \tau \qquad \Omega \vdash e_2 \in \tau}{\Omega \vdash (e_1 \mid e_2) \in \tau}\,\mathsf{tpalt}$$

$$\frac{\Omega \vdash e_1 \in \tau_1 \qquad \Omega \vdash e_2 \in \tau_2}{\Omega \vdash (e_1, e_2) \in \tau_1 \star \tau_2}\,\mathsf{tppair} \qquad \frac{\Omega \vdash e \in \tau_1 \star \tau_2}{\Omega \vdash \mathrm{fst}\ e \in \tau_1}\,\mathsf{tpfst} \qquad \frac{\Omega \vdash e \in \tau_1 \star \tau_2}{\Omega \vdash \mathrm{snd}\ e \in \tau_2}\,\mathsf{tpsnd}$$

$$\frac{\Omega \vdash e \in \tau}{\Omega, (\Gamma; \Phi) \vdash \mathrm{pop}\ e \in \Box\tau}\,\mathsf{tppop} \qquad \frac{\Omega, (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash e \in \Box\tau}{\Omega, (\Gamma; \Phi) \vdash \nu x : A.\, e \in \tau}\,\mathsf{tpnew}$$

$$\frac{\Omega, (\Gamma, x : A; \Phi) \vdash e \in \tau}{\Omega, (\Gamma; \Phi) \vdash \nabla x : A.\, e \in \tau}\,\mathsf{tpnabla} \qquad \frac{\Omega \vdash \sigma \to \tau : \mathit{type}}{\Omega \vdash \mathrm{empty} \in \sigma \to \tau}\,\mathsf{tpempty}$$

Fig. 3. The static semantics of the ∇-calculus

$s\ z$ but not both. Consequently, our calculus adopts a call-by-name evaluation strategy. We can define computational-level λ-abstraction "lambda $u \in \tau.\, e$" as syntactic sugar for $(\epsilon u \in \tau.\, u \mapsto e)$ and "let $u \in \tau = e_1$ in e_2 end" as syntactic sugar for $((\epsilon u \in \tau.\, u \mapsto e_2)\ e_1)$.

Definition 2 (Values). *The set of values of the ∇-calculus is defined as follows.*

$$\textit{Values: } v ::= \langle M \rangle \mid (e_1, e_2) \mid \mathrm{pop}\ e \mid e_1 \mapsto_\tau e_2$$

The operational semantics of the ∇-calculus combines a system of reduction rules of the form $\Omega \vdash e \to e'$ with an equivalence relation on meta-level expressions $\Omega \vdash e \equiv e' \in \tau$. We give the reduction rules in Figure 4 and the equality rules in Figure 5. During runtime, all ϵ-quantified variables are instantiated with concrete objects, so evaluation always takes place in a scope stack of the form $\Omega ::= \cdot \mid \Omega, (\Gamma; \cdot)$, where Γ contains only ν-quantified parameter declarations.

The rules in Figure 4 are organized into three parts. The top part shows the essential reduction rules redbeta and rednupop. The rule rednupop states that it is unnecessary to traverse into a new scope to return an expression that is valid in the previous scope.

Among the second block of rules, redalt_1 and redalt_2 express a non-deterministic choice in the control flow. Similarly, redsome and redsomeM express a non-deterministic choice of instantiations. The abbreviations f/u and M/x stand for

$$\frac{\varOmega \vdash e_1 \equiv e_1' \in \tau}{\varOmega \vdash (e_1 \mapsto_\tau e_2) \cdot e_1' \rightarrow e_2}\,\mathsf{redbeta} \qquad \frac{}{\varOmega \vdash \nu x : A.\, \mathrm{pop}\; e \rightarrow e}\,\mathsf{rednupop}$$

$$\frac{}{\varOmega \vdash (e_1 \mid e_2) \rightarrow e_1}\,\mathsf{redalt}_1 \qquad \frac{}{\varOmega \vdash (e_1 \mid e_2) \rightarrow e_2}\,\mathsf{redalt}_2$$

$$\frac{\varGamma \vdash M : A}{\varOmega, (\varGamma; \cdot) \vdash \epsilon x : A.\, e \rightarrow [M/x]e}\,\mathsf{redsome} \qquad \frac{\varOmega \vdash f \in \tau}{\varOmega \vdash \epsilon u \in \tau.\, e \rightarrow [f/u]e}\,\mathsf{redsomeM}$$

$$\frac{\varGamma(y) = A}{\varOmega, (\varGamma; \cdot) \vdash \nabla x : A.\, e \rightarrow [y/x]e}\,\mathsf{rednabla} \qquad \frac{}{\varOmega \vdash \mathrm{fix}\; u \in \tau.\, e \rightarrow [\mathrm{fix}\; u \in \tau.\, e/u]e}\,\mathsf{redfix}$$

$$\frac{\varOmega \vdash e_1 \rightarrow e_1'}{\varOmega \vdash e_1 \cdot e_2 \rightarrow e_1' \cdot e_2}\,\mathsf{redfun} \qquad \frac{\varOmega, (\varGamma; \cdot), (\varGamma, x : A; \cdot) \vdash e \rightarrow e'}{\varOmega, (\varGamma; \cdot) \vdash \nu x : A.\, e \rightarrow \nu x : A.\, e'}\,\mathsf{rednew}$$

Fig. 4. Small-step semantics (Reductions)

single-point substitutions that can easily be expanded into simultaneous substitutions given in Definition 3. During evaluation, the current scope only contains parameters introduced by ν, and thus rednabla expresses a non-deterministic choice of parameters. Finally, redfix implements the unrolling of the recursion operator.

The bottom two rules are necessary to give us a congruence closure for reductions on ∇-expressions. Because the ∇-calculus is call-by-name, we do not evaluate e_2 in the rule redfun. Finally, rednew reduces under the ν after appropriately copying and extending the current scope.

Thus, equivalence on functions is decided only by syntactic equality, as shown by rule eqfun in Figure 5. For all other types, we give three rules: the rule ending in V refers to the case where the left and right hand side are already values, while the rules ending in L or R are used when further reduction steps are required on the left or right side of the equality, respectively.

5 Meta Theory

We study the meta-theory of the ∇-calculus culminating in the type-preservation theorem, which entails that parameters cannot escape their scope.

