Comparing Hagino's categorical programming language and typed lambda-calculi

Hans Dybkjær*
Centre for Cognitive Informatics, Roskilde University, P.O. Box 260, DK-4000 Roskilde, Denmark

Austin Melton** ***
Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

Abstract

Hagino (1987) develops CPL, a categorical programming language based on dialgebras which include algebras, coalgebras, products, sums and exponentials. We give an introduction to dialgebras and CPL.

Working from the well-known correspondence between cartesian closed categories (CCCs) and λ-calculi (Lambek, 1980; Curien, 1986), we study the relationship between CPL and $F_1$, Church’s simply typed λ-calculus. We show that the reduction rules of CPL correspond to β-reduction in first-order contexts.

Iteration over inductive types may be added to $F_1$, obtaining $F^i_1$ (Pierce et al., 1989). We show how to represent $F^i_1$ inductive types in CPL. Thus Ackermann’s function is in CPL. We argue that all natural number functions $\lambda x . y \rightarrow x$ provably total in first-order arithmetic can be expressed in CPL.

1. Introduction

Recently, Hagino [10] developed a theory of dialgebras. Initial and final dialgebras can define data types complete with constructors, destructors and computation rules.

*Correspondence to: H. Dybkjær, Centre for Cognitive Informatics, Roskilde University, P.O. Box 260, DK-4000 Roskilde, Denmark. Email: dybkjaer@ruc.dk.

**Research is supported in part by ONR Grant N00014-88-K-0455.

***Present address: Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA. Email: austin@cs.mtu.edu.
Dialgebras generalize algebras (e.g., natural numbers, lists, trees, ...) and coalgebras (e.g., natural numbers with an infinite element, streams, ...). Using dialgebras we can also represent sums, products, and even exponentials.

Hagino uses dialgebras in defining CPL ("Categorical Programming Language"), which is a combinator language with abstract data types and ML-like polymorphism. It has some similarity to CAML [11,4], but is not as ad hoc since all combinators and reduction rules arise from an abstract data type declaration mechanism based on dialgebras. Hagino has shown CPL to be strongly normalizing and to be at least primitive recursive.

We compare CPL with $\mathcal{F}_1$, the simply typed $\lambda$-calculus with unspecified base types [16]. In CPL we can define the types unit $1$, product $A \times B$, and function $A \Rightarrow B$; thus CPL is built on a cartesian closed category (CCC); and imply typed $\lambda$-calculi correspond closely to CCCs [13, 5]. However, being a programming language, CPL has deterministic reduction rules, in contrast to the nondeterministic conversion rules of $\mathcal{F}_1$. Further, not all CCC equations are used as reduction rules in CPL. We show that CPL cannot simulate $\eta$-reduction and that $\beta$-reduction is fully simulated only within terms of first-order type.

In [16] inductive types with iterators are added to $\mathcal{F}_1$, producing $\mathcal{F}_1^i$. We show how to extend the $\mathcal{F}_1$-to-CPL translation to $\mathcal{F}_1^i$. As an application we encode Ackermann's function in CPL, which shows that CPL is more than primitive recursive. We argue that CPL can represent all functions $\downarrow \rightarrow \downarrow$ provably total in first-order arithmetic.

In this paper we first give introductory descriptions of dialgebras and CPL with emphasis on reduction rules. Next we translate $\mathcal{F}_1$ into CPL, and we then extend the translation to inductive types, i.e., to $\mathcal{F}_1^i$. Finally, we discuss related work and possibilities for further research.

1.1. Notational conventions. $x = y$ is equality, and $x =_{D} y$ is definitional equality. Diagrammatic notation—$a^* f$ for application ($a$ transformed by $f$) and $f; g$ for composition—is used in mathematical parts. In programs the applicative notations $f a$ and $g \circ f$ are used. $\langle x, y \rangle$ is a pair of elements in $X \times Y$, not to be confused with the arrow pair$(f, g): Z \rightarrow X \times Y$. $i.X. Expr$ is the abstraction of $X$ over $Expr$, not to be confused with the program $\lambda x.e$. For a set $X$, the set of finite sequences of elements from $X$ is denoted $X^*$. $[e_1/x]e_2$ denotes substitution of $e_1$ for $x$ in $e_2$. Diagrams commute. The natural numbers always include zero, successor and iterator. $i$ denotes the identity.

Readers are assumed to be familiar with the concepts of (cartesian closed) category, functor, and adjoint, and to have seen some programming language semantics.

2. Dialgebras

The dialgebras will be used prescriptively rather than descriptively, so we carefully describe specifications of categories before introducing dialgebras. The presentation of dialgebras will be followed by a parallel presentation of CPL.
2.1. Category specifications

A rather low-level language for the specification of categories is described. The language is similar to, but simpler than, the categorical specification language of [10] and is related to sketches [2]. It is also simpler than—and not as powerful as—the type stack formalism of Chen and Cockett [17]. The ideas are close to the equational algebraic specification methods [7].

We explain what theories as signatures with equations and what models are; in Section 3.1 the free category gives a weak formulation of a categorical programming language.

2.1.1. Variances. Let \( C \) be a category. Define \( C^\perp \) as \( \perp \) (the category with one object and one arrow), \( C^{\perp} \) as the dual of \( C \), \( C^+ \) as \( C \), and \( C^T \) as the discrete category with \( C^T_{\text{obj}} = C_{\text{obj}} \).

The set \( \text{Var} = \{ \perp, -, +, T \} \) of variances is both a lattice and a monoid with unit + and composition \( \bullet \) defined as in Fig. 1 [10].

More precisely, \( (\text{Var}, \text{lub}, \bullet, \perp) \) is a commutative semiring with unit +.

2.1.2. Definition. Let \( F : A \to B \) be a mapping of objects in \( A \) to objects in \( B \) and arrows in \( A \) to arrows in \( B \). We call \( F \)
- free-variant if \( F \) does not depend on its argument (i.e., \( F \) is a constant) (\( F \) is essentially the functor \( A \to B^\perp \));
- contravariant if \( F \) maps \( f : A_1 \to A_2 \) to \( f^* F : A_2^* \to A_1^* F \) (\( F \) is a functor \( A \to B^\perp \));
- covariant if \( F \) is a functor \( A \to B \);
- fixvariant if \( F \) is contravariant on some arrows and covariant on others (\( F \) defines a functor \( A \to B^\perp \)).

In specifications, the ("polymorphic") objects will be based on functors of the form \( F : \prod C^\perp \to C \).

2.1.3. Definition. Let \( \Phi \equiv \bigcup_{\tau \in \text{Var}^*} \Phi_\tau \) be a \( \text{Var}^* \) indexed set of sets of functor names. The set \( \Phi \) of sets of closed functorial expressions is defined as:
- \( \lambda X_1 \cdots X_n : F C_{\text{obj}} \in \Phi_{\tau_1 \cdots \tau_n} \) where \( \tau_i = \perp \) if \( i \neq j \) and \( \tau_i = + \) if \( i = j \) (projection);
- if \( \Phi_{\tau_1 \cdots \tau_n} \), then \( \lambda X_1 \cdots X_k : F(X_{j_1}, \ldots, X_{j_n}) \in \Phi_{\tau_1 \cdots \tau_n} \) where \( \{ X_{j_1}, \ldots, X_{j_n} \} \subseteq \{ X_1, \ldots, X_k \} \) and \( \tau_i = \bigwedge_{m \geq 1} [m]_{X_m = X_i} \tau_m \) (base);

\[
\begin{array}{c}
\text{T} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\end{array}
\quad
\begin{array}{c}
\text{T} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\text{\perp} \\
\text{\neg} \\
\end{array}
\]

Fig. 1
• if $K \in \Phi_{u_1 \ldots u_n}$ and $K_j \in \Phi_{r_1 \ldots r_m}$ for $j = 1, \ldots, n$, then $K [ K_1, \ldots, K_n ] \in \Phi_{u_1 \ldots u_n}$ where
  
  $w_j = \bigsqcup_{i=1}^m v_{ij} \bullet u_i$ (composition), where
  
  $K [ K_1, \ldots, K_n ] = \lambda Y_1 \ldots Y_k. [ E_1 \ldots E_n / X_1 \ldots X_n ] E$
  
  for $K = \lambda X_1 \ldots X_n. E$ and $K_j = \lambda Y_1 \ldots Y_k. E_j$, $j = 1, \ldots, n$.
  
  If $F \in \Phi_{r_1 \ldots r_s}$, we write $F [ K_1, \ldots, K_n ]$ for $( \lambda X_1 \ldots X_n. F ( X_1 \ldots X_n ) ) [ K_1, \ldots, K_n ]$.
  
  $E$ is a functorial expression if there is $\tilde{X} = \langle X_1, \ldots, X_n \rangle$ such that $\lambda \tilde{X} . E \in \Phi_{r_1 \ldots r_s}$ is
  a closed functorial expression; we write $E \in \Phi_{r_1 \ldots r_s} ( \tilde{X} )$.

  Note how the abstraction of variables (representing objects from $C^{n_1} \times \cdots \times C^{n_s}$) are
  syntactically kept at the outermost, leftmost position. We treat only first-order
  functors, i.e., there are no exponential objects at the functor level. Interpreting
  functors as types yields a restricted form of polymorphism, called functorial
  polymorphism.

2.1.4. Example. The functor names relevant to a cartesian closed category are

  \[ \Phi_{+} = \{ 1 \}, \quad \Phi_{+} = \{ \text{prod} \}, \quad \Phi_{-} = \{ \text{exp} \}, \]

  where ( ) is the empty sequence of variances. This will also be written as

  $\Phi = \{ 1, \text{prod}_{+}, \text{exp}_{-} \}$.

  Then $\hat{\Phi}$ contains, e.g., $\lambda X. 1( ) \in \hat{\Phi}_{+}$, $\lambda X. X \in \hat{\Phi}_{+}$ (the first projection of a one-tuple
  amounts to the identity), $\lambda X Y Z. \text{prod} ( \text{exp} ( X, Y )$, $X ) \in \hat{\Phi}_{+}$, and $\lambda X Y. Y \in \hat{\Phi}_{-}$.

  Now we are prepared for the definition of signatures.

2.1.5. Definition. A (category) signature $\Sigma$ is a pair $\langle \Phi, \Psi \rangle$ where

  - $\Phi$ is a $\text{Var}^*$ indexed set of functor names $F, G, \ldots$
  
  - $\Psi$ is a $(\hat{\Phi} \times \hat{\Phi})^* \times (\hat{\Phi} \times \hat{\Phi})$ indexed set of arrow names.

  For each “index” of elements in $\Psi$, the involved functorial expressions must be of the
  same arity.

2.1.6. Example. The signature for cartesian closed categories is given by

  $\Phi = \{ 1, \text{prod}_{+}, \text{exp}_{-} \}$,

  $\Psi = \{ 1 \langle \lambda X. X. \lambda X. 1( ) \rangle \}$ (the unique arrow into 1),

  $\text{pl}_{\langle \lambda X Y. \text{prod} ( X, Y ), \lambda X Y. X \rangle}$,

  $\text{pr}_{\langle \lambda X Y. \text{prod} ( X, Y ), \lambda X Y. X \rangle}$,

  $\text{pair}_{\langle \lambda X Y Z. \lambda X Y Z. X Y Z, \lambda X Y Z. \lambda X Y Z. \text{prod} ( X, Y ) \rangle}$,

  $\text{app}_{\langle \lambda X Y. \text{prod} ( \text{exp} ( X, Y ), X ), \lambda X Y. X \rangle}$,

  $\text{cur}_{\langle \lambda X Y Z. \text{prod} ( X, Y ), \lambda X Y Z. \lambda X Y Z. \lambda X Y Z. \text{exp} ( Y, Z ) \rangle}$.
Fortunately, we have informal and more intuitive notation for the above. For example, we can write

\[ p_1 : \text{prod}(X, Y) \to X \quad \text{and} \quad f : \text{prod}(X, Y) \to Z \]

where the abstractions have been moved to the meta-level, arguments to arrows are explicitly named, and parentheses after 0ary arrow names are omitted.

2.1.7. Definition. A model of a signature \( \Sigma = \langle \Phi, \Psi \rangle \) is a pair \((C, \bar{-})\) of a category \(C\) and an assignment \(\bar{-}\) such that

- If \(F : \Phi_{\ell_1, \ldots, \ell_n} \to C\), then \(F : C^{\ell_1} \times \cdots \times C^{\ell_n} \to C\) is a functor.
- If \(\psi \in \Psi_{\langle K_1, K'_1 \rangle \cdots \langle K_m, K'_m \rangle \langle K, K' \rangle}, \bar{X} \in \mathbb{C}_{\text{obj}} \times \cdots \times \mathbb{C}_{\text{obj}}\)
  and \(e_j : \bar{X} \cdot K_j \to \bar{X} \cdot K'_j, j = 1, \ldots, m\) are arrows in \(C\),
  then \(\langle e_1, \ldots, e_m \rangle \) is an arrow in \(C\).

The functor assignment extends in a unique way to \(\bar{\Phi}\).

2.1.8. Definition. Let \(\Sigma\) be a signature and \(X\) be a \(\Phi \times \Phi\) indexed set of arrow variables. The free arrow algebra \(A_{\Sigma, X}\) is formed inductively as

- \(1 \in A_{\Sigma, X}^{K - K'}\);
- \(e_1 \in A_{\Sigma, X}^{K - K'}\) and \(e_2 \in A_{\Sigma, X}^{K' - K''}\), then \(e_2 \circ e_1 \in A_{\Sigma, X}^{K - K''}\);
- \(f \in X_{\langle K, K' \rangle}\), then \(f \in A_{\Sigma, X}^{K - K'}\);
- \(K \in \Phi_{\ell_1, \ldots, \ell_n}\)

and for \(j = 1, \ldots, n\) \[ e_j \in A_{\Sigma, X}^{K, \ell_j - K'} \quad \text{if} \quad v_j = \bot \quad \text{or} \quad v_j = +, \]
\[ e_j \in A_{\Sigma, X}^{K, \ell_j - K'} \quad \text{if} \quad v_j = -, \]
\[ e_j = \mathbf{1} \in A_{\Sigma, X}^{K, 0} \quad \text{if} \quad v_j = \top, \]

then \(K(e_1, \ldots, e_n) \in A_{\Sigma, X}^{K, \ell} = k_{\ell_1, \ldots, \ell_n} \).

- If \(\psi \in \Psi_{\langle K_1, K'_1 \rangle \cdots \langle K_m, K'_m \rangle \langle K, K' \rangle}\) and \(e_j \in A_{\Sigma, X}^{K, \ell_j - K'}\) for \(j = 1, \ldots, m\), then \(\bar{\psi}(e_1, \ldots, e_m) \in A_{\Sigma, X}^{K, K'}\).

2.1.9. Definition. Let \(\Sigma\) be a signature. A conditioned equation in \(\Sigma\) is a formula

\[ \forall f^1_{\ell_1} \cdots f^k_{\ell_k} \cdot K, e_1 = e'_1 \land \cdots \land e_l = e'_l \quad \Rightarrow \quad e_0 = e'_0 \]

where \(e_j, e'_j \in A_{\Sigma, X}^{K, \ell_j - K'}\), \(j = 0, \ldots, l\).

2.1.10. Definition. A category specification (or theory) is a pair \((\Sigma, \Xi)\) of a signature \(\Sigma\) and a set \(\Xi\) of conditioned equations in \(\Sigma\).