Substituting for ϵ and ∇-bound variables is essential for defining the operational meaning of our expressions. In this section we elaborate on representation-level and computation-level substitutions, as well as substitution stacks, which are defined on scope stacks.

$$\frac{\Gamma \vdash M \equiv N : A}{\Omega, (\Gamma; \cdot) \vdash \langle M \rangle \equiv \langle N \rangle \in \langle A \rangle}\ \mathsf{eqinjV}$$

$$\frac{\Omega \vdash e_1 \to e_1' \quad \Omega \vdash e_1' \equiv e_2 \in \langle A \rangle}{\Omega \vdash e_1 \equiv e_2 \in \langle A \rangle}\ \mathsf{eqinjL} \qquad \frac{\Omega \vdash e_2 \to e_2' \quad \Omega \vdash e_1 \equiv e_2' \in \langle A \rangle}{\Omega \vdash e_1 \equiv e_2 \in \langle A \rangle}\ \mathsf{eqinjR}$$

$$\frac{\Omega \vdash e_1 \equiv e_1' \in \tau_1 \quad \Omega \vdash e_2 \equiv e_2' \in \tau_2}{\Omega \vdash (e_1, e_2) \equiv (e_1', e_2') \in \tau_1 \star \tau_2}\ \mathsf{eqpairV}$$

$$\frac{\Omega \vdash e_1 \to e_1' \quad \Omega \vdash e_1' \equiv e_2 \in \tau_1 \star \tau_2}{\Omega \vdash e_1 \equiv e_2 \in \tau_1 \star \tau_2}\ \mathsf{eqpairL} \qquad \frac{\Omega \vdash e_2 \to e_2' \quad \Omega \vdash e_1 \equiv e_2' \in \tau_1 \star \tau_2}{\Omega \vdash e_1 \equiv e_2 \in \tau_1 \star \tau_2}\ \mathsf{eqpairR}$$

$$\frac{\Omega \vdash e_1 \equiv e_2 \in \tau}{\Omega, (\Gamma; \cdot) \vdash \mathrm{pop}\ e_1 \equiv \mathrm{pop}\ e_2 \in \Box\tau}\ \mathsf{eqpopV}$$

$$\frac{\Omega \vdash e_1 \to e_1' \quad \Omega \vdash e_1' \equiv e_2 \in \Box\tau}{\Omega \vdash e_1 \equiv e_2 \in \Box\tau}\ \mathsf{eqpopL} \qquad \frac{\Omega \vdash e_2 \to e_2' \quad \Omega \vdash e_1 \equiv e_2' \in \Box\tau}{\Omega \vdash e_1 \equiv e_2 \in \Box\tau}\ \mathsf{eqpopR}$$

$$\frac{}{\Omega \vdash e \equiv e \in \tau_1 \Rightarrow \tau_2}\ \mathsf{eqfun}$$

Fig. 5. Small-step semantics (Equality)

Definition 3 (Substitutions).

$$\begin{array}{ll}
\textit{Representation-Level Substitutions:} & \gamma ::= \cdot \mid \gamma, M/x \\
\textit{Computation-Level Substitutions:} & \varphi ::= \cdot \mid \varphi, f/u \\
\textit{Substitution Stacks:} & \omega ::= \cdot \mid \omega, (\gamma; \varphi)
\end{array}$$

Representation-level substitutions are conventional, whereas computation-level substitutions and substitution stacks require some explanation. Their definitions can be motivated by the same intuition that explains computation-level contexts and scope stacks. Computation-level contexts are responsible only for identifying the types of free computation-level variables, whereas the responsibility for typing more complex terms (e.g. those involving pop and ν) falls upon scope stacks. Similarly, computation-level substitutions are responsible only for acting upon free computation-level variables, whereas the responsibility for performing substitutions on more complex terms falls upon substitution stacks. Note that, despite the fact that meta-level substitutions only act on meta-level variables, such an action can result in an arbitrary meta-level term. This term's validity can only be determined by a full scope stack. Hence, the co-domain of a computation-level substitution is a scope stack, which is reflected in Figure 6.

Figure 6 defines the typing judgments for substitutions: $\Gamma \vdash \gamma : \Gamma'$, $\Omega \vdash \phi : \Phi$, and $\Omega \vdash \omega \in \Omega'$. The domains of the substitutions are Γ', Φ, and Ω', respec-

$$\frac{}{\Gamma \vdash \cdot : \cdot}\ \mathsf{tpEObjS} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash \gamma : \Gamma'}{\Gamma \vdash (\gamma, M/x) : (\Gamma', x : A)}\ \mathsf{tpIObS}$$

$$\frac{}{\Omega \vdash \cdot : \cdot}\ \mathsf{tpEMetaS} \qquad \frac{\Omega \vdash f \in \tau \quad \Omega \vdash \varphi : \Phi}{\Omega \vdash (\varphi, f/u) : (\Phi, u \in \tau)}\ \mathsf{tpIMetaS}$$

$$\frac{}{\Omega \vdash \cdot : \cdot}\ \mathsf{tpEStackS} \qquad \frac{\Gamma \vdash \gamma : \Gamma' \quad \Omega, (\Gamma; \Phi) \vdash \varphi : \Phi' \quad \Omega \vdash \omega : \Omega'}{\Omega, (\Gamma; \Phi) \vdash \omega, (\gamma; \varphi) : \Omega', (\Gamma', \Phi')}\ \mathsf{tpIStackS}$$

Fig. 6. The static semantics of substitutions

tively, and the codomains of the substitutions are Γ, Ω, and Ω, respectively. The definition of substitution application is given in Figure 7.

$$\begin{aligned}
[\gamma, M/x]x &= M \\
[\gamma, M/x]y &= [\gamma]y \\
[\gamma]c &= c \\
[\gamma](N_1\ N_2) &= ([\gamma]N_1)\ ([\gamma]N_2) \\
[\gamma](\lambda x : A.\, N) &= \lambda x : A.\, [\gamma, x/x]N \\
\\
[\varphi, e/u]u &= e \\
[\varphi, e/u]v &= [\varphi]v
\end{aligned}$$

$$\begin{aligned}
[\omega, (\gamma; \varphi)]u &= [\varphi]u \\
[\omega, (\gamma; \varphi)]\langle N \rangle &= \langle [\gamma]N \rangle \\
[\omega, (\gamma; \varphi)](\text{pop } e) &= \text{pop } [\omega]e \\
[\omega](e_1 \mapsto_\tau e_2) &= ([\omega]e_1) \mapsto_\tau ([\omega]e_2) \\
[\omega](e_1 \cdot e_2) &= ([\omega]e_1) \cdot ([\omega]e_2) \\
[\omega](e_1 \mid e_2) &= ([\omega]e_1 \mid [\omega]e_2) \\
[\omega](e_1, e_2) &= ([\omega]e_1, [\omega]e_2) \\
[\omega](\text{fst } e) &= \text{fst } [\omega]e \\
[\omega](\text{snd } e) &= \text{snd } [\omega]e \\
[\omega](\text{empty}) &= \text{empty} \\
[\omega, (\gamma; \varphi)](\text{fix } u \in \tau.\, e) &= \text{fix } u \in \tau.\, [\omega, (\gamma; \varphi, u'/u)]e \\
[\omega, (\gamma; \varphi)](\epsilon x : A.\, e) &= \epsilon x : A.\, [\omega, (\gamma, x'/x; \varphi)]e \\
[\omega, (\gamma; \varphi)](\epsilon u \in \tau.\, e) &= \epsilon u \in \tau.\, [\omega, (\gamma; \varphi, u'/u)]e \\
[\omega, (\gamma; \varphi)](\nabla x : A.\, e) &= \nabla x : A.\, [\omega, (\gamma, x'/x; \varphi)]e \\
[\omega, (\gamma; \varphi)](\nu x : A.\, e) &= \nu x : A.\, [\omega, (\gamma; \varphi), (\gamma, x'/x; \varphi)]e
\end{aligned}$$

for fresh u' and x' in the above

Fig. 7. Substitution Application

Recall that the rules defining the operational semantics summarized in Figures 4 and 5 used an intuitive notion of single point substitutions. They can be seen as shorthands for notion of substitutions, and be directly expanded by the use of identity substitutions.

Definition 4 (Identity Substitutions).

Representation Level	*Computation Level*	*Scope Stacks*
$id_{\cdot} = \cdot$	$id_{\cdot} = \cdot$	$id_{\cdot} = \cdot$
$id_{\Gamma,x:A} = id_{\Gamma}, x/x$	$id_{\Phi,u\in\tau} = id_{\Phi}, u/u$	$id_{\Omega,(\Gamma;\Phi)} = id_{\Omega}, (id_{\Gamma}; id_{\Phi})$

Definition 5 (Single Point Substitutions).

Representation level: *If $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : A$ we typically use the abbreviation $[N/x]M$ to stand for the term $[id_{\Gamma}, N/x]M$; note that $\Gamma \vdash (id_{\Gamma}, N/x) : (\Gamma, x : A)$.*

Computation level: *If $\Omega; (\Gamma, x : A; \Phi) \vdash e : \tau$ and $\Gamma \vdash N : A$ we typically use the abbreviation $[N/x]e$ to stand for the term $[id_{\Omega}, (id_{\Gamma}, N/x; id_{\Phi})]e$; note that $\Omega, (\Gamma; \Phi) \vdash id_{\Omega}, (id_{\Gamma}, N/x; id_{\Phi}) \in \Omega; (\Gamma, x : A; \Phi)$.*

Similarly, if $\Omega; (\Gamma; \Phi, u \in \tau) \vdash e : \tau'$ and $\Omega; (\Gamma; \Phi) \vdash f : \tau$ we typically use the abbreviation $[f/u]e$ to stand for the term $\vdash [id_{\Omega}, (id_{\Gamma}; id_{\Phi}, f/u)]e$; note that $\Omega, (\Gamma; \Phi) \vdash id_{\Omega}, (id_{\Gamma}; id_{\Phi}, f/u) \in \Omega, (\Gamma, x : A; \Phi, u \in \tau)$.

We will find the usual object-level substitution lemma useful later on.

Lemma 1 (Object-Level Substitution).

1. *If $\Gamma \vdash M : A$ and $\Gamma' \vdash \gamma : \Gamma$ then $\Gamma' \vdash [\gamma]M : A$*
2. *If $\Gamma \vdash A :$ type and $\Gamma' \vdash \gamma : \Gamma$ then $\Gamma' \vdash A :$ type*

Proof. The first part is proven by induction on the structure of $\Gamma \vdash M : A$, where the case for variables is proven by induction on the structure of $\Gamma' \vdash \gamma : \Gamma$. The second part is proven by straightforward induction on the structure of $\Gamma \vdash A :$ *type* (note that, because we don't have dependencies, this part is trivially true, since the only thing that influences the validity of a type A is the signature Σ, which is presumed to be fixed). $\square$

The following lemma is useful for proving several of the lemmas in this section.

Lemma 2 (The Empty Scope Stack is Useless). *If $\Omega \vdash e \in \tau$ then $\Omega = \Omega', (\Gamma; \Phi)$.*

Proof. By straightforward induction over the structure of $\Omega \vdash e \in \tau$

We find it useful to define ordering relations on contexts and on scope stacks:

Definition 6 (Size Orderings). *We define $<$ and $\leq$ for Γ, Φ and Ω as follows: where $\Gamma < \Gamma'$ and $\Phi \leq \Phi'$ are defined by:*

$$\frac{}{\Gamma < \Gamma, x : a} \qquad \frac{\Gamma < \Gamma'}{\Gamma < \Gamma', x : A} \qquad \frac{\Gamma < \Gamma'}{\Gamma, x : A < \Gamma', x : A}$$

$$\frac{}{\Gamma \leq \Gamma} \qquad \frac{\Gamma \leq \Gamma'}{\Gamma \leq \Gamma', x : A} \qquad \frac{\Gamma \leq \Gamma'}{\Gamma, x : A \leq \Gamma', x : A}$$

$$\frac{}{\Phi \leq \Phi} \qquad \frac{\Phi \leq \Phi'}{\Phi \leq \Phi', u \in \tau} \qquad \frac{\Phi \leq \Phi'}{\Phi, x : A \leq \Phi', x : A}$$

$$\frac{}{\cdot \leq \Omega} \qquad \frac{\Omega \leq \Omega' \quad \Gamma \leq \Gamma' \quad \Phi \leq \Phi'}{\Omega, (\Gamma; \Phi) \leq \Omega', (\Gamma'; \Phi')}$$

Note that, for all of the above syntactic categories, $<$ and $\leq$ are both transitive, and $\Gamma < \Gamma'$ implies $\Gamma \leq \Gamma'$ (by straightforward inductions).

Although most expressions can be considered well-typed in any number of scope stacks (e.g. if $\Gamma \vdash M : A$ then $\Omega, (\Gamma; \Phi) \vdash \langle M \rangle \in \langle A \rangle$ for any Ω and any Φ), intuitively we only care about scope stacks that occur during the type-checking of a closed expression in a scope stack which consists of single world on top of the empty stack. Such stacks are characterized by our definition of well-formedness.