2.1.11. Example. The uniqueness of pairing is given by the (conditioned) equation

\[ \forall h : \text{prod}(A, B), \text{pair}(p_1 \circ h, p_2 \circ h) = h. \]
This equation and most other equations we shall see have no conditions. Conditions will be needed for recursive structures like natural numbers and streams (Section 2).

For readability and without loss of significant precision, the abstractions are often dropped. For example, the conditioned equations for a cartesian closed theory are given by

\[(!') \quad ! = f,\]
\[(fst) \quad \text{plopair}(f, g) = f,\]
\[(snd) \quad \text{p2o(pair}(f, g) = g,\]
\[(pair') \quad \text{pair}(p1 o h, p2 o h) = h,\]
\[(app) \quad \text{app(pair(cur(f)), g) = f o pair(i, g)),}\]
\[(cur') \quad \text{cur(app(pair(h o p1, p2))) = h.}\]

A superscript `' indicates that the equation expresses uniqueness.

2.1.12. Definition. A model of a category specification \(C = (\Sigma, \Xi)\) is a category \(\mathcal{C}\) and an assignment \(\gamma\) such that \((\mathcal{C}, \gamma)\) is a model of \(\Sigma\) and such that for each equation

\[\forall f^1, \ldots, f^k : \mathcal{C}^{\text{obj}} \times \cdots \times \mathcal{C}^{\text{obj}}, \quad e_1 = e'_1 \land \cdots \land e_l = e'_l \Rightarrow e_0 = e'_0,\]

objects \(\overline{X} = \langle X_1, \ldots, X_n \rangle \in \mathcal{C}^{\text{obj}} \times \cdots \times \mathcal{C}^{\text{obj}}\), and arrows \(g^1, \ldots, g^k : \mathcal{C}^{\text{obj}} \times \cdots \times \mathcal{C}^{\text{obj}}\), the proposition

\[\overline{e}_1 = \overline{e}_1' \land \cdots \land \overline{e}_l = \overline{e}_l' \Rightarrow \overline{e}_0 = \overline{e}_0'\]

holds in \(\mathcal{C}\) under the assumptions \(\overline{f} = \gamma, \quad r = 1, \ldots, k.\)

2.2. Dialgebras

Dialgebras generalize algebras and coalgebras. Thus, for example, the natural numbers which are definable as an initial algebra are also definable as an initial dialgebra. In addition dialgebras, which "combine" algebras and coalgebras, are more powerful than the "union" of algebras and coalgebras. Using dialgebras one can define sums, products, and even exponentials.

2.2.1. Definition. Let \(F, G : \mathcal{C} \to D\) be two functors. An \(F, G\)-dialgebra is a pair \(\langle C, f \rangle\) in \(\mathcal{C}^{\text{obj}} \times D^{\text{arr}}\) where \(f : C^* F \to C^* G\).

We can define a category \(\text{DiAlg}_{F, G}\) in which objects are \(F, G\) dialgebras and arrows \(h : \langle A, f \rangle \to \langle B, g \rangle\) are arrows \(\overline{h} \in \text{Hom}_C(A, B)\) such that

\[
\begin{array}{ccc}
A^* F & \xrightarrow{f} & A^* G \\
\downarrow \text{h}^* F & & \downarrow \text{h}^* G \\
B^* F & \xrightarrow{g} & B^* G
\end{array}
\]
We often use \( \langle \mu(F, G), x \rangle \) to designate the initial object of \( \text{DiAlg}_{F,G} \); thus, \( \mu(F, G) \in \text{Obj} \) and \( x \in \text{Hom}_0((\mu(F, G))^*F, (\mu(F, G))^*G) \). Given an initial object \( \langle \mu(F, G), x \rangle \) and an arbitrary \( F,G \)-dialgebra \( \langle X, f \rangle \), the unique arrow or factorizer \( \langle \mu(F, G), x \rangle \rightarrow \langle X, f \rangle \) is denoted by \( f^* \). Dually \( \langle \nu(F, G), z \rangle \) denotes the final object, and again \( f^* \) denotes the factorizer. Note that initial and/or final objects need not exist.

2.2.2. Natural numbers. Let \( F \equiv \lambda X. \langle 1, X \rangle \) and \( G \equiv \lambda X. \langle X, X \rangle \) be functors \( \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{C} \). The initial \( F,G \)-dialgebra \( \langle N, \langle z, s \rangle \rangle \) fills in the diagram shown in Fig. 2. Since this diagram lives in \( \mathcal{C} \times \mathcal{C} \) we can look at the first and second components separately. The diagram unfolds to the diagram in \( \mathcal{C} \) shown in Fig. 3. Thus by using dialgebras, natural numbers can be defined without presuming \( + \). Moreover, we get the zero and successor arrows for free; they satisfy the equations

\[
\begin{align*}
(z) & \quad z \cdot \langle f, g \rangle^* \psi = f, \\
(s) & \quad s \cdot \langle f, g \rangle^* \psi = \langle f, g \rangle^* \psi : g.
\end{align*}
\]

2.2.3. Sum. For fixed \( A, B \) and arbitrary \( C \), define

\( F \equiv \lambda C. \langle A, B \rangle \) and \( G \equiv \lambda C. \langle C, C \rangle \)

to be functors \( \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{C} \). Let \( \langle S, \langle i_1, i_2 \rangle \rangle \equiv \langle \mu(F, G), x \rangle \), and let \( \langle C, \langle f, g \rangle \rangle \) be an arbitrary \( F,G \)-dialgebra. Then \( \langle f, g \rangle^* \psi : \langle S, \langle i_1, i_2 \rangle \rangle \rightarrow \langle C, \langle f, g \rangle \rangle \) is the unique

\[
\begin{align*}
\langle 1, N \rangle & \quad \downarrow \langle z, s \rangle \\
\langle i_1, \langle f, g \rangle^* \psi \rangle & \quad \leftarrow \quad \langle (f, g)^* \psi, \langle f, g \rangle^* \psi \rangle \\
\langle 1, X \rangle & \quad \downarrow \langle f, g \rangle \\
\langle X, X \rangle & \quad \rightarrow
\end{align*}
\]

Fig. 2

\[
\begin{align*}
1 & \quad \downarrow \quad z \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow f \quad \downarrow g \\
N & \quad \rightarrow \quad \langle f, g \rangle^* \psi \\
\rightarrow \quad \rightarrow \quad \rightarrow \\
N & \quad \rightarrow \quad \langle f, g \rangle^* \psi \\
\rightarrow \quad \rightarrow \quad \rightarrow \\
X & \quad \rightarrow \quad X
\end{align*}
\]

Fig. 3
factorizer which fills out the diagram in Fig. 4. We may unfold the diagram as in Fig. 5. S is normally written as \( \langle A, B \rangle \) * sum or \( A + B \) with injections \( t_1 \) and \( t_2 \), and the factorizer \( \langle f, g \rangle \psi \) as \( \langle f, g \rangle \) * case or \( [f, g] \). This sum is an example of an initial dialgebra.

2.2.4. **Parametrized dialgebras.** Note that \( A, B \) are free variables in the construction of sums. In general, let \( F : \mathcal{C} \times \mathcal{K} \to \mathcal{D} \) and \( G : \mathcal{C} \times \mathcal{K}^{\text{op}} \to \mathcal{D} \) be functors. For \( X \in \mathcal{K}^{\text{obj}} \), define \( F_X, G_X : \mathcal{C} \to \mathcal{D} \) by \( F_X \equiv \lambda C. \langle C, X \rangle \) * \( F \) and \( G_X \equiv \lambda C. \langle C, X \rangle \) * \( G \). Further, define (if they exist) the functions \( \text{Left}_{F,G} : \text{Obj}(\mathcal{K}) \to \text{Obj}(\mathcal{C}) \) by

\[
(\text{Left}) \quad X^{\ast} \text{Left}_{F,G} \equiv \mu(F_X, G_X)
\]

and the dual

\[
(\text{Right}) \quad X^{\ast} \text{Right}_{F,G} \equiv \nu(F_X, G_X).
\]

We say that \( F_X, G_X \)-dialgebras are **parametrized** with respect to \( X \).

2.2.5. **Functors from parametrized dialgebras.** In [10] it is shown that \( \text{Left}_{F,G} \) extends to a functor \( \mathcal{K} \to \mathcal{C} \). For \( f : A \to B \) an arrow in \( \mathcal{K} \), the action of \( \text{Left}_{F,G} \) on \( f \) is defined as the unique arrow

\[
(\text{Left'}) \quad f^{\ast} \text{Left}_{F,G} = (\langle i, f \rangle \ast F; z_B; \langle i, f \rangle \ast G) \psi = h
\]

\[
\begin{array}{c}
\langle A, B \rangle \\
\downarrow \langle i_A, i_B \rangle \\
\langle A, B \rangle
\end{array} \xrightarrow{\langle t_1, t_2 \rangle} \begin{array}{c}
\langle S, S \rangle \\
\downarrow \langle (f, g) \psi, (f, g) \psi \rangle \\
\langle C, C \rangle
\end{array}
\]

Fig. 4.

\[
\begin{array}{c}
A \\
\downarrow f \\
C \\
\downarrow g \\
B
\end{array}
\xrightarrow{\langle f, g \rangle \psi} \begin{array}{c}
S \\
\downarrow t_1 \\
A \\
\downarrow t_2 \\
B
\end{array}
\]

Fig. 5.
Comparing Hagino's CPL and typed \(\eta\)-calculi

that fills out the diagram shown in Fig. 6. From this diagram we see the reason for the "op" in the type of \(G\). It is relatively easy to verify that \(\text{Left}_{F,G}\) satisfies the functor axioms.

Of course we have the dual result for \(\text{Right}_{F,G}\):

\[(\text{Right}') \quad f^* \text{Right}_{F,G} \equiv (\langle i, f \rangle)^*F ; x_B ; \langle i, f \rangle^*G)^*\psi.\]

2.2.6. Sum (continued). Define \(F, G : C \times (C \times C) \rightarrow C \times C\) by \(F \equiv \lambda C(AB) \langle A, B \rangle\) and \(G \equiv \lambda C(AB) \langle C, C \rangle\). The functor \(\text{Left}_{F,G} : C \times C \rightarrow C\) is defined by

\[\langle A, B \rangle^* \text{Left}_{F,G} \equiv \mu(F(\langle A, B \rangle), G(\langle A, B \rangle)) = A + B\]

and for \(\langle f, g \rangle : \langle A, B \rangle \rightarrow \langle A', B' \rangle\)

\[\langle f, g \rangle^* \text{Left}_{F,G} \equiv (\langle f, g \rangle ; \langle t_1, t_2 \rangle ; \langle i_{A' + B'}, i_{A' + B} \rangle)^*\psi = \langle f ; t_1, g ; t_2 \rangle^{*\text{case}}.\]

To illustrate that dialgebras generalize adjoints (and thus algebras), we have the following theorem.

2.2.7. Theorem [10, Proposition 3.1.5]. For a functor \(F : C \rightarrow D\), its left adjoint functor (if it exists) can be denoted by

\[\text{Left}^{\langle (X, Y), Y \rangle, X^*F} \quad \text{and, dually, its right adjoint functor (if it exists) can be denoted by}\]

\[\text{Right}^{\langle (X, Y), X^*F \rightarrow \langle (X, Y), Y \rangle}.\]

In the category \(D\) \quad \text{In the category \(\text{DiAlg}_{F,G}\)}

\[
\begin{array}{ccc}
\langle A', \text{Left}_{F,G}, A \rangle^*F & \xrightarrow{\alpha_A} & \langle A', \text{Left}_{F,G}, A \rangle^*G \\
(h, i)^*F & \downarrow & (h, i)^*G \\
\langle B', \text{Left}_{F,G}, A \rangle^*F & \xrightarrow{(B', \text{Left}_{F,G}, (i, f)^*F; \alpha_B; (i, f)^*G)} & \langle B', \text{Left}_{F,G}, A \rangle^*G \\
(i, f)^*F & \downarrow & (i, f)^*G \\
\langle B', \text{Left}_{F,G}, B \rangle^*F & \xrightarrow{\alpha_B} & \langle B', \text{Left}_{F,G}, B \rangle^*G \\
\end{array}
\]

Fig. 6.
2.3. Notation for data types defined via dialgebras.

As we have seen above, if an initial or final parametrized dialgebra exists, we obtain a functor (Left_{F,G} or Right_{F,G}), arrows (the ψs), and a unique mapping (induced by initiality or finality). In the rest of this paper, we consider the restricted case where we have a base category C and two functors \( F, G : \text{C} \times \text{C}^n \rightarrow \text{C}^m \), corresponding to \( K \equiv \text{C}^n \) and \( D \equiv \text{C}^m \). We will consider (polymorphic) data types as functors \( \text{C}^n \rightarrow \text{C} \). The \( n \) signifies that there are \( n \) type variables in the definition of a polymorphic type. The \( \text{C}^m \) corresponds to the \( m \) constructors/destructors that are defined by the initial/final dialgebra.

We define a syntax for denoting initial and final dialgebras, complete with the derived constructors and destructors, and we give a couple of examples. We assume that \( (\Sigma, \Xi) \) is a given category specification with \( \Xi = (\Phi, \Psi) \).

2.3.1. Definition. Following [10] we use the notation

left object \( L(X_1, \ldots, X_n) \) with \( \psi_L \) is

\[
\begin{align*}
\alpha_{L,1} & : E_{L,1} \rightarrow \hat{E}_{L,1} \\
& \quad \ldots \\
\alpha_{L,m} & : E_{L,m} \rightarrow \hat{E}_{L,m}
\end{align*}
\]

end

where \( E_{L,j}, \hat{E}_{L,j} \in \Phi(L, X_1, \ldots, X_n) \) are functorial expressions for \( j = 1, \ldots, m \).

Let \( v_j, v'_{j,1}, \ldots, v'_{j,n} \) and \( v''_{j,1}, \ldots, v''_{j,n} \) be the variances of \( E_j \) and \( \hat{E}_j \), respectively. We use the notation \( E_j^A \) for \( [A/L] E_j \) and \( \hat{E}_j^A \) for \( [A/L] \hat{E}_j \). The declaration adds

- to \( \Phi \) the functor name \( L \) of variance \( u_1 \ldots u_n \),
- where \( u_j = \bigsqcup_{i=1}^m (v_{j,i} \cup (-v'_{j,i})) \)
- to \( \Psi \langle X, E_j^A, \hat{E}_j^A \rangle \) the transformation name \( \alpha_{L,j} \) for each \( j = 1, \ldots, m \).
- to \( \Psi \rangle X, X, X, \ldots, X, X, X, \ldots, X, X, X, X, X \rangle \) the factorizer name \( \psi_L \).
- to \( \Xi \) the equations expressing commutativity and initiality of left objects; see the defining diagrams and uniqueness equations of Section 2.4.

Note how \( L \) plays a double role: in the type of \( \alpha_{L,j} \) as the applied functor \( L(X) \), and in the argument types of \( \psi_L \) as a parameter \( X \).