Definition 7 (Well-Formed Scope Stacks). *We say that a context stack Ω is well-formed if the proposition $\vdash \Omega$ ok can be proved using the following judgments:*

$$\frac{}{\vdash \cdot \ \mathsf{ok}}\,\mathsf{okempty} \qquad \frac{}{\vdash \cdot, (\Gamma; \Phi)\ \mathsf{ok}}\,\mathsf{okinit} \qquad \frac{\vdash \Omega, (\Gamma; \Phi)\ \mathsf{ok} \quad \Gamma < \Gamma' \quad \Phi \leq \Phi'}{\vdash \Omega, (\Gamma; \Phi), (\Gamma'; \Phi')\ \mathsf{ok}}\,\mathsf{oknew}$$

At first glance, it may seem strange to consider $\cdot$ to be an ok context-stack, since we've already proven that the scope stack alone cannot be used to successfully typecheck any ∇-calculus term. However, the following lemma, which is useful for proving both substitution and subject reduction, would not be true were this not the case.

Lemma 3. *If* $\vdash \Omega, (\Gamma; \Phi)$ ok *then* $\vdash \Omega$ ok

Proof. By cases.

The following can be thought of as a weakening lemma for the ok judgment.

Lemma 4. *If* $\vdash \Omega, (\Gamma; \Phi)$ ok *and* $\Gamma \leq \Gamma'$ *and* $\Phi \leq \Phi'$ *then* $\Omega, (\Gamma'; \Phi')$ ok *and* $\Omega, (\Gamma, \Phi) \leq \Omega, (\Gamma'; \Phi')$

Proof. By cases on $\vdash \Omega, (\Gamma; \Phi)$ ok, using transitivity.

We find it useful to define a general purpose weakening lemma for terms and substitutions on all levels.

Lemma 5 (Weakening).

1. *If* $\Gamma \vdash M : A$ *and* $\Gamma \leq \Gamma'$ *then* $\Gamma' \vdash M : A$.
2. *If* $\Phi(u) = \tau$ *and* $\Phi \leq \Phi'$ *then* $\Phi'(u) = \tau$.
3. *If* $\Omega \vdash e \in \tau$ *and* $\Omega \leq \Omega'$ *then* $\Omega' \vdash e \in \tau$.
4. *If* $\Gamma \vdash \gamma : \Gamma'$ *and* $\Gamma \leq \Gamma''$ *then* $\Gamma'' \vdash \gamma : \Gamma'$.
5. *If* $\Omega \vdash \varphi \in \Phi$ *and* $\Omega \leq \Omega'$ *then* $\Omega' \vdash \varphi \in \Phi$.
6. *If* $\Omega \vdash \omega : \Omega'$ *and* $\Omega \leq \Omega''$ *then* $\Omega'' \vdash \omega : \Omega$.

Proof. 1 by straightforward induction on the given typing derivations and 2 by straightforward induction on the definition of $\Phi \leq \Phi'$. 3 by straightforward induction on the given typing derivation, using 1 and 2. 4 by straightforward induction on the given typing derivation, using 1. 5 by straightforward induction on the given typing derivation, using 3. 6 can be proven directly, using 4 and 5. It is worth noting that the proof of part 3 is the only place we use any of the size ordering rules in the rightmost column.

The following lemma is useful for allowing us to apply weakening.

Lemma 6. *If* $\vdash \Omega$ ok *and* $\vdash \Omega, (\Gamma; \Phi)$ ok *then* $\Omega \leq \Omega, (\Gamma; \Phi)$.

Proof. By straightforward induction over the structure of $\vdash \Omega$ ok.

The following lemma is used in proving the meta-variable case for the general substitution lemma which follows.

Lemma 7 (Meta-Variable Subs.). *If* $\Phi(u) = \tau$ *and* $\Omega \vdash \varphi : \Phi$ *then* $\Omega \vdash [\varphi]u \in \tau$.

Proof. By induction on the structure of $\Omega \vdash \varphi : \Phi$. Note that we consider two separate instances of $\mathsf{tpIMetaS}$: when the top variable being substituted is u and when the top variable being substituted is distinct from u.

Case:

$$\mathcal{D} = \frac{}{\Omega \vdash \cdot : \cdot}\,\mathsf{tpEMetaS}$$

$\cdot(u) \neq \tau$ for any u and for any τ, and thus the theorem is trivially true.

$$\mathcal{D} = \frac{\Omega \vdash f \in \tau \quad \Omega \vdash \varphi : \Phi}{\Omega \vdash (\varphi, f/u) : (\Phi, u \in \tau)}\,\mathsf{tpIMetaS}$$

$\Omega \vdash f \in \tau$	by this case
$(\Phi, u \in \tau)(u) = \tau$	by definition
$[\varphi, f/u]u = f$	by def of subs
$\Omega \vdash [\varphi, f/u]u \in \tau$	by the above.

$$\mathcal{D} = \cfrac{\Omega \vdash f \in \sigma \quad \overset{\mathcal{D}_1}{\Omega \vdash \varphi : \Phi}}{\Omega \vdash (\varphi, f/v) : (\Phi, v \in \sigma)} \;\textsf{tpIMetaS}$$

$(\Phi, v \in \sigma)(u) = \Phi(u)$ by definition
$[\varphi, f/v]u = [\varphi]u$ by def of subs
If $\Phi(u) = \tau$ then $\Omega \vdash [\varphi]u \in \tau$ by the IH on $\mathcal{D}_1$
If $(\Phi, v \in \sigma)(u) = \tau$ then $\Omega \vdash [\varphi, f/v]u \in \tau$
by the above

The following is the key lemma of this section.

Lemma 8 (Substitution).

1. *If $\Omega \vdash e \in \tau$, Ω $\textsf{ok}$, Ω' $\textsf{ok}$ and $\Omega' \vdash \omega \in \Omega$, then $\Omega' \vdash [\omega]e \in \tau$*
2. *If $\Omega \vdash \tau : type$, Ω $\textsf{ok}$, Ω' $\textsf{ok}$ and $\Omega' \vdash \omega \in \Omega$, then $\Omega' \vdash \tau : type$*

Proof. By induction on the structure of $\Omega \vdash e \in \tau$.
Case:

$$\mathcal{D} = \cfrac{\Phi(u) = \tau}{\Omega, (\Gamma; \Phi) \vdash u \in \tau} \;\textsf{tpvar}$$

$\vdash \Omega, (\Gamma; \Phi)$ $\textsf{ok}$
$\vdash \Omega'$ $\textsf{ok}$
$\Omega' \vdash \omega : \Omega, (\Gamma, \Phi)$ given
$\omega = \omega', (\gamma, \varphi)$
$\Omega' \vdash \varphi : \Phi$ by inversion on the above
$\Phi(u) = \tau$ by this case
$\Omega' \vdash [\varphi]u \in \tau$ by lemma 7
$[\omega', (\gamma; \varphi)]u = [\varphi]u$ by definition
$\Omega' \vdash [\omega]u \in \tau$ by the above

Case:

$$\mathcal{D} = \cfrac{\Gamma \vdash M : A}{\Omega, (\Gamma; \Phi) \vdash \langle M \rangle \in \langle A \rangle} \;\textsf{tpinj}$$