2.3.2. Sum again. Using the above notation, the sum may be defined as:

left object \( \text{sum}(A, B) \) with case is

\[
\begin{align*}
in1 & : A \rightarrow \text{sum} \\
in2 & : B \rightarrow \text{sum}
\end{align*}
\]

end
Comparing Hayino's CPL and typed $\lambda$-calculi

If $\bar{E}, \tilde{E}$ are the expressions $\langle A, B \rangle$ and $\langle S, S \rangle$, respectively, then $\langle \langle A, B \rangle^* \quad \text{sum}, \quad \langle \text{in1}, \text{in2} \rangle \rangle$ is the initial $\lambda S. \langle A, B \rangle, \lambda S. \langle S, S \rangle$-dialgebra, and case takes two arrows $f: A \to S, g: B \to S$ and produces a (unique) arrow $\langle f, g \rangle^* \text{case}: \langle A, B \rangle^* \quad \text{sum} \to S$.

We will use italics to denote the assigned meanings, e.g., sum, case, in1 and in2 for sum, case, in1 and in2, respectively.

2.3.3. Dually we have right objects which define functors $R: C^n \to C$ such that $\langle X^* R, \cdot \rangle$ is the final $F_X, G_X$-dialgebra. For example, the product may be specified as right object $\text{prod}(A, B)$ with pair is

$P_1 : \text{prod} \to A$

$P_2 : \text{prod} \to B$

end

Note that both this and the previous sum declaration were made without presuming any names, i.e., they might be made on the basis of an empty category specification.

2.3.4. Models. In accordance with Definition 2.3.1, models $(C, -)$ of $(\Sigma, \Xi)$ require the existence of:

- a functor

$L \equiv \text{Left}_{F, G}: \mathbb{C} \to \mathbb{C}$

where $F, G: C^1 \to \mathbb{C}^m$ are defined by $\langle L, X \rangle^* F \equiv \bar{E}_L$ and $\langle L, X \rangle^* G \equiv \tilde{E}_L$, respectively,

- a transformation

$\hat{x} : (X^* L)^* F_{\hat{x}} \to (X^* L)^* G_{\hat{x}}$

such that, for a given $\hat{x}$, $\langle X^* L, \hat{x} \rangle = \langle \mu(F_{\hat{x}}, G_{\hat{x}}), x \rangle = \langle X^* \text{Left}_{F, G}, x \rangle$, and

- a factorizer such that for given $X, \hat{X}$

$\psi : \text{Hom}_{\mathbb{C}}(X^* F_{\hat{x}} , X^* G_{\hat{x}}) = \text{Hom}_{\mathbb{C}}(X^* L, X)$

takes $m$ arrows $f_1, \ldots, f_m$ into the universal arrow $\tilde{j}^* \psi : (X^* L, x) \to (X, \tilde{j})$ (i.e., the underlying arrow in $\mathbb{C}$).

We always have a model, namely the category 1 with one arrow. The interesting, and still very open, question is under which conditions nontrivial models exist.

2.4. Equations of dialgebras

In the definition of a left object, we may see $\hat{x}$ as the constructors and $\psi$ as the destructor of the defined data type, and vice versa for the right objects. There are
a number of commuting diagrams, or equations, which are associated with these data types. We give some of these diagrams below.

2.4.1. Defining diagram. Let $L$ be a left object as defined in the previous section. Let $\tilde{X} \in C^n$. Then by definition and initiality the defining diagram given in Fig. 7 commutes.

Since this is a diagram in $C^n$, we may regard it as $m$ simultaneously defined diagrams or equations in $C$:

$$(L_{\text{def}}) \quad \alpha_j; \langle \tilde{f}^*\psi_i, i_X^* \rangle^* G_j = \langle \tilde{f}^*\psi_i, i_X^* \rangle^* F_j; \tilde{f}_j.$$

We have earlier seen examples of defining diagrams for the left objects $\text{sum}$ and $\text{nat}$. Dually, there is a defining diagram for right objects, with equations in $C$:

$$(R_{\text{def}}) \quad \langle \tilde{f}^*\psi_i, i_X^* \rangle^* F_j; \alpha_j = f_j; \langle \tilde{f}^*\psi_i, i_X^* \rangle^* G_j.$$

2.4.2. Streams as right object example. For right objects we have defining diagrams dual to the above. Assume the declaration:

```plaintext
right object stream(A) with srec is
  \text{hd:} \text{stream } \rightarrow \text{A}
  \text{tl:} \text{stream } \rightarrow \text{stream}
end
```

Then we have the diagrams shown in Fig. 8 and, unfolded, in Fig. 9. Intuitively $\langle f, g \rangle^* \text{srec}$ replaces a sequence $tl; \cdots; tl; \text{hd}$ with a sequence $g; \cdots; g; f$.

2.4.3. Uniqueness. Let $L$ be a left object as defined above. The uniqueness of the factorizer $\tilde{f}^*\psi$ is expressed as indicated in Fig. 10 or linearly by

$$(u_L) \quad \alpha; \langle h, i_X^* \rangle^* G - \langle h, i_X^* \rangle^* F; \tilde{f} \Rightarrow h - \tilde{f}^*\psi.$$

![Diagram Fig. 7](image-url)
Comparing Hagino's CPL and typed $\lambda$-calculi

\[
\begin{align*}
\langle X, X \rangle & \xrightarrow{\langle f, g \rangle} \langle A, X \rangle \\
\langle (f, g)'srec, (f, g)'srec \rangle & \xrightarrow{(i_A, (f, g)'srec)} \langle A, (f, g)'srec \rangle \\
\langle A'stream, A'stream \rangle & \xrightarrow{(hd, td)} \langle A, A'stream \rangle
\end{align*}
\]

Fig. 8.

\[
\begin{align*}
X & \xrightarrow{g} X \\
\langle f, g)'srec \rangle & \xrightarrow{f} A'stream \\
\langle f, g)'srec \rangle & \xrightarrow{tl} \langle f, g)'srec, td = g; (f, g)'srec \rangle \\
\langle f, g)'srec; td = g; (f, g)'srec \rangle & \xrightarrow{hd = f} A
\end{align*}
\]

Fig. 9.

\[
\begin{align*}
\langle X' \cdot L, X' \rangle \cdot F & \xrightarrow{\overline{\alpha}} \langle X' \cdot L, X' \rangle \cdot G \\
\langle h, i_{\bar{X}} \rangle \cdot F & \xrightarrow{\overline{\alpha}} \langle h, i_{\bar{X}} \rangle \cdot G \\
\langle X, X' \rangle \cdot F & \xrightarrow{h = \bar{f}'\psi} \langle X, X' \rangle \cdot F
\end{align*}
\]

Fig. 10.

A common special case is when $E_i$ does not contain $L$. Then the $j$th component of $\langle h, i_{\bar{X}} \rangle \cdot F: \bar{f}$ simplifies to $f_j$, and we may substitute the $j$th component of $\bar{z}; \langle h, i_{\bar{X}} \rangle \cdot G$ for $f_j$ in the right-hand side of $\bar{g} = \bar{f}'\psi$ instead of having $x; \langle h, i_{\bar{X}} \rangle \cdot G$ as a condition. If none of $E_1, \ldots, E_n$ contains $L$, we call the left object $L$ unconditioned.

The dual statement for right objects is

\[(u_R) \quad \langle h, i_{\bar{X}} \rangle \cdot F; \bar{z} = \bar{f}; \langle h, i_{\bar{X}} \rangle \cdot G \Rightarrow h = \bar{f}'\psi\]

and the right object is unconditioned if $F$ does not depend on $h$, i.e., if no $E_1, \ldots, E_n$ contains $L$.

An object that is not unconditioned is called conditioned or recursive.
2.4.4. **Uniqueness examples.** We give the uniqueness conditions for the four examples we have seen:

- **(uN)** \( s; h = h; g \Rightarrow h = \langle z; h, g \rangle^* \psi, \)
- **(uSum)** \( h = \langle t_1; h, t_2; h \rangle^* \text{case}, \)
- **(uProd)** \( h = \langle h; p_1, h; p_2 \rangle^* \text{pair}, \)
- **(uStream)** \( h; tl = g; h \Rightarrow h = \langle h; hd, g \rangle^* \text{rec}. \)

We see that *sum* and *product* are unconditioned while *N* and *stream* are recursive.

2.4.5. **Functor expansion.** Recalling the definition of \( L \), we have:

- \( \tilde{f}^* L = \tilde{f}^* \text{Left}_{f, g} = (\langle i, \tilde{f} \rangle^* F; \tilde{x}; \langle i, \tilde{f} \rangle^* G)^* \psi. \)

From this and from \( \tilde{f}^* L = i \) (because \( L \) is a functor), we get the special case

- **(iL)** \( i = (\langle i, \tilde{i} \rangle^* F; \tilde{x}; \langle i, \tilde{i} \rangle^* G)^* \psi = \tilde{x}^* \psi. \)

Dually for right objects we have

- **(R)** \( \tilde{f}^* R = (\langle i, \tilde{f} \rangle^* F; \tilde{x}; \langle i, \tilde{f} \rangle^* G)^* \psi. \)

2.4.6. **Streams continued.** For the stream example we obtain

- **(stream)** \( \tilde{f}^* \text{stream} = \langle \text{hd}; f, \text{tl} \rangle^* \text{rec}, \)
- **(iSec)** \( \langle \text{hd}, \text{tl} \rangle^* \text{rec} = i. \)

2.4.7. **Promotability.** Let us determine when \( \tilde{f}^* \psi; g = \tilde{f}^* \psi \) for the left objects. By uniqueness this can be obtained if

\[
\alpha; \langle \tilde{f}^* \psi; g, i_x \rangle^* G = \langle \tilde{f}^* \psi; g, i_x \rangle^* F; \tilde{f}^* \]

which, since \( G, F \) are functors, is equivalent to

\[
\alpha; \langle \tilde{f}^* \psi, i_x \rangle^* G; \langle g, i_x \rangle^* G = \langle \tilde{f}^* \psi, i_x \rangle^* F; \langle g, i_x \rangle^* F; \tilde{f}^* \]

which by the defining diagram is equivalent to

\[
\langle \tilde{f}^* \psi, i_x \rangle^* F; f; \langle g, i_x \rangle^* G = \langle \tilde{f}^* \psi, i_x \rangle^* F; \langle g, i_x \rangle^* F; \tilde{f}^* \]

So we get the equation

- **(Lprom)** \( \text{if } \tilde{f}; \langle g, i_x \rangle^* G = \langle g, i_x \rangle^* F; \tilde{f}^* \text{then } \tilde{f}^* \psi; g = \tilde{f}^* \psi. \)

In particular, if \( F \) does not depend on \( g \), then

- **(Liprom)** \( \tilde{f}^* \psi; g = (\tilde{f}; \langle g, i_x \rangle^* G)^* \psi. \)
Dually we have

\[ (R_{\text{prom}}) \quad \text{if} \quad \langle g, i_x \rangle \cdot F; f = f' ; \langle g, i_x \rangle \cdot G \quad \text{then} \quad g ; f' \cdot \psi = f \cdot \psi, \]

\[ (R_{\text{iprom}}) \quad g ; f \cdot \psi = (\langle g, i_x \rangle \cdot G ; f') \cdot \psi \quad (\text{if} \ G \ \text{does not depend on} \ g). \]

Similar equations for the restricted (co-)algebraic cases are well-known under the name promotability [15]. However, using dialgebras we can present these equations in a very general and yet simple manner.

2.4.8. Example. For the product and stream examples, we get

\[ (\text{prod}_{\text{iprom}}) \quad g ; \langle f_1, f_2 \rangle \cdot \text{pair} = \langle g ; f_1, g ; f_2 \rangle \cdot \text{pair}, \]

\[ (\text{stream}_{\text{prom}}) \quad \text{if} \quad g ; f_2 = f'_2 ; g \quad \text{then} \quad g ; \langle f_1, f_2 \rangle \cdot \text{srec} = \langle g ; f_1, f'_2 \rangle \cdot \text{srec}. \]

The (iprom) rule is better known as distributivity and will be used as a reduction rule in CPL, next section. The general (prom) rule may, e.g., be used to optimize CPL programs.

3. CPL

In parallel to category specifications we first define categorical combinator languages, and then CPL is defined as a categorical combinator language with a dialgebra based uniform scheme for introducing new types with constructors and destructors.

3.1. Categorical combinator languages

Given a category specification (or theory), a programming language may be derived. The principles to be followed are:

- Programs are terms in \( A_2 \) and are also called combinators, we call this the programs as arrows paradigm;
- Types are terms in \( \Phi \), this association gives us functorial polymorphism;
- Reduction rules are based on directed equations; we may use all or some of those in \( \Xi \) or some derived equations.

We now spell out in detail the common core of categorical combinator languages as we see them. A category specification \( C = ((\Phi, \Psi), \Xi) \) is assumed.

3.1.1. Syntax. Programs are terms generated by the grammar

\[ \text{Exp:} \quad e ::= 1 \mid e + e \mid \psi(e_1, \ldots, e_m) \mid F(e_1, \ldots, e_n). \]

The admissible terms (those of \( A_2 \)) are restricted via type rules.
3.1.2. The type of a term $e$ is a pair $(K, K')$ of (closed) functorial expressions, called source and target; notation $e : K \to K'$. The type is defined relative to the signature $(\Phi, \Psi)$ which is implicit in the rules. The set of types is preordered by the more general than ordering defined now.

3.1.3. Definition. A type $K \to K'$ is more general than type $K'' \to K'''$ if $K$ and $K'$ are more general than $K''$ and $K'''$, respectively. A functorial expression $K = \lambda X_1, \ldots, X_n. E$ is more general than $K'$ if there are functorial expressions $K_1, \ldots, K_n$ such that $[K_1, \ldots, K_n] = K'$.

Up to $\alpha$-conversion and the number of variables unused in $E$, the more general than ordering has a top element, the most general type. If $K$ is more general than $K'$, then $K_1, \ldots, K_n$ can be formed using standard methods of unification.

3.1.4. Type rules. In the rules (comp) and (arr), $K_1, \ldots, K_n$ and $K_1', \ldots, K_n'$ are found by unification.

(id) \hspace{1cm} 1 : K \to K

\hspace{3cm} e_1 : K \to K'

(comp) \hspace{1cm} e_2 : K' \to K''

\hspace{3cm} e_2 \circ e_1 : K \to K''

\hspace{1cm} F \in \Phi_1, \ldots, \Phi_n, \text{ and}

for $i = 1, \ldots, n$ \hspace{1cm} \begin{cases} e_i : K_i \to K_i' & \text{if } v_i = \perp \text{ or } v_i = +, \\ e_i : K_i \to K_i & \text{if } v_i = -, \\ 1_i : K_i \to K_i & \text{if } v_i = \top, \end{cases}

(func) \hspace{1cm} F(e_1, \ldots, e_n) : F[K_1, \ldots, K_n] \to F[K'_1, \ldots, K'_n]

\hspace{1cm} \psi \in \Psi_{(\langle K_1, K'_1 \rangle, \ldots, \langle K_n, K'_n \rangle, \langle K, K' \rangle)} \text{, and}

\hspace{3cm} \text{for } i = 1, \ldots, m : \hspace{1cm} e_j : K_j[K'_1, \ldots, K'_n] \to K_j'[K_1, \ldots, K_n]

\hspace{3cm} \psi(e_1, \ldots, e_n) : K'[K'_1, \ldots, K'_n] \to K[K_1, \ldots, K_n]

3.1.5. Examples. Assuming the specification for cartesian closed categories the following types may be derived:

$1 : \lambda X. X \to \lambda X. X$

$\text{cur}(\text{p2}) : \lambda ABC.A \to \lambda ABC. \exp(B, C)$

$\text{cur}(\text{cur}(\text{app opair}(\text{p2 op1}, \text{app opair}(\text{p2 op1}, \text{p2}))))$

$\lambda X Y. Y \to \lambda X Y. \exp(\exp(X, X), \exp(X, X))$
The last example is an encoding of the function \( \text{twice} \equiv \lambda f : X \to X. x^X \cdot f(x) \); compare the translation in Section 5.

Terms in \( A_2 \) are seen as programs; for input and output a non-empty subset \( V \) of \( A_2 \) of values \( v, v' \) is used. Reduction rules can now be defined.

### 3.1.6. Reduction rules

Let \( C = (\Phi, \Psi) \) be a category specification and \( V \subseteq A_2 \) a set of values. The relationship

\[
e \bullet v \rightarrow v'
\]

expresses that program \( e \) applied to input \( v \) reduces to output \( v' \). If the most general typings of \( e, v, \) and \( v' \) are \( e : K_1 \to K'_1 \), \( v : K_2 \to K'_2 \) and \( v' : K_3 \to K'_3 \), then the following must hold.

- \( e \in A_2 \) and \( v, v' \in V \).
- \( K_1 \) is more general than \( K'_2 \) (\( e \) and input \( v \) are related, and \( e \) uses properties possessed by \( v \)).
- \( K'_1 \) is more general than \( K'_3 \) (output \( v' \) possesses at least those properties promised by \( e \)).
- \( e \circ v = v' \) is satisfied in all models of \( C \) (the set of reduction rules is a subset of the equational theory \( \Xi \)).
- The judgement contains the following two basic rules:

\[
\begin{align*}
(id) & \quad e \bullet v \rightarrow v' \\
(comp) & \quad e_2 \circ v' \rightarrow v'' \\
& \quad e_2 \circ e_1 \bullet v \rightarrow v''
\end{align*}
\]

(the category axioms are the common core of categorical programming languages).

Note that by this definition the derived programming languages are in general weaker than the original category specification in the sense that the equivalence closure of the reduction rule (i.e., if \( e \bullet v \rightarrow v' \) in the reduction relation, then \( e \circ v = v' \) is an equation, and such equations are closed under symmetry and transitivity) yields a weaker theory than that of \( \Xi \).

### 3.2. Definition of CPL

We obtain \( \text{CPL} \) by adding a dialgebra based declaration scheme to the above definition of categorical combinator language.

Initially, the theory \( (\Sigma, \Xi) \) is empty, i.e., \( \Phi \) and \( \Psi \) are both empty, and the only programs are \( e ::= 1 \mid e \circ e \) (which all reduce to 1). The theory is extended via a series of (restricted) dialgebra declarations.
3.2.1. **Declarations.** A declaration has one of the two forms:

- **Left object** $L(\tilde{X})$ with $\psi_L$ is
- $a_{L,1} : E_{L,1} \rightarrow L$
- ...
- $a_{L,m} : E_{L,m} \rightarrow L$

- **Right object** $R(\tilde{X})$ with $\psi_R$ is
- $a_{R,1} : E_{R,1} \rightarrow \hat{E}_{R,1}$
- ...
- $a_{R,m} : E_{R,m} \rightarrow \hat{E}_{R,m}$

with the additional restrictions:
- $L$ and $R$ may only occur covariantly in the $E$s and $R$
- $E_{R,1}, \ldots, E_{R,m}$ must be productive in $R$, see below.

In effect $((\Phi, \Psi), \Xi)$ is extended with names and equations as defined in Section 2.

Generally speaking, the right objects are robust enough to define even exponential objects, and the left objects are restricted enough so that the theory does not collapse.

A functorial expression $E$ is productive in $X$ if in models having products $E$ is equivalent to $X \ast E'$ for some other functorial expression $E'$. The formal definition is the following.

3.2.2. **Productive.** Assume a list of dialgebra declarations. **Productive** functorial expressions $E$ in $X$ are defined inductively as follows:
- $X$ is productive in $X$.
- $R(\tilde{X}_1, \ldots, \tilde{X}_n)$ is productive in $X$ if
  - $X$ occurs in precisely one of $E_1, \ldots, E_n$, say $E_j$,
  - $E_j$ is productive in $X$, and
  - $R$ is declared as an unconditioned right object $R(\tilde{X}_1, \ldots, \tilde{X}_n)$ with $\psi_R$ is
    - $a_{R,1} : E_{R,1} \rightarrow \hat{E}_{R,1}$
    - ...
    - $a_{R,i} : R \rightarrow \hat{E}_{R,i}$
    - ...
    - $a_{R,m} : E_{R,m} \rightarrow \hat{E}_{R,m}$

    where for precisely one $i \in \{1, \ldots, m\}$ the parameter $X_j$ occurs in $\hat{E}_{R,i}$ which must be productive in $X_j$.

$R$ is **unconditioned** if $R$ does not occur in $\hat{E}_{R,1}, \ldots, \hat{E}_{R,m}$. The equations of an unconditioned right object have no conditions; compare $(u_L)$ and $(u_R)$ of Section 2.4.
3.2.3. Canonical expressions. Expressions are generated by the grammar

\[
\text{Exp: } e ::= 1 \mid e \circ e \mid \varepsilon \mid \psi(e_1, \ldots, e_m) \mid F(e_1, \ldots, e_n).
\]

Canonical expressions are expressions where destructors only occur within constructors of right objects (i.e., in arguments to right factorizers which are kind of “lazy”):

\[
\text{CanExp: } e ::= 1 \mid \alpha_L \circ e \mid \psi_R(e_1, \ldots, e_m) \circ e.
\]

With this definition left values are *eager* and right values are *lazy*, a viewpoint which is also present in [8].

3.2.4. Values are a bit more restricted (eager):

\[
\text{val: } v ::= 1 \mid \alpha_L \circ v \mid \psi_R(e_1, \ldots, e_m) \circ v \mid \psi_U(q_1, \ldots, q_m)
\]

where \(\psi_U\) is the factorizer of an unconditioned right object, and where for \(i = 1, \ldots, m\) we (syntactically) define

\[
q_i = \begin{cases} 
\text{an expression } e & \text{if } E_{U,i} \neq U, \\
\text{a value } v & \text{otherwise}.
\end{cases}
\]

In addition to the reduction rules (id) for identity and (comp) for composition, there are reduction rules obtained from (L), (R), (L\text{def}), (R\text{def}) and (R\text{prom}) (cf. Section 2.4):

3.2.5. Right object reductions. In the rule R\text{-fact}, \(R\) must be recursive.

\[
(\text{R\text{fun}}) \quad \psi_R([1/R, e_1/X_1, \ldots, e_n/X_n]E_{R,j} \circ \alpha_{R,j} \\
\circ [1/R, e_1/X_1, \ldots, e_n/X_n]E_{R,j}, j = 1, \ldots, m) \bullet v \rightsquigarrow v'
\]

\[
\frac{R(e_1, \ldots, e_n) \bullet v \rightsquigarrow v'}{(\text{R\text{fun}})}
\]

\[
(\text{R\text{fact}}) \quad \psi_R(e_1, \ldots, e_n) \bullet v \rightsquigarrow \psi_R(e_1, \ldots, e_m) \circ v
\]

For \(j = 1, \ldots, m\),

\[
(\text{U\text{fact}}) \quad e_j \bullet v \rightsquigarrow e'_j \quad \text{if } E_{U,j} = U
\]

\[
\frac{e'_j = e_j \circ [v/U, 1/X_1, \ldots, 1/X_n]E_{U,j} \circ e_j \bullet v \rightsquigarrow v'}{\psi_U(e_1, \ldots, e_m) \bullet v \rightsquigarrow \psi_U(e_1, \ldots, e_m)}
\]

\[
(\text{R\text{nat}}) \quad [\psi_R(e_1, \ldots, e_m)/R, 1/X_1, \ldots, 1/X_n]E_{R,j} \circ e_j \bullet v(1) \rightsquigarrow v'
\]

\[
\frac{x_{R,j} \bullet v(\psi_R(e_1, \ldots, e_m)) \rightsquigarrow \text{t}_{\text{CPL}} v'}{x_{R,j} \bullet v(\psi_R(e_1, \ldots, e_m)) \rightsquigarrow \text{t}_{\text{CPL}} v'}
\]

where \(v(\ )\) is a value term with one placeholder; see 3.2.7.

Note that (U\text{fact}) is a simple example of promotability.
3.2.6. Left object reductions.

\[ (L_{\text{fun}}) \quad \psi_L(\alpha_{L,j} \circ [1/L, e_1/X_1, \ldots, e_n/X_n] E_{L,j}, j = 1, \ldots, m) \bullet v \mapsto v' \]

\[ L(e_1, \ldots, e_n) \bullet v \mapsto v' \]

\[ (L_{\text{fact}}) \quad e_j \circ [\psi_L(e_1, \ldots, e_m)/L, 1/X_1, \ldots, 1/X_n] E_{L,j} \bullet v \mapsto v' \]

\[ \psi_L(e_1, \ldots, e_m) \bullet \alpha_{L,j} \circ v \mapsto v' \]

\[ (L_{\text{nat}}) \quad \alpha_L \bullet v \mapsto \alpha_L \circ v. \]

3.2.7. The shape of \( v( ) \) in \( (R_{\text{nat}}) \) is determined by \( E_{R,j} \) where \( \alpha_{R,j}: E_{R,j} \rightarrow \tilde{E}_{R,j} \) and \( F_{R,j} \) is productive in \( R \). \( \psi_R(e_1, \ldots, e_m) \) will appear exactly once in \( v(\psi_R(e_1, \ldots, e_m)) \); thus we can easily pick it out and replace it by \( 1 \). However, to ease comparison with reductions of inductive types (Section 4) we show how to pick out \( \psi_R(e_1, \ldots, e_m) \) by induction on \( E_{R,j} \), and how to replace it with an \( 1 \). We obtain

\[ \psi_R(e_1, \ldots, e_m) = E_{R,j} \cdot \theta^R_{\psi_R(e_1, \ldots, e_m)} \]

where \( \theta^R \) is defined as

\[ R^* \cdot \theta^R_{\psi_R} = \psi_R(e_1, \ldots, e_m), \]

\[ P(F_1, \ldots, F_{m'}) \cdot \theta^R_{\psi_R} \equiv ([E_i/X_i] \tilde{E}_{P,j}) \cdot \theta^R_{\psi_R}, \quad (\ast) \]

and we obtain

\[ v(1) = E_{R,j} \cdot \eta^R_{\psi_R(e_1, \ldots, e_m)} \]

where \( \eta^R \) is defined as

\[ R^* \cdot \eta^R_{\psi_R} = i, \]

\[ P(F_1, \ldots, F_{m'}) \cdot \eta^R_{\psi_R} \equiv \psi_P(e_1, \ldots, ([E_i/X_i] \tilde{E}_{P,j}) \cdot \eta^R_{\psi_R}, \ldots, e_m), \quad (\ast) \]

where the condition \( (\ast) \) is

\[ R \in E_i, \quad P \text{ productive in } X_i, \quad \alpha_{P,j}: P \rightarrow \tilde{E}_{P,j} \quad \text{and} \quad X_i \in \tilde{E}_{P,j}. \]

Declaration examples

In the rules \( (R_{\text{fun}}, L_{\text{fun}}) \), functors are replaced by factorizers, and we have simplified the expressions by removing identity arrows like \( i \) and \( \text{pair}(p1, p2) \). We can always perform such optimizations which are valid in any model.

3.2.8. Terminal object.

right object \( 1( ) \) with \( 1 \) is

end

\[ 1( ) \mapsto 1( ) \]

\[ ! ( ) \bullet v \mapsto 1( ) \]
3.2.9. Product.

right object \( \text{Prod}(A, B) \) with \( \text{pair} \) is

\[
\begin{align*}
\text{pl} : & \text{Prod} \to A \\
\text{p2} : & \text{Prod} \to B \\
\end{align*}
\]

\[
\text{pair}(e_1 \circ \text{pl}, e_2 \circ \text{p2}) \cdot v \rightsquigarrow v' \\
\text{Prod}(e_1, e_2) \cdot v \rightsquigarrow v'
\]

Please note that the premises in (p1, p2) are superfluous. The product is unconditioned with \( E_{\text{Prod}, 1} = E_{\text{Prod}, 2} = \text{Prod} \). Thus, \( \text{pair} \) is an \textit{eager} constructor, and \( v_1, v_2 \) are already in normal form. See (app) for an example of the opposite.

3.2.10. Exponential object.

right object \( \text{Exp}(A, B) \) with \( \text{cur} \) is

\[
\begin{align*}
\text{app} : & \text{Prod}(\text{Exp}, A) \to B \\
\end{align*}
\]

\[
\text{cur}(e_2 \circ \text{app} \circ \text{pair}(\text{pl}, e_1 \circ \text{p2})) \cdot v \rightsquigarrow v' \\
\text{Exp}(e_1, e_2) \cdot v \rightsquigarrow v'
\]

The premise in (app) is not superfluous. The exponential is unconditioned, but \( E_{\text{Exp}, 1} \) is not simply \( \text{Exp} \), so the argument to \( \text{cur} \) is \textit{lazy} and in general unevaluated. The rule (cur) is equivalent to

\[
\text{cur}(e) \cdot v \rightsquigarrow \text{cur}(e \circ \text{pair}(v \circ \text{pl}, \text{p2})).
\]

Note that recursive right objects are lazy, too.
3.2.11. Natural numbers.

left object $N(\ )$ with pr is

$\begin{align*}
  z &: 1 \rightarrow N \\
  s &: N \rightarrow N
\end{align*}$

end

$\begin{align*}
  (nat) &\quad pr(z \circ !(\ ), succ) \circ v \rightarrow v' \\
  &\quad N(\ ) \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (pr_z) &\quad e_1 \circ !(\ ) \circ v \rightarrow v' \\
  &\quad pr(e_1, e_2) \circ z \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (pr_s) &\quad e_2 \circ pr(e_1, e_2) \circ v \rightarrow v' \\
  &\quad pr(e_1, e_2) \circ s \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (z) &\quad z \circ v \rightarrow z \circ v' \\
  (s) &\quad s \circ v \rightarrow s \circ v
\end{align*}$


left object $tree(X)$ with itree is

$\begin{align*}
  \text{nil} &: 1 \rightarrow \text{tree} \\
  \text{forest} &: \text{Exp}(X, \text{tree}) \rightarrow \text{tree}
\end{align*}$

end

The branches of a tree are determined by the parameter type $X$. In the category Set with $|X|=2$, we get binary trees.

$\begin{align*}
  (tree) &\quad \text{itree}(\text{nil} \circ !(\ ), \text{forest} \circ \text{cur}(\text{app} \circ \text{pair}(\text{pl}, \text{eop}2))) \circ v \rightarrow v' \\
  &\quad \text{tree}(e) \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (it_{nil}) &\quad e_1 \circ !(\ ) \circ v \rightarrow v' \\
  &\quad \text{itree}(e_1, e_2) \circ \text{nil} \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (it_{for}) &\quad e_2 \circ \text{cur}(\text{itree}(e_1, e_2) \circ \text{app}) \circ v \rightarrow v' \\
  &\quad \text{itree}(e_1, e_2) \circ \text{forest} \circ v \rightarrow v'
\end{align*}$

$\begin{align*}
  (nil) &\quad \text{nil} \circ v \rightarrow \text{nil} \circ v' \\
  (\text{for}) &\quad \text{forest} \circ v \rightarrow \text{forest} \circ v
\end{align*}$

Program examples

CPL is strong enough to define primitive recursion over the natural numbers [10]. The construction is well-known for cartesian closed categories with a natural numbers
Comparing Hagino’s CPL and typed λ-calculi

object [14]. Here the emphasis is on an operational understanding of the definition. First the simpler case of defining a predecessor is explained. Then a definition of primitive recursion is given and explained in detail, and some simple examples are listed.