$\vdash \Omega, (\Gamma; \Phi)$ $\textsf{ok}$
$\vdash \Omega'$ $\textsf{ok}$
$\Omega' \vdash \omega : \Omega, (\Gamma, \Phi)$ given
$\omega = \omega', (\gamma, \varphi)$
$\Omega' = \Omega'', (\Gamma', \Phi')$
$\Gamma' \vdash \gamma : \Gamma$ by inversion on the above
$[\omega', (\gamma; \varphi)]M = [\gamma]M$ by def of subs
$\Gamma' \vdash [\gamma]M : A$ by object substitution
$\Omega'', (\Gamma', \Phi) \vdash \langle [\gamma]M \rangle : \langle A \rangle$ by rule
$\Omega' \vdash [\omega]\langle M \rangle : \langle A \rangle$ by the above

Case:

$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Omega \vdash e_1 \in \tau_1 & \Omega \vdash e_2 \in \tau_2\end{array}}{\Omega \vdash (e_1, e_2) \in \tau_1 \star \tau_2}\,\textsf{tppair}$$

$\vdash \Omega$ ok	
$\vdash \Omega'$ ok	
$\Omega' \vdash \omega : \Omega$	given
$\Omega' \vdash [\omega]e_1 \in \tau_1$	by the IH on $\mathcal{D}_1$
$\Omega' \vdash [\omega]e_2 \in \tau_2$	by the IH on $\mathcal{D}_2$
$\Omega' \vdash ([\omega]e_1, [\omega]e_2) \in \tau_1 \star \tau_2$	by rule
$\Omega' \vdash [\omega](e_1, e_2) \in \tau_1 \star \tau_2$	by def of subs

Case:

$$\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e \in \tau_1 \star \tau_2\end{array}}{\Omega \vdash \text{fst } e \in \tau_1}\,\textsf{tpfst}$$

$\vdash \Omega$ ok	
$\vdash \Omega'$ ok	
$\Omega' \vdash \omega : \Omega$	given
$\Omega' \vdash [\omega]e \in \tau_1 \star \tau_2$	by the IH on $\mathcal{D}_1$
$\Omega' \vdash \text{fst } [\omega]e \in \tau_1$	by rule
$\Omega' \vdash [\omega]\text{fst } e \in \tau$	by def of subs

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e \in \tau_1 \star \tau_2\end{array}}{\Omega \vdash \text{snd } e \in \tau_2}\,\textsf{tpsnd}$$

See tpfst

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e \in \tau\end{array}}{\Omega, (\Gamma; \Phi) \vdash \text{pop } e \in \Box\tau}\,\textsf{tppop}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok	
$\vdash \Omega'$ ok	
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$	given
$\omega = \omega', (\gamma, \varphi)$	
$\Omega' = \Omega'', (\Gamma', \Phi')$	
$\Omega'' \vdash \omega' : \Omega$	by inversion on the above
$\vdash \Omega$ ok	by lemma 3

$\vdash \Omega''$ ok — by lemma 3
$\Omega'' \vdash [\omega']e \in \tau$ — by the IH on $\mathcal{D}_1$
$\Omega' \vdash \text{pop } [\omega']e \in \tau$ — by rule
$\Omega' \vdash [\omega]\text{pop } e \in \tau$ — by def of subs

Case:

$$\mathcal{D} = \cfrac{\Omega \vdash e_1 \in \tau \quad \Omega \vdash e_2 \in \sigma}{\Omega \vdash e_1 \mapsto_\tau e_2 \in \tau \to \sigma}\,\mathsf{tpfun}$$

See tppair

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega \vdash e_1 \in \sigma \to \tau\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ \Omega \vdash e_2 \in \sigma\end{matrix}}{\Omega \vdash e_1 \cdot e_2 \in \tau}\,\mathsf{tpapp}$$

See tppair

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega \vdash e_1 \in \tau\end{matrix} \quad \begin{matrix}\mathcal{D}_2 \\ \Omega \vdash e_2 \in \tau\end{matrix}}{\Omega \vdash (e_1 \mid e_2) \in \tau}\,\mathsf{tpalt}$$

See tppair

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega, (\Gamma; \Phi, u \in \tau) \vdash e \in \tau\end{matrix}}{\Omega, (\Gamma; \Phi) \vdash \text{fix } u \in \tau.\, e \in \tau}\,\mathsf{tpfix}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok
$\vdash \Omega'$ ok
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$ — given
$\omega = \omega', (\gamma, \varphi)$
$\Omega' = \Omega'', (\Gamma'; \Phi')$
$\Gamma' \vdash \gamma : \Gamma$
$\Omega'', (\Gamma'; \Phi') \vdash \varphi : \Phi$
$\Omega'' \vdash \omega' : \Omega$ — by inversion on the above
$\Gamma' \leq \Gamma'$ and $\Phi' \leq \Phi', u \in \tau$ — by definition
$\vdash \Omega, (\Gamma; \Phi, u \in \tau)$ ok — by lemma 4
$\Gamma' \leq \Gamma'$ and $\Phi' \leq \Phi', u' \in \tau$ — by definition
$\vdash \Omega'', (\Gamma'; \Phi', u' \in \tau)$ ok
$\Omega'', (\Gamma'; \Phi') \leq \Omega'', (\Gamma'; \Phi', u \in \tau)$ — by lemma 4
$\Omega'', (\Gamma'; \Phi', u' \in \tau) \vdash u' \in \tau$ — by rule
$\Omega'', (\Gamma'; \Phi', u' \in \tau) \vdash \varphi \in \Phi$ — by weakening
$\Omega'', (\Gamma'; \Phi', u' \in \tau) \vdash (\varphi, u'/u) \in (\Phi, u \in \tau)$ — by rule
$\Omega'', (\Gamma'; \Phi', u' \in \tau) \vdash \omega', (\gamma; \varphi, u'/u) \in \Omega, (\Gamma; \Phi, u \in \tau)$ — by rule

$\Omega'', (\Gamma'; \Phi', u' \in \tau) \vdash [\omega', (\gamma; \varphi, u'/u)]e \in \tau$ by the IH on $\mathcal{D}_1$
$\Omega'', (\Gamma'; \Phi') \vdash \text{fix } u \in \tau.\, ([\omega', (\gamma; \varphi, u'/u)]e) \in \tau$ by rule
$\Omega'', (\Gamma'; \Phi') \vdash [\omega', (\gamma; \varphi)](\text{fix } u \in \tau.\, e \in \tau)$ by def of subs
$\Omega' \vdash [\omega](\text{fix } u \in \tau.\, e \in \tau)$ by equality