3.2.13. Mixfix notation. Following common practice we use the mixfix notation:

\[
\begin{align*}
\text{Prod}(A, B) & \quad \text{pair}(f, g) \\
A \times B & \quad \langle f, g \rangle \\
\text{Exp}(B, C) & \quad B \rightarrow C
\end{align*}
\]

which will be assumed in the rest of the report.

3.2.14. Predecessor. As a first step to explain the primitive recursion, we define a CPL program for the predecessor

\[
pred 0 = 0,
pred (m + 1) = m.
\]

Given a (representation of a) natural number \(n = s \circ \cdots \circ s \circ z\) the only way to destruct it is via the iterator \(pr\), where for \(f : 1 \rightarrow A\) and \(g : A \rightarrow A\) we have

\[
pr(f, g) \circ n = g \circ \cdots \circ g \circ f.
\]

If \(n\) could be \(z\), but then what should \(g\) be? We still need to obtain an expression with one \(s\) “removed”.

The standard solution is to maintain \(n - 1\) and \(n\) in parallel in the reconstruction \(\cdots \circ g \circ f\), i.e., to have a pair of \(n - 1\) and \(n\):

\[
\begin{align*}
f & \equiv \langle z, z \rangle, \\
g & \equiv \langle p2, s \circ p2 \rangle.
\end{align*}
\]

The base is then

\[
pr(f, g) \circ z = \langle z, z \rangle
\]

and the inductive step is

\[
pr(f, g) \circ s \circ n = \langle p2, s \circ p2 \rangle \circ \langle n - 1, n \rangle = \langle n, n + 1 \rangle.
\]

The final definition is

\[
pred = p \circ pr(\langle z, z \rangle, \langle p2, s \circ p2 \rangle).
\]

3.2.15. Standard primitive recursion. The basis is the functions zero, succ, and projections. Moreover, we have composition, and if \(g : A \rightarrow B\) and \(h : \mathcal{N} \rightarrow \mathcal{N} \rightarrow A \rightarrow B\) are primitive recursive functions, then so is \(f : \mathcal{N} \rightarrow A \rightarrow B\) which is defined by the primitive recursion scheme

\[
\begin{align*}
f0a & = ga, \\
f(m + 1)a & = h(fm a) \circ a.
\end{align*}
\]
3.2.16. **CPL primitive recursion.** If \( g: A \rightarrow B \) and \( h: B \ast (N \ast A) \rightarrow B \) are primitive recursive arrows, then so is \( f: N \ast A \rightarrow B \) defined by

\[
\begin{align*}
fo\langle z o i, i \rangle &= g, \\
fo\langle s o p1, p2 \rangle &= h o \langle f, i \rangle.
\end{align*}
\]

This arrow \( f \) is constructed as follows:

\[
\begin{align*}
f &\equiv \text{app}\circ((\text{plo pr}(g', h'))\ast i) \\
&: N \ast A \rightarrow B
\end{align*}
\]

where

\[
\begin{align*}
g' &\equiv \langle \text{cur}(g \circ p2), z \rangle \\
&: 1 \rightarrow (A \Rightarrow B) \ast N, \\
h' &\equiv \langle \text{cur}(h \circ (\text{app}\circ(p1 \ast i), (p2 \ast i))), s o p2 \rangle \\
&: 1 \rightarrow (A \Rightarrow B) \ast N.
\end{align*}
\]

This construction is explained in two steps. First note that

\[
\text{plo pr}(g', h'): N \rightarrow (A \Rightarrow B)
\]

is similar to \( \text{pred} \). In the base case we obtain \( \text{cur}(g \circ p2) \). The \( p2 \) turns \( g: A \rightarrow B \) into a combinator \( N \ast A \rightarrow B \) that does not depend on \( N \). In the inductive case the combinator

\[
h \circ (\text{app}\circ(p1 \ast i), (p2 \ast i)): ((A \Rightarrow B) \ast N \ast A) \rightarrow B
\]

uses an apply and a rebuilding to get something of type \( B \ast (N \ast A) \) which is then processed by \( h \).

Thus, after having built a (huge) "closure" \( A \Rightarrow B \), a final \( \text{app} \) is activated to obtain the desired value of type \( B \).

3.2.17. **Add.** We will define \( add m n \) to be the addition of \( m \) and \( n \).

\[
\begin{align*}
add 0 n &= n, \\
g &= 1, \\
add (m + 1) n &= s(add m n), \\
h &= s o p1.
\end{align*}
\]

In Section 5.2 we shall see another representation of \( add \). Actually, the simplest representation seems to be

\[
\begin{align*}
add &\equiv \text{app}\circ(\text{pr}(\text{cur}(p2), \text{cur}(s o \text{app})) \ast i).
\end{align*}
\]

In [6] the general \( (prom) \) equation is applied to reduce the other \( add \) representations into this one; also, other program examples are given, notably a polymorphic sorting algorithm \( \text{sort}: (A \ast A \Rightarrow \text{Bool}) \ast \text{List}(A) \rightarrow \text{List}(A) \).
4. $\mathcal{F}_1$, the simply typed $\lambda$-calculus

In this section we describe $\mathcal{F}_1$, the simply typed $\lambda$-calculus. $\mathcal{F}_1$ is defined relative to unspecified sets $B$ of base types and $b$ of basic constants. In the next section a scheme—inductive types—for uniformly adding new basic types and constants including booleans and integers is given. Much of the presentation is based on [9] and [16]. See also [1].

We describe the syntax, the type rules, and the reduction rules. Finally the description is extended with an inductive type scheme to $\mathcal{F}_1^1$.

4.1. Syntax. The syntax of $\mathcal{F}_1$-programs is given by

Type: $t ::= B \mid (t \rightarrow t)$

Exp: $e ::= b \mid i \mid (e,e) \mid (\lambda e)$

Index: $i ::= 0 \mid 1 \mid 2 \mid \ldots$

In general, $B$ and $b$ signify base types and constants, respectively. Again, in examples, we use a more informal variable notation, e.g., assuming base type $\mathbb{N}$ and constant $\text{add}$ we may write $\lambda m^n n^2 . \text{add} m n$ for $(\lambda \mathbb{N} ((\lambda \mathbb{N} ((\text{add} 1) 0)))$. As usual, $\rightarrow$ will be right-associative and most parentheses will be omitted, e.g., $(t_1 \rightarrow t_2) \rightarrow t_3 \rightarrow t_4$ for $((t_1 \rightarrow t_2) \rightarrow (t_3 \rightarrow t_4))$.

4.2. Typing. Well-formed $\mathcal{F}_1$-programs are defined to be those that are well-typed according to the rules below. We introduce a type environment

Env: $\Gamma ::= e \mid \Gamma, t$

to represent the types of free de Bruijn indices. We write $\Gamma, i$ to mean the $(i+1)$th element from the right in the sequence $\Gamma$. We introduce the relation

$\Gamma \vdash e : t$

to mean "in the given environment $\Gamma$, the expression $e$ has type $t". Then the type rules are the following:

(ind$_\lambda$) $\Gamma \vdash i : \Gamma, i$

$\Gamma \vdash e_1 : t_1$

(app$_\lambda$) $\Gamma \vdash e_2 : t_1 \rightarrow t_2$

$\Gamma \vdash (e_2 e_1) : t_2$

(abs$_\lambda$) $\Gamma, t_1 \vdash e : t_2$

$\Gamma \vdash (\lambda t_1 e) : t_1 \rightarrow t_2$

These rules yield the unicity of types property: any term has at most one type.
Note that there are no type rules for basic constants; these rules must be stated separately as the constants are introduced.

4.3. Reductions in \( \mathcal{F}_1 \). We have \( \beta \) and \( \eta \) reductions in \( \mathcal{F}_1 \) as in the untyped \( \lambda \)-calculus, the type information is just ignored. Given these reduction rules, \( \mathcal{F}_1 \) can be shown to be terminating; it is even strongly normalising, i.e., any term reduces with any applicable reduction path to a normal form, and Church-Rosser. Thus any term reduces to a unique normal form \([9]\). The reduction rules may also be shown to respect the type rules (subject reduction), i.e., they preserve types of expressions \([1]\).

The relation \( e_1 \rightarrow_{\mathcal{F}_1} e_2 \) means that \( e_1 \) reduces to \( e_2 \) for \( e_1, e_2 \in \mathcal{F}_1 \) and is defined by the rules:

\[
(\beta) \quad ((\lambda e_2) \ e_1) \rightarrow_{\mathcal{F}_1} e_1 \ [e_1 / 0] e_2
\]

\[
(\eta) \quad (\lambda (\langle 1, 0, e \rangle \uparrow \text{lift } \ O)) \rightarrow_{\mathcal{F}_1} e
\]

\( \alpha \beta \eta \)-equivalence of terms \( e_1 \) and \( e_2 \) is denoted \( e_1 \approx e_2 \). In the \( \beta \)-rule, \([e_1 / 0] e_2 \) is the substitution of \( e_1 \) for “free occurrences” of \( 0 \) in \( e_2 \). In the \( \eta \)-rule, \( \langle 1, 0, e \rangle \uparrow \text{lift } \ O \) is \( e \) with all free indices increased by \( 1 \); thus, it is an expression in which \( 0 \) does not occur free. Substitution and lifting are defined below.

4.4 Substitution of \( e_1 \) for free occurrences of \( i \) in \( e_2 \), notation \([e_1 / i] e_2\), is defined by

\[
[e_1 / i] j = \begin{cases} 
  j & \text{if } j < i \quad \text{(the index } i \text{ is bound),} \\
  \langle i, 0, e \rangle \uparrow \text{lift } j = i & \text{(the free variables in } e \text{ must be increased by the number of } \lambda \text{’s that } e \text{ has been moved inside of),} \\
  j - 1 & \text{if } j > i \quad \text{(the free index } j \text{ counts one } \lambda \text{ less);}
\end{cases}
\]

\[ [e_1 / i] (e_2 \ e_1) = ([e_1 / i] e_2 \ [e_1 / i] e_1) \quad \text{(substitute in each branch);} \]

\[ [e_1 / i] (\lambda t e_2) = (\lambda t \ [e_1 / i + 1] e_2) \quad \text{(free indexes count yet a } \lambda \text{).} \]

Note that this definition of substitution does more than just substitute: it also adjusts free variable indices by \(-1\) (case \( j > i \)). This decrementation has been included since substitution in \( \lambda \)-calculi will always be applied in connection with \( \beta \) reduction.

4.5. Lifting. The lifting \( \langle m, n, e \rangle \uparrow \text{lift} \) means that “free variables” \( i \) in \( e \) (i.e., indices greater than or equal to \( n \)) are “lifted” to \( i + m \). Formally the definition is:

\[ \langle m, n, i \rangle \uparrow \text{lift} = \begin{cases} 
  m + i & \text{if } i \geq n \quad \text{(} i \text{ is free),} \\
  i & \text{if } i < n \quad \text{(} i \text{ is bound);}
\end{cases} \]

\[ \langle m, n, (e_2 \ e_1) \rangle \uparrow \text{lift} = (\langle m, n, e_2 \rangle \uparrow \text{lift} \ \langle m, n, e_1 \rangle \uparrow \text{lift} ) \quad \text{(lift each branch),} \]

\[ \langle m, n, (\lambda t e) \rangle \uparrow \text{lift} = (\lambda t \ \langle m, n + 1, e \rangle \uparrow \text{lift} ) \quad \text{(count the } \lambda \text{ and lift the body).} \]
\[ \mathcal{F}_1, \text{ the system } \mathcal{F}_1 \text{ with inductive types.} \]

With no base types \( \mathcal{F}_1 \) is an empty language, closed lambda-abstractions are syntactically impossible. We could add specific types and constants to \( \mathcal{F}_1 \). However, such additions are \textit{ad hoc}; we prefer to add a type-generating method. Thus, we extend \( \mathcal{F}_1 \) with inductive types to get \( \mathcal{F}_1^i \); cf. [16].

4.6. Syntax. An \( \mathcal{F}_1^i \)-program is a (nonrecursive) sequence of declarations of inductive types, followed by an expression. The syntax of \( \mathcal{F}_1^i \)-programs is given by:

\[
\text{Program:}\quad p ::= 1 e \mid 1 p
\]

\[
\text{Indtypedcl:}\quad 1 ::= \text{indtype } T \text{ is}
\]

\[
c_{T,1} : t_{11} \rightarrow \cdots \rightarrow t_{1n_1} \rightarrow T
\]

\[
\quad \cdots
\]

\[
\text{and } c_{T,m} : t_{m1} \rightarrow \cdots \rightarrow t_{mm} \rightarrow T
\]

\[
\text{Type:}\quad t ::= T \mid t \Rightarrow t
\]

\[
\text{Expression:}\quad e ::= i \mid (e e) \mid (\lambda t \theta) \mid c_{T,j} \mid \text{iter } T[ t ]
\]

In the declaration of an inductive type, \( T \) is an identifier naming the type, and the types \( t_{mj} \) have the extra condition that \( T \) may not occur \textit{negatively} in them.

4.7. Positive and negative. Formally, the positive and negative occurrences may be computed via the functions

\[
pos : \text{Type} \rightarrow \wp(\text{Identifier}) \quad \text{and} \quad \neg : \text{Type} \rightarrow \wp(\text{Identifier})
\]

defined via

\[
T^* \pos = \{ T \}, \quad T^* \neg = \emptyset,
\]

\[
(t_1 \rightarrow t_2)^* \pos = t_1^* \pos \cup t_2^* \pos, \quad (t_1 \Rightarrow t_2)^* \neg = t_1^* \pos \cup t_2^* \neg.
\]

Now how the function arrow flips back and forth between negative and positive. The terms \textit{negative} and \textit{positive} come from logic where the equivalence \( t_1 \Rightarrow t_2 = \neg t_1 \lor t_2 \) shows that \( T \) occurs negatively if it is within an odd number of negations.