Case:

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Omega, (\Gamma, x : A; \Phi) \vdash e \in \tau}}{\Omega, (\Gamma; \Phi) \vdash \epsilon x : A.\, e \in \tau}\,\mathsf{tptheobj}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok
$\vdash \Omega'$ ok
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$ given
$\omega = \omega', (\gamma, \varphi)$
$\Omega' = \Omega'', (\Gamma'; \Phi')$
$\Gamma' \vdash \gamma : \Gamma$
$\Omega'', (\Gamma'; \Phi') \vdash \varphi : \Phi$
$\Omega'' \vdash \omega' : \Omega$ by inversion on the above
$\Gamma \leq \Gamma, x : A$ and $\Phi \leq \Phi$ by definition
$\vdash \Omega, (\Gamma, x : A; \Phi)$ ok by lemma 4
$\Gamma' \leq \Gamma', x' : A$ and $\Phi' \leq \Phi'$ by definition
$\vdash \Omega', (\Gamma', x' : A; \Phi')$ ok
$\Omega', (\Gamma'; \Phi') \leq \Omega', (\Gamma', x' : A; \Phi')$ by lemma 4
$\Gamma', x' : A \vdash \gamma : \Gamma$ by weakening
$\Gamma', x' : A \vdash x' : A$ by rule
$\Gamma', x' : A \vdash (\gamma, x'/x) : (\Gamma, x : A)$ by rule
$\Omega'', (\Gamma', x' : A; \Phi') \vdash \varphi : \Phi$ by weakening
$\Omega'', (\Gamma', x' : A; \Phi') \vdash \omega', (\gamma, x'/x; \varphi) \in \Omega, (\Gamma, x : A; \Phi)$ by rule
$\Omega'', (\Gamma', x' : A; \Phi') \vdash [\omega', (\gamma, x'/x; \varphi)]e \in \tau$ by the IH on $\mathcal{D}_1$
$\Omega'', (\Gamma'; \Phi') \vdash \epsilon x : A.\, ([\omega', (\gamma, x'/x; \varphi)]e) \in \tau$ by the IH on $\mathcal{D}_1$
$\Omega'', (\Gamma'; \Phi') \vdash [\omega', (\gamma; \varphi)](\epsilon x : A.\, e) \in \tau$ by def of subs
$\Omega' \vdash [\omega](\epsilon x : A.\, e) \in \tau$ by equality

Case:

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Omega, (\Gamma; \Phi, u \in \tau) \vdash e \in \tau}}{\Omega, (\Gamma; \Phi) \vdash \epsilon u \in \tau.\, e \in \tau}\,\mathsf{tpthemeta}$$

See tpfix

Case:

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Omega, (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash e \in \Box\tau}}{\Omega, (\Gamma; \Phi) \vdash \nu x : A.\, e \in \tau}\,\mathsf{tpnew}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok	
$\vdash \Omega'$ ok	
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$	given
$\omega = \omega', (\gamma, \varphi)$	
$\Omega' = \Omega'', (\Gamma'; \Phi')$	
$\Gamma' \vdash \gamma : \Gamma$	
$\Omega'', (\Gamma'; \Phi') \vdash \varphi : \Phi$	
$\Omega'' \vdash \omega' : \Omega$	by inversion on the above
$\Gamma < \Gamma, x : A$ and $\Phi \leq \Phi$	by rule
$\vdash \Omega, (\Gamma; \Phi), (\Gamma, x : A, \Phi)$ ok	by rule
$\Gamma' < \Gamma', x' : A$ and $\Phi' \leq \Phi'$	by rule
$\vdash \Omega', (\Gamma', x' : A, \Phi')$ ok	by rule
$\Omega' \leq \Omega', (\Gamma', x' : A; \Phi')$	by lemma 6
$\Omega', (\Gamma', x' : A; \Phi') \vdash \varphi : \Phi$	by weakening
$\Gamma', x : A \vdash x : A$	by rule
$\Gamma', x' : A \vdash \gamma : \Gamma$	by weakening
$\Gamma', x' : A \vdash (\gamma, x'/x) : \Gamma, x : A$	by rule
$\Omega', (\Gamma', x' : A; \Phi') \vdash \omega, (\gamma, x'/x; \varphi) : \Omega, (\Gamma; \Phi), (\Gamma, x : A; \Phi)$	by rule
$\Omega', (\Gamma', x' : A; \Phi') \vdash [\omega, (\gamma, x'/x; \varphi)]e : \Box\tau$	by the IH on $\mathcal{D}_1$
$\Omega' \vdash \nu x : A. ([\omega, (\gamma, x'/x; \varphi)]e) : \tau$	by rule
$\Omega' \vdash [\omega](\nu x : A. .e) \in \tau$	by def of subs

Case:

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Omega, (\Gamma, x : A; \Phi) \vdash e \in \tau}}{\Omega, (\Gamma; \Phi) \vdash \nabla x : A. e \in \tau}\ \textsf{tpnabla}$$

See tptheobj

Case:

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Omega \vdash \sigma \to \tau : \mathit{type}}}{\Omega \vdash \text{empty} \in \sigma \to \tau}\ \textsf{tpempty}$$

Ω ok	
Ω' ok	
$\Omega \vdash \omega : \Omega'$	given
$\Omega' \vdash \sigma \to \tau : \mathit{type}$	by the IH on $\mathcal{D}_1$
$\Omega' \vdash \text{empty} : \mathit{type}$	by rule

Case:

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Gamma \vdash A : \mathit{type}}}{\Omega, (\Gamma; \Phi) \vdash \langle A \rangle : \mathit{type}}\ \textsf{wfinj}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok
$\vdash \Omega'$ ok
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$ given
$\omega = \omega', (\gamma'; \phi')$
$\Gamma' \vdash \gamma' : \Gamma$ by — by inversion on the above
$\Gamma' \vdash A : \mathit{type}$ — by the object substitution lemma (Lemma 1)
$\Omega' \vdash \langle A \rangle : \mathit{type}$ — by rule

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega \vdash \tau : \mathit{type}\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Omega \vdash \sigma : \mathit{type}\end{matrix}}{\Omega \vdash \tau \Rightarrow \sigma : \mathit{type}}\ \mathsf{wffun}$$

$\vdash \Omega$ ok
$\vdash \Omega'$ ok
$\Omega' \vdash \omega : \Omega$ — given
$\Omega' \vdash \tau : \mathit{type}$ — by the IH on $\mathcal{D}_1$
$\Omega' \vdash \sigma : \mathit{type}$ — by the IH on $\mathcal{D}_2$
$\Omega' \vdash \tau \to \sigma : \mathit{type}$ — by rule

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega \vdash \tau : \mathit{type}\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Omega \vdash \sigma : \mathit{type}\end{matrix}}{\Omega \vdash \tau \star \sigma : \mathit{type}}\ \mathsf{wfprod}$$

See wffun

Case:

$$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Omega \vdash \tau : \mathit{type}\end{matrix}}{\Omega, (\Gamma; \Phi) \vdash \Box\tau : \mathit{type}}\ \mathsf{wfbox}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok
$\vdash \Omega'$ ok
$\Omega' \vdash \omega : \Omega, (\Gamma; \Phi)$ — given
$\Omega' = \Omega'', (\Gamma'; \Phi')$
$\omega = \omega', (\Gamma'; \Phi')$ — by inversion on the above
$\vdash \Omega$ ok — by Lemma 3
$\vdash \Omega''$ ok — by Lemma 3
$\Omega' \vdash \tau : \mathit{type}$ — by the IH on $\mathcal{D}_1$
$\Omega', (\Gamma'; \Phi') \vdash \tau : \mathit{type}$ — by rule

We are now ready to prove our theorem.

Theorem 2 (Type Preservation). *If* $\vdash \Omega$ ok *and* $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \to e'$ *then* $\Omega \vdash e' \in \tau$

Proof. By induction on the structure of $\Omega \vdash e \to e'$.

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e \to e'\end{array}}{\Omega \vdash \text{fst } e \to \text{fst } e'}$$

$\vdash \Omega$ ok	
$\Omega \vdash \text{fst } e \in \tau$	by assumption
$\Omega \vdash e \in \tau \star \tau'$	by inversion
$\Omega \vdash e' \in \tau \star \tau'$	by the IH on $\mathcal{D}_1$
$\Omega \vdash \text{fst } e' \in \tau$	by rule

Case:

$$\mathcal{D} = \dfrac{}{\Omega \vdash \text{fst } (e_1, e_2) \to e_1}$$

$\vdash \Omega$ ok	
$\Omega \vdash \text{fst } (e_1, e_2) \in \tau$	by assumption
$\Omega \vdash (e_1, e_2) \in \tau \star \tau'$	by inversion
$\Omega \vdash e_1 \in \tau'$	by inversion

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e \to e'\end{array}}{\Omega \vdash \text{snd } e \to \text{snd } e'}$$

See the equivalent rule for fst .

Case:

$$\mathcal{D} = \dfrac{}{\Omega \vdash \text{snd } (e_1, e_2) \to e_2}$$

See the equivalent rule for snd .

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e_1 \to e_1'\end{array}}{\Omega \vdash e_1 \cdot e_2 \to e_1' \cdot e_2}$$

$\vdash \Omega$ ok	
$\Omega \vdash e_1 \cdot e_2 \in \tau$	by assumption
$\Omega \vdash e_1 \in \tau' \Rightarrow \tau$	
$\Omega \vdash e_2 \in \tau'$	by inversion
$\Omega \vdash e_1' \in \tau' \Rightarrow \tau$	by the IH on $\mathcal{D}_1$
$\Omega \vdash e_1' \cdot e_2 \in \tau$	by rule

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash e_1 \equiv e_1' \in \tau'\end{array}}{\Omega \vdash (e_1 \mapsto_{\tau'} e_2) \cdot e_1' \to e_2}$$

$\vdash \Omega$ ok	
$\Omega \vdash (e_1 \mapsto_{\tau'} e_2) \cdot e_1' \in \tau$	by assumption
$\Omega \vdash (e_1 \mapsto_{\tau'} e_2 \in \tau' \Rightarrow \tau$	by inversion
$\Omega \vdash e_2 \in \tau$	by inversion

Case:

$$\mathcal{D} = \cfrac{}{\Omega \vdash (e_1 \mid e_2) \to e_i}$$

$\vdash \Omega$ ok	
$\Omega \vdash (e_1 \mid e_2) \in \tau$	by assumption
$\Omega \vdash e_i \in \tau$	by inversion

Case:

$$\mathcal{D} = \cfrac{}{\Omega \vdash \text{fix } u \in \tau.\, e \to [\text{fix } u \in \tau.\, e/u]e}$$

$\vdash \Omega$ ok	
$\Omega \vdash (\text{fix } u \in \tau.\, e) \in \tau$	by assumption
$\Omega = \Omega', (\Gamma; \Phi)$	by lemma 2
$\Omega', (\Gamma; \Phi, u \in \tau) \vdash e \in \tau$	by inversion
$\Omega \vdash \text{id}_\Omega, (\text{id}_\Gamma; \text{id}_\phi, \text{fix } u \in \tau.\, e/u) : \Omega', (\Gamma'; \Phi, u \in \tau)$	by definition 5
$\Gamma \leq \Gamma$	by rule
$\Phi \leq \Phi$	by rule
$\Phi \leq \Phi, u \in \tau$	by rule
$\vdash \Omega', (\Gamma'; \Phi, u \in \tau)$ ok	by lemma 4
$\Omega \vdash [\text{fix } u \in \tau.\, e/u]e : \tau$	by lemma 8

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash M : A\end{array}}{\Omega, (\Gamma; \Phi) \vdash \epsilon x : A.\, e \to [M/x]e}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok	
$\Omega, (\Gamma; \Phi) \vdash \epsilon x : A.\, e \in \tau$	by assumption
$\Omega, (\Gamma, x : A; \Phi) \vdash e \in \tau$	by inversion
$\Omega, (\Gamma; \Phi) \vdash \text{id}_\Omega, (\text{id}_\Gamma, M/x; \text{id}_\Phi) : \Omega, (\Gamma, x : A; \Phi)$	by definition 5
$\Gamma \leq \Gamma$	by rule
$\Gamma \leq \Gamma, x : A$	by rule
$\Phi \leq \Phi$	by rule
$\vdash \Omega, (\Gamma, x : A; \Phi)$ ok	by lemma 4
$\Omega, (\Gamma; \Phi) \vdash [M/x]e \in \tau$	by lemma 8

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \vdash f \in \tau\end{array}}{\Omega \vdash \epsilon u \in \tau.\, e \rightarrow [f/u]e}$$

$\vdash \Omega$ ok	
$\Omega \vdash \epsilon u \in \tau.\, e \in \tau$	by assumption
$\Omega = \Omega', (\Gamma; \Phi)$	by lemma 2
$\Omega', (\Gamma; \Phi, u \in \tau) \vdash e \in \tau$	by inversion
$\Omega', (\Gamma; \Phi) \vdash \mathrm{id}_\Omega, (\mathrm{id}_\Gamma; \mathrm{id}_\Phi, f/u) : \Omega', (\Gamma; \Phi, u \in \tau)$	by definition 5
$\Gamma \leq \Gamma$	by rule
$\Phi \leq \Phi$	by rule
$\Phi \leq \Phi, u \in \tau$	by rule
$\vdash \Omega', (\Gamma; \Phi, u \in \tau)$ ok	by lemma 4
$\Omega', (\Gamma; \Phi) \vdash [f/u]e : \tau$	by the big substitution lemma
$\Omega \vdash [f/u]e : \tau$	by equality