4.8. Type Rules. To be strictly formal the environment should carry information of declarations of inductive types \( T \), but a fixed set of declarations will be assumed. In addition to the \( \mathcal{F}_1 \) type rules \((\text{ind}, \text{app}, \text{abs})\), \( \mathcal{F}_1^i \) also has the following type rules:

\[(\text{constr})\quad \Gamma \vdash c_{T,i} : (t_{11} \rightarrow (\cdots \rightarrow (t_{in_i} \rightarrow T) \cdots ))\]

\[(\text{iter})\quad \Gamma \vdash \text{iter } T[x] : t_{11}^* \Rightarrow \cdots \Rightarrow t_{in_i}^* \Rightarrow x
\]

\[
\quad \Rightarrow \cdots
\]

\[
\quad \Rightarrow t_{m1}^* \Rightarrow \cdots \Rightarrow t_{mm}^* \Rightarrow x
\]

\[
\quad \Rightarrow T \Rightarrow x
\]
4.9. Reductions. As for \( F_1 \) there are \( \beta \) and \( \eta \) reductions. The substitution in the \( \beta \)-rule is defined similarly to the \( \beta \)-rule for \( F_1 \) except that the new expression forms \( \text{iter} T[V] \) and \( \epsilon \) are returned unaffected. Additionally there are reductions for the constants declared in inductive types:

\[
\alpha_j = \alpha_j \cdot \tilde{t}_{ij} \quad (\text{for } j = 1, \ldots, n_i)
\]

\[
\Gamma \vdash (\text{iter} T[t] e_1 \ldots e_m (\epsilon_{T,i} a_1 \ldots a_n)) \rightarrow e_i (\alpha_i \cdot \tilde{t}_1 \cdot \ldots \cdot \tilde{t}_m)
\]

An explanation of this rule is: the \( e_1, \ldots, e_m \) are destruction “actions” (functions) corresponding to \( \epsilon_{T,1}, \ldots, \epsilon_{T,n} \) and given an argument \( (\epsilon_{T,1} a_1 \ldots a_n) \) the “action” \( e_i \) should be applied to \( a_1 \ldots a_n \). However, \( \text{iter} T[t] \) is a kind of induction, so \( e_i \) is not applied until \( \text{iter} T[t] \) \( e_1 \ldots e_m \) has been recursively applied to subterms of \( a_1 \ldots a_n \) that correspond to occurrences of \( T \) in \( t_{i1}, \ldots, t_{in} \).

For given, fixed expressions \( e_1, \ldots, e_m \) and type \( t \), we define the function

\[
\hat{\cdot} : F_1^1 \text{types} \rightarrow F_1^1 \text{expressions} \rightarrow F_1^1 \text{expressions}
\]

by

\[
a^* \hat{T} \equiv (\text{iter} T[t] e_1 \ldots e_m a)
\]

\[
a^* t_1 \rightarrow \hat{t}_2 = (\lambda x'. (a (x^* \hat{t}_1)))^* \hat{t}_2
\]

\[
a^* U \equiv a \quad (U \text{ is a type variable which is not } T).
\]

Given these reduction rules, \( F_1 \) has the Church–Rosser, strongly normalization and subject reduction properties; these follow since \( F_1 \) can be translated into \( F_2[16] \).

4.10. Natural numbers can be defined as

\[
\text{indtype } \mathbb{N} \text{ is}
\]

\[
\text{zero : } \mathbb{N}
\]

and \( \text{succ} : \mathbb{N} \rightarrow \mathbb{N} \)

The derived computation rules are

\[
(\text{zero}) \quad (\text{iter}\mathbb{N}[t] z s \text{zero}) \rightarrow \mathcal{F}_1^1 z,
\]

\[
(\text{suc}) \quad (\text{iter}\mathbb{N}[t] z s (\text{succ } n)) \rightarrow \mathcal{F}_1^1 (s(\text{iter}\mathbb{N}[t] z s n)).
\]

For example, programs for addition and multiplication are defined by:

\[
\text{add} \equiv \lambda m n. \text{iter}\mathbb{N}\mathbb{N} m \text{ succ } n,
\]

\[
\text{mul} \equiv \lambda m n. \text{iter}\mathbb{N}\mathbb{N} \text{ zero (add m) n}.
\]

4.11. Ackermann’s function. The function defined by

\[
\langle 0, n \rangle^* \text{ack} = n + 1.
\]

\[
\langle m + 1, 0 \rangle^* \text{ack} = \langle m, 1 \rangle^* \text{ack},
\]

\[
\langle m + 1, n + 1 \rangle^* \text{ack} = \langle m, \langle m + 1, 0 \rangle^* \text{ack} \rangle^* \text{ack}.
\]
is called Ackermann’s function and is interesting because it grows faster than any primitive recursive function [12]. Still it is terminating, as is proved by the \( F_1 \) representation

\[
ack = \lambda m n. \text{iter} N [N \to N] \text{succ} \; m, \\
\text{it} = \lambda f \; n. \text{iter} N [N \to (\text{succ} \; 0)] \; f \; (\text{succ} \; n).
\]

Note that \((\ack m n) = (\text{it}^m \text{succ}) \; n\) where \( \text{it} \) is the function which when applied to \( f \) and \( n \) returns \( f^{n+1}(1) \). Consider, for example, \( \ack 2 3 \). The application of \( \ack \) applied to \( 2 \equiv (\text{succ} (\text{succ} \; 0)) \) returns \((\text{it} \; (\text{it} \; \text{succ}))\), and \((\text{it} \; (\text{it} \; \text{succ}))\) applied to \( 3 \equiv (\text{succ} (\text{succ} (\text{succ} \; 0))) \) returns \(((\text{it} \; \text{succ})^4 \; (1)) \) which reduces to 9.

4.12. Branching trees. We define branching trees over a type \( U \), i.e., trees whose “node arity” or “degree” are given by the “index set” \( U \).

\[
\text{indtype} \; \text{Tree} \; U \; \text{is} \; \\
\text{nil} : \text{Tree} \; U \; \\
\text{and forest} : (U \to \text{Tree} \; U) \to \text{Tree} \; U
\]

The derived computation rules are

\[
(\text{nil}) \quad (\text{iterTree} \; U \; t \; n \; \text{nil}) \mapsto_{F_1} n, \\
(\text{forest}) \quad (\text{iterTree} \; U \; t \; n \; (\text{forest} \; g)) \\
\quad \mapsto_{F_1} (f (\lambda U (\text{iterTree} \; U \; t \; n \; (g \; 0)))).
\]

5. Embedding simply typed lambda calculi in CPL

We compare CPL to \( F_1 \), the simply typed \( \_ \)-calculus with unspecified base types. In CPL the types unit \( 1 \), product \( A \times B \) and function \( A \Rightarrow B \) can be defined because CPL is built on a cartesian closed category (CCC). Simply typed \( \_ \)-calculi correspond closely to CCCs [13,5]. However, CPL has deterministic reduction rules, in contrast to the non-deterministic conversion rules of \( F_1 \). Also, not all CCC equations are used as reduction rules in CPL. Specifically, CPL cannot simulate \( \eta \)-reduction, and \( \beta \)-reduction is fully simulated only within terms of first-order type. In Section 5.1 we give a translation of \( F_1 \) into CPL.

In [16] inductive types with iterators are added to \( F_1 \), producing \( F_1^1 \). In Section 5.2 the \( F_1 \)-to-CPL translation is extended to \( F_1^1 \). The difficulty here is to treat free variables in arguments to \( F_1^1 \)-iterators which are curried whereas CPL-iterators at the syntactical level are uncurried. As an application, Ackermann’s function is encoded in CPL which shows that CPL is more than primitive recursive. Another result is that CPL can represent all functions \( \N \to \N \) provably total in first-order arithmetic.
As shown in the section on CPL reduction rules, CPL contains a cartesian closed structure. This fact allows us to get a natural translation of \( \mathcal{F} \) into CPL following, for example, [5]. In what follows CPL\(_{ccc}\) denotes CPL with declared products and exponentials.

Whenever a new type is declared in CPL, its semantics is based on a corresponding initial or final dialgebra. This dialgebra entails a number of equations, some of which are used in restricted form as rewrite rules in CPL. The equations of CPL\(_{ccc}\)— see Fig. 11—are precisely those of CCL\(\beta\) in [5, p. 25] where it is shown that they can simulate a \(\beta\)-theory. In order to simulate \(\eta\), too, the uniqueness equation for curry must be added. However, as we shall see in the section on reduction preservation, the rewrite rules in CPL corresponding to a CCC yield an equivalence (and theory) weaker than CCL\(\beta\).

5.1. Translation of \( \mathcal{F} \) into CPL.

The translation of typed \(\lambda\)-calculi into CCCs is well established; see, for example, [5]. For the presentation given in Fig. 12 we were inspired by Philip Wadler. We first state how to translate the expressions, then correctness is discussed, and finally some intuition about the translation of reduction rules is given.

5.1.1. Expressions. For any \( \mathcal{F} \), expression, type, or environment \( X \), let \( \bar{X} \) denote its representation in CPL. The general translation is guided by the translation of the type relation

\[
\Gamma \vdash e : t
\]

which becomes the arrow

\[
\bar{e} : \bar{\Gamma} \rightarrow \bar{t}.
\]

\[
\begin{align*}
(\text{Ass}) & \quad (e_1 \circ e_2) \circ e_3 = e_1 \circ (e_2 \circ e_3) \\
(Id) & \quad e \circ 1 = e = 1 \circ e \\
(Fst) & \quad p_1 \circ \langle e_1, e_2 \rangle = e_1 \\
(Snd) & \quad p_2 \circ \langle e_1, e_2 \rangle = e_2 \\
(Dpair) & \quad \langle e_1, e_2 \rangle \circ e_3 = \langle e_1 \circ e_3, e_2 \circ e_3 \rangle \\
(Beta) & \quad app \circ \langle \text{cur}(e_1), e_2 \rangle = e_1 \circ \langle i, e_2 \rangle \\
(Dcur) & \quad \text{cur}(e_1) \circ e_2 = \text{cur}(e_1 \circ \langle e_2 \circ p_1, p_2 \rangle)
\end{align*}
\]

Fig. 11. CCC equations used by CPL. The same as CCL\(\beta\) in [5, p. 25].
We call such arrows local elements of $\overrightarrow{t}$. If $\Gamma$ is empty — in which case $e$ is closed — then $\overrightarrow{\vec{v}}:1 \rightarrow \overrightarrow{1}$ is a global element or just an element. The translation is given in Fig. 12.

5.1.2. Correctness. The programming languages $\mathcal{F}_1$ and CPL each consists of three parts: a set of terms (syntax), a mapping of terms to types (typing), and a relation of terms to terms (reductions, operational semantics). The translation in Fig. 12 takes care of syntax and typing, and we should check that it respects the reductions. The translation is correct iff

$$e_1 \cong_{\mathcal{F}_1} e_2 \text{ if and only if } \overrightarrow{e_1} \cong_{CPL} \overrightarrow{e_2}$$

where $\cong_{\mathcal{F}_1}$ and $\cong_{CPL}$ are equivalence relations in $\mathcal{F}_1$ and CPL, respectively. The “only if” part says that equivalent terms are mapped to equivalent terms, and the “if” part says that nonequivalent terms are kept distinct. $\cong_{\mathcal{F}_1}$ is $\alpha\beta\eta$-convertibility whereas $\cong_{CPL}$ is operational equivalence (to be defined in 5.1.5).

5.1.3. Reductions. In $\mathcal{F}_1$ we have $\beta$ and $\eta$ rules with auxiliary definitions of substitution and lifting. In order to complete the translation, we should state how these rules are simulated in CPL, and in the next subsection we will discuss this in detail. However, first we give some background and supporting information; we describe to what the rules correspond in the theory CCL$\beta$ underlying CPL$_{ccc}$.

We first consider the translation of the environment $\Gamma$ for a term $e$. In translation $e$ becomes an arrow from the product $\overrightarrow{\Gamma}$; the indices become projections that access parts of the environment; and all other parts of the translation serve to manipulate the

$$\begin{array}{ll}
\text{In } \mathcal{F}_1: & \text{In } CPL: \\
\text{(Type)} & t ::= t_1 \rightarrow t_2 \\
\text{(Env)} & \Gamma ::= \varepsilon \mid \Gamma, t \\
\text{(Ind)} & \Gamma \vdash i: \Gamma, i \\
\Gamma \vdash e_1:t_1 & \overrightarrow{e_1} : \overrightarrow{\Gamma} \rightarrow \overrightarrow{1} \\
\Gamma \vdash e_2:t_1 \rightarrow t_2 & \overrightarrow{e_2} : \overrightarrow{\Gamma} \rightarrow \overrightarrow{t_1} \Rightarrow \overrightarrow{t_2} \\
\Gamma \vdash (e_2,e_1):t_2 & \text{app o } \overrightarrow{(e_2,e_1)} : \overrightarrow{\Gamma} \rightarrow \overrightarrow{t_2} \\
\Gamma, t_1 \vdash e:t_2 & \overrightarrow{\vec{v}} : \overrightarrow{\Gamma} \rightarrow \overrightarrow{t_1} \Rightarrow \overrightarrow{t_2} \\
\Gamma \vdash (\lambda t_1 e):t_1 \rightarrow t_2 & \text{cur} (\overrightarrow{\vec{v}}) : \overrightarrow{\Gamma} \rightarrow \overrightarrow{t_1} \Rightarrow \overrightarrow{t_2}
\end{array}$$

Fig. 12. Translation of $\mathcal{F}_1$ to CPL.
environment so that these indices access the appropriate parts. Consider the environment
\[ F = (\cdots ((1 \ast e_i) \ast \cdots e_i) \ast \cdots) \ast e_0. \]

The standard operations we would like to do are access a value, insert a value, change a value, and remove a value:
- The access of \( e_i \) is simply given by the translation of the index \( i \), i.e., the projection \( \Psi \circ p_1 \).
- How is a value \( v : F \rightarrow V \) inserted between \( e_{i-1} \) and \( e_i \)? The insertion of \( v \) rightmost in \( F \) (below \( e_0 \)) is done by \( \langle 1, v \rangle : F \rightarrow F \ast V \). To insert \( v \) between \( e_0 \) and \( e_1 \), it is to insert \( v \) below \( e_1 \) and then pair with \( e_0 \); this is done by \( \langle \langle 1, v \rangle \circ p_1, p_2 \rangle \). In order to generalize this, we define

\[ P^0(e) = e, \]

\[ P^{n+1}(e) = \langle P^n(e) \circ p_1, p_2 \rangle, \]

where for \( e : A \rightarrow B \),

\[ P^n(e) : (\cdots ((A \ast X_{n-1}) \ast \cdots) \ast X_0 \rightarrow \cdots ((B \ast X_{n-1}) \ast \cdots) \ast X_0. \]

We can say that \( P^n(e) \) preserves the \( n \) lowest places in \( F \) (i.e., \( e_0 \ldots e_{n-1} \)) and transforms the rest of the environment (i.e., \( e_n \ldots e_1 \)). The required insertion operation becomes \( P^i(\langle 1, v \rangle) \).
- \( \langle p_1, v \rangle \) changes \( e_0 \) to \( v \), and to change \( e_i \) we use \( P^i(\langle p_1, v \rangle) \).
- \( p_1 \) removes \( e_0 \), and to remove \( e_i \) we use \( P^i(p_1) \).

Consider \[ [e/i] e' \], i.e., the substitution of the term \( e \) for the index \( i \) in \( e' \) while decreasing free indices by 1. In the translation, instead of decreasing indices we will extend the environment. Thus \[ [e/i] e' \] simply becomes \( \overline{e'} \circ P^i(\langle 1, e \rangle) \).

Consider \[ \langle m, n, e \rangle \ast \text{lift} \], i.e., the lifting of “free variables” \( i \) in \( e \) (i.e., indices greater than or equal to \( n \)) to \( i + m \). We could lift the index \( i \) by translating \( i \) to \( \Psi \circ p_1^{i+m} \). However, we can instead change the environment. If we want the index 0 in \( e \) to refer to the \( m \)th place, we can transform the environment appropriately, obtaining \( \overline{e} \circ p_1^m \). So to let indices \( i \) greater than \( n-1 \) refer to \( i+m \), we make a translation to \( \overline{e} \circ P^m(p_1^m) \). We note that this is the same as removing the places \( e_n, \ldots, e_{n+m-1} \).

Once we understand the manipulation of the environment, it is easy to see that the \( \beta \) rule
\[ (\lambda t (e') e) \rightarrow_{\beta} [e/0] e' \]
corresponds to the equation (\( \beta \)) in Fig. 1. Note that
\[ \overline{[e/0]} e' = \overline{e'} \circ P^0(\langle 1, e \rangle) = \overline{e'} \circ \langle 1, e \rangle. \]

Finally, the \( \eta \) rule
\[ (\lambda t (\langle 1, 0, e \rangle \ast \text{lift 0})) \rightarrow_{\eta} e \]
does not correspond to any equation in $CCL\beta$. However, it does correspond to the uniqueness equation

$\text{(unicur)} \quad \text{cur}(\text{app } o \langle e \circ p1, p2 \rangle) = e$

from the dialgebra underlying the exponential object. Note that

$$\langle 1, 0, e \rangle^{\text{lift}} = e \circ P^0(p1 \downarrow) = e \circ p1.$$

### Reduction preservation

In the previous section we saw that the $\beta$ reduction corresponds to the equations underlying $CPL$. However, the $CPL$ reduction rules are not quite the same as the $CCL\beta$ equations: the equations have become directed (as rewrite rules), and they are applied in a certain deterministic order. Thus, it is not obvious that the translation is correct.

Actually the translation is not strictly correct: as noted there is no equation corresponding to $\eta$-reduction (in [5] the rule $\text{unicur}$ is derived from $CCL\rho\alpha\sigma\rho\delta\eta\psi\varphi$ which is shown to be equivalent to $\mathcal{T}_1$). Moreover, we do not believe that full $\beta$-reduction can be represented in $CPL$:

#### 5.1.4. Conjecture

There is no translation of $\mathcal{T}_1$ to $CPL$ that preserves $\beta$-reduction (let alone $\eta$-reduction).

What we shall do is prove $\beta$-reduction for a restricted subpart of $\mathcal{T}_1$. The restriction we shall make is to exclude higher-order types from the possible result values. The structure of this proof of reduction preservation is close to the equivalence proof in [5]. We shall use $\simeq$ as a shorthand for $\cong_{CPL}$.

#### 5.1.5. Definition

$CPL$ programs $e_1 : K \rightarrow K'$ and $e_2 : K \rightarrow K'$ are equivalent, notation $e_1 \cong e_2$, iff

$$e_1 \circ v \rightarrow v' \Leftrightarrow e_2 \circ v \rightarrow v' \quad (\text{for all } v, v' \in V).$$

In other words, $e_1$ and $e_2$ are equivalent if and only if their compositions with arbitrary compatible values produce equal results. It is relatively easy to see that $\cong$ is in fact an equivalence relation.

We note that equivalent terms may be substituted for each other if they occur in contexts that are not "lazy", i.e. if they will be reduced by the rules of $CPL$.

#### 5.1.6. Definition

An $\mathcal{T}_1$ type $t$ is first order if $t$ is a base type and if for all constructors $t_1 \rightarrow \cdots \rightarrow t_m \rightarrow t$, all the $t_1, \ldots, t_m$ are first order. An $\mathcal{T}_1$ function $f : t_1 \rightarrow t_2$ is first order if $t_2$ is first order.
Similarly a CPL type \( K \) is first order if for \( K = \lambda X_1 \ldots X_k. E \)
\[
E = \begin{cases} 
L(E_1, \ldots, E_n) & \text{L left object, } E_1, \ldots, E_n \text{ first order,} \\
E_1 \ast E_2 & \text{ } E_1, E_2 \text{ first order.}
\end{cases}
\]
A CPL program \( e : K \rightarrow L \) is first order if \( L = \lambda Y_1 \ldots Y_l. E_1 \rightarrow E_2 \) and \( E_2 \) is first order.

It is easy to see that for \( e : t \in F_1 \), if \( t \) is first order then the normal form of \( e \) does not contain \( \lambda \). Similarly, for \( e : K \rightarrow L \) in CPL, the normal form of \( e \) does not contain cur.

The notion of first order functions and programs captures the idea that whenever the function or program is applied to something the result is first order.

5.1.7. Lemma. If \( F_1 \) base types are translated to first-order left objects, then the translation in Fig. 12 preserves first-orderness.

5.1.8. Proof of Lemma 5.1.7. Let \( t \) be a first-order \( F_1 \) type. By structural induction on \( t \), we see that \( \bar{t} \) is first order in CPL. So if \( \Gamma \vdash e : t' \rightarrow t \) is a first order \( F_1 \) function, then \( \bar{e} : \Gamma \rightarrow t' \rightarrow \bar{t} \) is a first order CPL program.

5.1.9. Theorem. For \( F_1 \) expressions of first-order type, translation is consistent with \( \beta \)-reduction:
\[
\begin{align*}
((\lambda t. e_2) e_1) \cdot v &= \text{app} \circ \text{cur}(e_2) \circ \bar{e_1} \cdot v \quad \text{(App, Abs)} \\
\text{cur}(e_2) \cdot v &\rightarrow \text{cur}(e_2 \circ (v \circ p1, p2)) \\
\bar{e_1} \cdot v &\rightarrow v' \\
\bar{e_2} \circ (v \circ p1, p2) \cdot \bar{1}, v' \quad \text{(comp, pair, app)} \\
\bar{e_2} \cdot \bar{1}, v' \\
[e_1/0] e_2 \cdot v &= \bar{e_2} \circ \bar{1}, \bar{e_1} \cdot v \quad \text{(sub)}
\end{align*}
\]
Comparing Hagino's CPL and typed $\iota$-calculi

\[
\frac{\varphi \cdot v \rightarrow v}{\overline{e_1} \cdot v \rightarrow v'}
\]

\[= \overline{e_2} \cdot \langle v, v' \rangle. \quad (\text{comp, pair})\]

5.1.11. Lemma. We get substitution by

\[(\text{sub}) \quad [e_1/n]e_2 = \overline{e_2} \circ P^n(\langle 1, \overline{e_1} \rangle).\]

5.1.12. Proof of Lemma 5.1.11. More precisely, with this definition the rules defining substitution are preserved by the translation. A related proof is given in [5, p. 31]. We redo it to check that only CPL equations are used and also the missing (easy) details are filled in. The Lemmas 5.1.13 and 5.1.14 are used. First the application (type: $\lambda X. Y. X \rightarrow \lambda X. Y. Y$):

\[
\frac{[e_1/n](e_2 e_3) \cdot v}{= \text{app} \circ \langle \overline{e_2}, \overline{e_3} \rangle \circ P^n(\langle 1, \overline{e_1} \rangle) \cdot v \quad (\text{sub, App})}
\]

\[
\frac{P^n(\langle 1, \overline{e_1} \rangle) \cdot v \rightarrow v_1}{\overline{e_2, e_3} \cdot v \rightarrow v_{2,3}}
\]

\[= \text{app} \cdot \langle v_2, v_3 \rangle. \quad (\text{comp, comp, pair})\]

\[
\frac{[e_1/n]e_2 [e_1/n]e_3 \cdot v}{= \text{app} \circ \langle \overline{e_2} \circ P^n(\langle 1, \overline{e_1} \rangle), \overline{e_3} \circ P^n(\langle 1, \overline{e_1} \rangle) \rangle \cdot v \quad (\text{App, sub, sub})}
\]

\[
\frac{\overline{e_2, e_3} \circ P^n(\langle 1, \overline{e_1} \rangle) \cdot v \rightarrow v_{2,3}}{= \text{app} \cdot \langle v_2, v_3 \rangle. \quad (\text{comp, comp, pair})}
\]

Then the abstraction (type: $\lambda X. Y. Z. X \rightarrow \lambda X. Y. Z. Z \Rightarrow Z$):

\[
\frac{[e_1/n] (\lambda \overline{e_2}) \cdot v}{= \text{cur}(\overline{e_2}) \circ P^n(\langle 1, \overline{e_1} \rangle) \cdot v \quad (\text{sub, Abs})}
\]

\[
\frac{P^n(\langle 1, \overline{e_1} \rangle) \cdot v \rightarrow v'}{\rightarrow \text{cur}(\overline{e_2} \circ \langle v' \circ p1, p2 \rangle) \quad (\text{comp, cur})}
\]
We see that the two sides have different normal forms. However, since they occur within first-order programs, they will at some later point either be discarded (for example, by a projection) or occur in a context

\[ \text{app} \bullet \langle \text{cur}(e), v \rangle = e \bullet \langle i, v \rangle, \]

i.e., the argument \( e \) to \( \text{cur} \) is "unpacked" and reduced to normal form. For the indices we have (type: \( \lambda X Y Z. X \rightarrow \lambda X Y Z. Y \Rightarrow Z \)):

\[
\begin{align*}
[\frac{e_1}{m}]n \bullet v &= p2 \circ p1^n \circ P^m(\langle i, \tilde{e} \rangle) \bullet v. & (\text{sub, Ind})
\end{align*}
\]

We proceed by cases of the relationship between \( m \) and \( n \).

Case \( m < n \):

\[
\begin{align*}
\frac{p2 \circ p1^n(\langle i, \tilde{e} \rangle) \circ p1^m \bullet v}{p1^{n-m} \bullet v \mapsto v'} & \quad \left( p1P, P^0 \right)
\end{align*}
\]

\[
\begin{align*}
1 \bullet v' & \mapsto v' \\
\tilde{e} \bullet v' & \mapsto v''
\end{align*}
\]

\[
\begin{align*}
= p2 \circ p1^{n-m-1} \bullet v'. & \quad (\text{pair, p1})
\end{align*}
\]

\[
\begin{align*}
\frac{n-1 \bullet v}{p1^n \bullet v \mapsto \text{CPL} v'} & \quad (\text{Ind, comp})
\end{align*}
\]

Case \( m = n \):

\[
\begin{align*}
= p2 \circ \langle i, \tilde{e} \rangle \circ p1^n \bullet v & \quad (p1P, P^0)
\end{align*}
\]

\[
\begin{align*}
p2 \circ \langle i, \tilde{e} \rangle & \equiv \tilde{e}
\end{align*}
\]

\[
\begin{align*}
= (\text{lift } m 0 e) \bullet v. & \quad (\text{lift})
\end{align*}
\]

Case \( m > n \):

\[
\begin{align*}
= p2 \circ P^{m-n}(\langle i, \tilde{e} \rangle) \circ p1^n \bullet v & \quad (p1P)
\end{align*}
\]

\[
\begin{align*}
= p2 \circ P^{m-n-1}(\langle i, \tilde{e} \rangle) \circ p1. p2 \circ p1^m \bullet v & \quad (P)
\end{align*}
\]
Comparing Hyguino's CPL and typed λ-calculi

\[
\begin{align*}
\text{p2} \circ \langle P^{m-n-1}(\langle i, e \rangle) \circ p1, \text{p2} \rangle & \cong \text{p2} \\
\nonumber
= \tilde{n} \bullet v. \\
\end{align*}
\]

(Ind)

5.1.13. Lemma.

\((\text{lift} \ m \ n \ e) \equiv \tilde{e} \circ P^n(p1^m).\)

Intuitively, \(P^n(p1^m)\) works on the part of the environment from the \((m+n)\)th place and higher, and it shifts the environment \(m\) places down. Or expressed differently, the places between \(n\) and \(m+n\) are removed.

The proof is similar to the proof of Lemma 5.1.11 and is omitted.

5.1.14. Lemma. (This is similar to Lemma 1.2.16 in [5, p. 31].)

\((p1P) \quad p1^m \circ P^n(e) \cong P^{n-m}(e) \circ p1^m, \quad 0 \leq m \leq n.\)

The relevant type is

\[
\begin{align*}
\lambda \text{ABX1} \ldots \text{Xn}. & (\cdots (A \ast \text{X1}) \ast \cdots) \ast \text{Xn} \\
& \rightarrow \lambda \text{ABX1} \ldots \text{Xn}. & (\cdots (B \ast \text{X1}) \ast \cdots) \ast \text{Xn}.
\end{align*}
\]

5.1.15. Proof of Lemma 5.1.14. Trivial for \(n = 0\). For \(n > 0\), we do induction on \(m\). For \(m = 0\) it is trivial. For \(0 < m < n\), we use \((p1P)\) as the induction hypothesis. After we have proved that

\[(*) \quad p1^{m+1} \circ P^n(e) \cong p1^m \circ P^{n-1}(e) \circ p1\]

it will be clear that by the induction hypothesis, the right-hand side is

\[P^{n-1}(e) \circ p1^m \circ p1 = P^{n-(m+1)}(e) \circ p1^{m+1},\]

which proves the lemma. The proof of \((*)\) is:

\[
\begin{align*}
p1^{m+1} \circ p^n(e) \bullet v \\
= p1^m \circ p1 \circ \langle P^{n-1}(e) \circ p1, \text{p2} \rangle \bullet v \quad (\cdot^n, P) \\
\mid P^{n-1}(e) \circ p1 \bullet v \rightarrow v' \\
p2 \bullet v \rightarrow v'' \\
= p1^m \circ p1 \bullet \langle v', v'' \rangle \quad \text{(comp, pair)} \\
= p1^m \bullet v'. \quad \text{(comp, p1)} \\
p1^{m+1} \circ P^{n-1}(e) \circ p1 \bullet v
\end{align*}
\]
5.1.16. Discussion. If we look back at the place where the first-order type restriction was used, we see that what was needed was to ensure that the arguments of cur are at some time evaluated (or discarded). We may consider a stronger rule for (cur) that reduces the argument of cur. But this would necessitate changes with respect to canonical expressions. Consider the strengthening:

\[
\text{(cur') } \quad \frac{e \bullet \langle v \circ p1, p2 \rangle \rightarrow v'}{\text{cur}(e) \bullet v \rightarrow \text{cur}(v')}.
\]

Here we get the problem that \( \langle v \circ p1, p2 \rangle \) is not a normal form, so something else has to be modified. Since in \( \mathcal{F}_1 \) the body of a \( \lambda \)-expression may be reduced without destroying the strong normalization, we expect a similar reduction will be possible in CPL.

5.2. Translating \( \mathcal{F}_1 \) into CPL

Inductive types are least or initial, and they act like initial dialgebras. Inductive types can be added naturally to \( \mathcal{F}_1 \). We shall give a relatively simple translation scheme extending the translation of \( \mathcal{F}_1 \) to CPL. The most difficult point is the iterators. Intuitively the factorizers must be used in the translation of these. However, the iterators are curried, and their arguments may contain free variables that by \( \beta \)-reduction “obtain” their values over “long” distances. In contrast, factorizers are (syntactically) uncurried, and the extra input is the element to make induction over. The “trick” solution will be similar to primitive recursion, Section 3.2: iteration over the induction parameter is done to build a function which then is applied to the environment storing values of free variables.