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma \vdash y : A\end{array}}{\Omega, (\Gamma; \Phi) \vdash \nabla x : A.\, e \rightarrow [y/x]e}$$

See the case for $\epsilon x : A.\, e$

Case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega, (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash e \rightarrow e'\end{array}}{\Omega, (\Gamma; \Phi) \vdash \nu x : A.\, e \rightarrow \nu x : A.\, e'}$$

$\vdash \Omega, (\Gamma; \Phi)$ ok	
$\Gamma, (\Gamma; \Phi) \vdash \nu x : A.\, e \in \tau$	by assumption
$\Gamma, (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash e \in \Box\tau$	by inversion
$\Gamma, (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash e' \in \Box\tau$	by the IH on $\mathcal{D}_1$
$\Gamma, (\Gamma; \Phi) \vdash \nu x : A.\, e' \in \tau$	by rule

Case:

$$\mathcal{D} = \dfrac{}{\Omega \vdash \nu x : A.\, \mathrm{pop}\ e \rightarrow e}$$

$\vdash \Omega$ ok	
$\Omega \vdash \nu x : A.\, \mathrm{pop}\ e : \tau$	by assumption
$\Omega = \Omega', (\Gamma; \Phi)$	by lemma 2
$\Omega', (\Gamma; \Phi), (\Gamma, x : A; \Phi) \vdash \mathrm{pop}\ e \in \Box\tau$	by inversion
$\Omega, (\Gamma; \Phi) \vdash e \in \tau$	by inversion

The following corollary is useful for proving the scope preservation corollary.

Corollary 1. *If* $\vdash \Omega$ ok *and* $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \rightarrow^* e'$ *then* $\Omega \vdash e' \in \tau$

Proof. By straightforward induction on the structure of $\Omega \vdash e \rightarrow^* e'$ using Theorem 2

Lemma 9 (Value Inversion). *If v is a value then all of the following hold*

1. *If* $\Omega \vdash v \in \langle A \rangle$ *then* $v = \langle M \rangle$, $\Omega = \Omega', (\Gamma; \Phi)$, *and* $\Gamma \vdash M : A$.
2. *If* $\Omega \vdash v \in \tau_1 \star \tau_2$ *then* $v = (e_1, e_2)$, $\Omega \vdash e_1 : \tau_1$, *and* $\Omega \vdash e_2 : \tau_2$.
3. *If* $\Omega \vdash v \in \tau_1 \Rightarrow \tau_2$ *then* $v = e_1 \mapsto e_2$, $\Omega \vdash e_1 : \tau_1$ *and* $\Omega \vdash e_2 : \tau_2$.
4. *If* $\Omega \vdash v \in \Box\tau$ *then* $v = \text{pop } e$, $\Omega = \Omega', (\Gamma; \Phi)$, *and* $\Omega' \vdash e \in \tau$

Proof. By cases on v and the given typing judgments.

As a corollary we obtain the property that parameters cannot escape their scope.

Corollary 2 (Scope Preservation). *If* $\vdash \Omega, (\Gamma; \cdot)$ ok, $\Omega, (\Gamma; \cdot) \vdash e \in \Box\tau$, $\Omega, (\Gamma; \cdot) \vdash e \rightarrow^* v$ *and v is a value then* $v = \text{pop } e'$ *and* $\Omega \vdash e' \in \tau$.

Proof. By direct applications of Corollary 1 and Lemma 9.

6 Conclusion

In this paper we have presented the ∇-calculus. We allow for evaluation under λ-binders, pattern matching against parameters, and programming with higher-order encodings. The ∇-calculus has been implemented as a stand-alone programming language, called ELPHIN [PS04]. The ∇-calculus solves many problems associated with programming with higher-order abstract syntax. We allow for, and can usefully manipulate, datatype declarations whose constructor types make reference to themselves in negative positions while maintaining a closed description of the functions.

The ∇-calculus is the result of many years of design, originally inspired by an extension to ML proposed by Dale Miller [Mil90]. Other influencing works include pattern-matching calculi as employed in ALF [CNSvS94] or proposed by Jouannaud and Okada [JO91], the type theory $\mathcal{T}^+_\omega$ [Sch01], and Hofmann's work on higher-order abstract syntax [Hof99]. A direct predecessor to the ∇-calculus is the modal λ-calculus with iterators [SDP01]. We conjecture that any function written in the modal λ-calculus with iterators can also be expressed in the ∇-calculus.

Closely related to our work are programming languages with freshness [PG00,GP99], which provide a built-in α-equivalence relation for first-order encodings but provide neither $\beta\eta$ nor any support for higher-order encodings. Also closely related to the ∇-calculus are meta-programming languages, such as MetaML [TS00,Nan02], which provide hierarchies of computation levels, but do not single out a particular level for representation. Many other attempts have been made to combine higher-order encodings and functional programming, in particular Honsell, Miculan, and Scagnetto's embedding of the π-calculus in Coq[HMS01], and Momgliano, Amber, and Crole's Hybrid system [MAC03].

In future work, we plan to extend the ∇-calculus to a dependently-typed logical framework, add polymorphism to the computation-level, and study termination and progression.

Acknowledgments. We would like to thank Henrik Nilsson, Simon Peyton-Jones, and Valery Trifonov for comments on earlier drafts of this paper.

References

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (Co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[Hof99] Martin Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In Gilles Kahn, editor, *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[MAC03] Alberto Momgliano, Simon Ambler, and Roy Crole. A definitional approach to primitive recursion over higher order abstract syntax. In Alberto Momgliano and Marino Miculan, editors, *Proceedings of the Merlin Workshop*, Uppsala, Sweden, June 2003. ACM Press.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, May 1990.

[Nan02] Aleksander Nanevski. Meta-programming with names and necessity. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02)*, Pittsburgh, PA, October 2002. ACM Press.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1996, May 1992.

[PG00] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[PS04] Adam Poswolsky and Carsten Schürmann. Elphin: Functional programming with higher-order encodings. Technical report, Yale University, 2004. to appear.

[Sch01] Carsten Schürmann. Recursion for higher-order encodings. In Laurent Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[SDP01] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, (266):1–57, 2001. Journal version of [DPS97].

[TS00] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.