5.2.1. The inductive types. We assume an \( \mathcal{F}_1 \) declaration

\[
\text{indtype } L \text{ is}
\]

\[
c_{L,1}: t_{i1} \rightarrow \cdots \rightarrow t_{im} \rightarrow L
\]

\[
\cdots
\]

\[
c_{L,m}: t_{m1} \rightarrow \cdots \rightarrow t_{mn} \rightarrow L
\]

of an inductive type \( L \) defined relative to a set \( \mathcal{A} \) of type names (of previously declared types). We abbreviate \( [\tau/L]t \) as \( t' \) for any subtype term \( t \) occurring in the declaration of \( L \). We define

\[
S_i = t_{j1} \rightarrow \cdots \rightarrow t_{in} \rightarrow L
\]
and have
\[
\text{iter } L[\tau] : S_1' \to \cdots \to S_m' \to L \to \tau.
\]

5.2.2. The goal. We wish to obtain in CPL a corresponding
left object \( L() \) with \( \psi_L \) is
\[
x_1 : \gamma_1 \to L
\]
\[
\vdots
\]
\[
x_m : \gamma_m \to L
\]
end
where \( \gamma_i \) corresponds to \( t_{i_1} \to \cdots \to t_{i_n} \). In the translation \( \mathcal{F}_i \) judgements \( \Gamma \vdash e : t \) become CPL arrows \( e : \vec{F} \to \vec{t} \). To fit into the \( \mathcal{F}_i \) to CPL translation, the translated operators should have types
\[
\text{left object } L() \text{ with } \psi_L \text{ is}
\]
\[
x_1 : \gamma_1 \to L
\]
\[
\vdots
\]
\[
x_m : \gamma_m \to L
\]
end
where \( \vec{F} \) is some product describing the environment of free variables, and \( x_i \) and \( \text{pr } L \) are used in the definitions of \( c_{L,i} \) and \( \text{iter } L[\tau] \), respectively.

5.2.3. Constructions and declarations. The constructor \( c_{L,i} \) of an inductive type takes \( n_i \) arguments. A constructor of a left object should have a type of the form \( t \to L \) which can be seen as an uncurried form of the type of \( c_{L,i} \). We introduce the CPL declaration
\[
\text{left object } L() \text{ with } \psi_L \text{ is}
\]
\[
x_1 : \cdots (t_{1_1} \star t_{1_2} \cdots) \star t_{1_n} \to L
\]
\[
\vdots
\]
\[
x_m : \cdots (t_{m_1} \star t_{m_2} \cdots) \star t_{m_m} \to L
\]
end
Note that for \( n_i = 0 \) the product collapses to 1 giving \( x_i : 1 \to L \). In accordance with this declaration, we translate the constructors as
\[
c_{L,i} \equiv \begin{cases} x_i \circ ! & \text{if } n_i = 0, c_{L,i} \text{ is a constant}, \\ \text{cur}(\text{cur}^{n_i-1}(x_i) \circ \text{p}2) & \text{if } n_i > 0. \end{cases}
\]
We could have incorporated \( \vec{F} \) in the CPL constructor type as
\[
c_{L,i} \equiv \text{cur}^{n_i}(x_i)
\]
\[
x_i : \cdots (\vec{F} \star \overline{t_{i_1}} \cdots) \star \overline{t_{i_m}} \to L
\]
However, this implies that a copy of $\tilde{F}$—the whole environment—would be made each time $\varepsilon_\gamma$ is used. For a natural number $n$ its representation would then contain $n+1$ copies of $\tilde{F}$. Since constructors are primitive expressions, they do not need the environment, and thus we have inserted the $\text{p2}$ to throw away $\tilde{F}$ before the actual constructor is applied.

5.2.4. The iterators. Intuitively, we want to use the factorizer of a left object for modelling the iterator of an $F_1$ inductive type. While an iterator $\text{iter } L[\tau]: S_1 \to \cdots \to S_m \to L \to \tau$ in $F_1$ has an internal and curried type (i.e., the type is part of the syntax, and we can write $\text{iter } L[\tau]$ in $F_1$-expressions), a factorizer $\psi: \text{Hom}_c(X^*F_x, X^*G_x) \to \text{Hom}_c(\tilde{X}^*I, X)$ in $CPL$ has an external and uncurried type (i.e., the type is not part of $CPL$'s syntax, and we must write $\psi(e_1, \ldots, e_m)$). The trick will be to let $X$ be the higher order type $\Gamma \Rightarrow \tau$. Then $\psi$ first iterates normally over $\tilde{X}^*L$, and next the result is applied to the environment containing values for free variables.

We determine $\phi$ as follows:

\[
\text{iter } L[\tau] \equiv \text{iter } m^{m+1}(\psi)
\]

\[
\phi: \Gamma^* L \to \tilde{\tau}
\]

\[
\Gamma \equiv \cdots (\Gamma^* S_1^*) \cdots \cdot S_m^* 
\]

$\Gamma$ corresponds to a translation (the bars are omitted) of the global environment $\Gamma'$ extended with values for the $m$ “cases” arguments of $\text{iter } L[\tau]$. In $CPL$ the iterator has the form

\[
\psi_L(f_1, \ldots, f_m): L \to Y,
\]

\[
f_i: \cdots (t_{i_1}^\Gamma \cdot t_{i_2}^\Gamma) \cdots \cdot t_{i_m}^\Gamma \to Y,
\]

where $f_1, \ldots, f_m$ correspond to the $m$ “case” arguments of $\text{iter } L[\tau]$. The $CPL$ computation rule for $\psi$ says that when applied to an element of type $L$ the result reduces to some expression involving $f_1, \ldots, f_m$. The actual “case” arguments are stored in the environment. To see how to distribute them properly, we first apply the trick:

\[
Y \equiv \Gamma \Rightarrow \tilde{\tau},
\]

\[
\phi \equiv \text{app } \circ \langle \psi_L(f_1, \ldots, f_m) \circ \text{p2, p1} \rangle: \Gamma^* L \to \tilde{\tau}.
\]

We see that we first iterate over the element of type $L$, obtaining a term of type $\Gamma \Rightarrow \tilde{\tau}$ which is then applied to $\Gamma$. We define

\[
f_i \equiv \text{cur}(f_i^\Gamma),
\]

\[
f_i^\Gamma: \cdots (t_{i_1}^\Gamma \cdot t_{i_2}^\Gamma) \cdots \cdot t_{i_m}^\Gamma \to Y.
\]
At some position in the environment we have

\[ S_i \equiv t_{i1} \rightarrow \cdots \rightarrow t_{in} \rightarrow L \]

which corresponds to the destructor to use in case \( i \). We define

\[
i' = \text{app} \circ \langle \cdots \langle \text{app} \circ \langle \text{position}(S_i) \circ \text{p2}, \delta^1_i \rangle, \delta^2_i \cdots, \delta^n_i \rangle, \delta_i^j \rangle ; (\cdots (t_{i1} \mapsto r \cdot t_{i2} \mapsto r \cdot \cdots) \cdot t_{in} \mapsto r) \cdot r \rightarrow t_{ij},
\]

The purpose of \( \delta_i^j \) is to pick out \( t_{ij} \mapsto r \) and to propagate \( r \) into it in order to obtain the \( t_{ij} \) needed by the actual destructor case. In this way the environment is made available to all subterms possibly containing free variables. We define a function \( A \) computing \( \delta_i^j \) according to the structure of \( t_{ij} \). The definition of \( A \) in Fig. 13 is quite similar to the \( F \) computation rule for iterators. The final translation of inductive types is given in Fig. 14. Below we go into detail with the natural numbers example. For another example, the reader may compare the \( F \) inductive type Tree from Section 1.3 with the CPL left object tree from Section 3.2.

### 5.2.5. Natural numbers.

Our translation of the \( \mathcal{F}_1 \) natural numbers (cf. 4.10) gives us the CPL declaration

\[
\text{left object } N(\_ \_ \_ \_) \text{ with prN is}
z : 1 \rightarrow N
s : N \rightarrow N
\]

end

---

Input:  
\( t \): \( \mathcal{F}_1 \) type defined relatively to the names \( A \in \mathcal{A} \) and \( L \)
(initially \( t_{ij} \))
\( \pi \): CPL term with codomain \( t_{ij} \mapsto r \mapsto t \)
(initially the projection of \( t_{ij} \mapsto r \mapsto i \))
\( \gamma \): CPL term (the position of \( r \))

Output:  
CPL term with codomain \( t' \)

\[
\Delta A \pi \gamma \pi = \pi \\
\Delta L \pi \gamma = \text{app} \circ \langle \pi, \gamma \rangle \\
\Delta t_1 \rightarrow t_2 \pi \gamma = \text{cur}(\Delta t_2 (\text{app} \circ \langle \pi \circ \text{p2}, \Delta t_1 \text{p2}(\gamma \circ \text{p2}) \rangle)(g \circ \text{p2}))
\]

Fig. 13. The \( \Delta \) function.
Fig. 14. The translation of inductive types.

with the derived CPL terms:

\[
\begin{align*}
\emptyset &:= \text{zo!}, \\
succ &:= \text{cur}(s \circ p2), \\
\text{iterN}[\tau] &:= \text{cur}(\text{cur}(\text{app} (\text{prN}(f_1, f_2) \circ p2, p1))), \\
f_1 &:= \text{cur}(p2 \circ p1 \circ p2), \\
f_2 &:= \text{cur}(\text{app} (p2 \circ p2, \text{app} (p1, p2))) \\
&\cong \text{cur}(\text{app} (p2 \circ p2, \text{app})).
\end{align*}
\]

It is routine to check that the reduction rules are preserved by this translation:

\[
\begin{align*}
\text{iterN}[\tau]/g\ n &\cong \text{app} (\text{app} (\text{app} \left(\text{iter} L[\tau], f\right), g), n) \\
\cong \text{app} (\text{prN}(f_1, f_2) \circ p2, p1) \circ (\langle\langle 1, f\rangle, g\rangle, n) \\
&\cong \text{app} (\text{prN}(f_1, f_2) \circ n, \langle\langle 1, f\rangle, g\rangle).
\end{align*}
\]
Take the example \(n = (\text{succ } 0)\). Then \(n \equiv \text{app } o (\text{cur } (s \circ p2), z \circ o)\) which reduces to \(s \circ z \circ o\), and the last line of the above reduction sequence reduces further:

\[
\begin{align*}
\text{app } o (f_2 \circ f_1 \circ o, \langle \langle 1, f_1 \rangle, \tilde{g} \rangle) \\
\text{app } o (\text{cur } (\text{app } o (p2 \circ p2, \text{app}) \\
\circ \langle \text{cur } (p2 \circ p1 \circ p2 \circ \langle l \circ p1, p2 \rangle) \circ p1, p2 \rangle), \\
\langle \langle 1, \tilde{f} \rangle, \tilde{g} \rangle) \\
\text{app } o (p2 \circ p2, \text{app}) \\
\circ \langle \text{cur } (p2 \circ p1 \circ p2 \circ \langle l \circ p1, p2 \rangle) \circ p1, p2 \rangle, \\
\langle \langle 1, \tilde{f} \rangle, \tilde{g} \rangle \\
\text{app } o (\tilde{g}, p2 \circ p1 \circ p2 \circ \langle l \circ p1, p2 \rangle) \\
\circ \langle 1, \langle 1, \tilde{f} \rangle, \tilde{g} \rangle \\
\text{app } o (\tilde{g}, \tilde{f}) \\
\equiv (f, g).
\end{align*}
\]

As final examples we give \textit{add} and \textit{ack} in \textit{CPL} (cf. 4.10):

\[
\begin{align*}
\text{add} & \equiv \text{cur } (\text{app } o (\text{app } o (\text{app } o (\text{iterN}[N], p2 \circ p1), \\
\text{cur } (s \circ p2), p2)), \\
\text{it} & \equiv \text{cur } (\text{app } o (\text{app } o (\text{app } o (\text{iterN}[N], s \circ o), \\
p2 \circ p1, s \circ p2)), \\
\text{ack} & \equiv \text{cur } (\text{app } o (\text{app } o (\text{app } o (\text{iterN}[N \rightarrow N], \text{cur } (s \circ p2), \\
it), s \circ p2))).
\end{align*}
\]

5.2.6. Discussion. We have given a translation of \(F_1\) inductive types to \textit{CPL}. This translation extends the translation of \(F_1\) to \textit{CPL}. As a special case the natural numbers may be translated. The natural numbers is a first order type. The result of Section 5.1 was that \textit{CPL} can represent all \(F_1\) functions with first order result-type. Thus \textit{CPL} can represent all \(F_1\) functions \(\tau \rightarrow \mathcal{N}\), for any type \(\tau\).

In particular, Ackermann's function, which is \(F_1\) computable and has type \(\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}\), can be computed in \textit{CPL} for any pair of natural numbers. Thus, \textit{CPL} is more than primitive recursive.
Further, in [9] Girard shows that $\mathcal{F}_1$ + natural numbers with recursion can represent all $\mathcal{N} \rightarrow \mathcal{N}$ functions provably total in first order Peano arithmetic. We have only natural numbers with iteration, but with an iterator we can encode the same $\mathcal{N} \rightarrow \mathcal{N}$ functions as with recursion [9, p. 51]. Thus, CPL can represent all $\mathcal{N} \rightarrow \mathcal{N}$ functions provably total in first order Peano arithmetic.

6. Conclusion

We summarize the results, discuss related work, and mention future directions.

6.1. Results. We have shown two properties of CPL relative to $\mathcal{F}_1$:
- CPL can represent $\mathcal{F}_1$ terms of first order types with $\beta$-redexes (but not $\eta$);
- CPL can represent $\mathcal{F}_1$ inductive types.
This has some consequences, of which we have shown
- CPL can represent Ackermann's function, so CPL is more than primitive recursive;
- CPL can represent (at least) all functions $\text{Nat} \rightarrow \text{Nat}$ provably total in first-order Peano arithmetic.
Finally we conjecture that
- CPL cannot represent the full $\beta$- or $\eta$-reduction of $\mathcal{F}_1$;
- The reduction rules of CPL can be changed so that (the new) CPL can represent the full $\beta$-reduction of $\mathcal{F}_1$.

6.2. Related work.
- Curien [5]. Our treatment of $\eta$ and $\beta$ reduction preservation is similar to parts of this book. However, we focus on the reductions of a programming language rather than the equations of an underlying theory, and our languages are typed.
- Lambek and Scott [14]. In this book they show an isomorphism between simply typed $\lambda$-calculi with surjective pairing and iterator (i.e., natural numbers), and CCC's with a weak natural numbers object. Our encoding is more general than theirs, and by specialization to natural numbers we get an explicit definition of the translated iterator.
- Wraith [18]. He translates most CPL into $\mathcal{F}_2$, the second order typed $\lambda$-calculus [9, system F] and [16]. An interesting result is that $\mathcal{F}_2$—which is strongly normalizing—can represent infinite objects like streams. In contrast to our comparison with $\mathcal{F}_1$, Wraith is most concerned with types and does not discuss preservation of reduction rules or normal form distinctions. In [6] a more direct translation is given which also encompass all of CPL.
- Böhm and Berarducci [3]. They treat iteratively defined functions between heterogeneous term algebras. Such algebras are similar to left objects, and the functions are similar to CPL programs. They are concerned with equation solving and the representation in $\mathcal{F}_2$ (which they call $A$).
6.3. Future directions. As further work we suggest

- strengthening of CPL’s reduction rules (primarily to obtain full representations of β and perhaps η reductions);
- programming in CPL, especially to take advantage of the infinity potentials using right objects;
- investigating fixed points in CPL, i.e., some equivalent of admitting loops (for example, \( X \simeq B \Rightarrow X \)) in the type system.

Acknowledgment

We thank Neil Jones for many deep comments on readability, contents, and goals, and Andrzej Filinski for a discussion of the natural numbers.

References