CPS Transform and Type Derivations in Dependent ML

(Extended Abstract)

Hongwei Xi
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH-0300
Tel: +1 513 556 4762 Fax: +1 513 556 7326
hwxi@ececs.uc.edu

Carsten Schürmann
Department of Computer Science
Yale University
New Haven, CT 06520-8285
Tel: +1 203 432 2349 Fax: +1 203 432 0593

CPS Transform and Type Derivations in Dependent ML

(Extended Abstract)

Version of April 2, 2001

Abstract

Dependent ML is a functional programming language that extends ML with a restricted form of dependent types. In this paper, we study a call-by-value continuation-passing style (CPS) transform for $\mathrm{ML}_0^{\Pi,\Sigma}$, a core of DML that excludes effects. In particular, we demonstrate how the type derivation of an expression in $\mathrm{ML}_0^{\Pi,\Sigma}$ can be transformed into the type derivation of the CPS transform of the expression, lifting CPS transform from the level of expressions to the level of type derivations. This work serves as the first step in our attempt to build a type-preserving compiler for DML by compiling the type derivation of a program instead of the program itself.

1 Introduction

The question of establishing the correctness of a compiler has been raised since the beginning moments of compiler studies. Let C be a compiler that translates a source program e into a target program $C(e)$. Then the task to establish the correctness of C amounts to proving that the semantics of e is the same as that of $C(e)$. In other words, a compiler is correct if it is semantics-preserving. However, it is often of great difficulty to prove that a compiler is semantics-preserving given that (a) the semantics of a realistic programming language is often difficult to formalize, (b) the execution models of source and target languages are often far different, and (c) a (minor) change to the complier (e.g., implementation of a new optimization strategy), which is likely to occur during the development cycle of a compiler, may require a renewed effort to reconstruct the correctness proof of the compiler. Therefore, it currently seems impractical to establish compiler correctness in a realistic setting.

We now take a look from a different angle at the question of establishing the correctness of a compiler. Suppose that the source program e possesses a property P, i.e., $P(e)$ holds. For instance, $P(e)$ could mean that e is terminating. Then we may know that $C(e)$ should also possess the property P if C is a correct compiler. In other words, C must be P-preserving if C is semantics-preserving. If P is such a property for which there is a practical means to check whether both $P(e)$ and $P(C(e))$ hold, then we can raise our confidence in the correctness of C through the following practice. Given a program e such that $P(e)$ is verified; after compiling e into $C(e)$, we verify whether $P(C(e))$ holds; if the verification succeeds, we gain confidence in the correctness of $C(e)$, i.e., $C(e)$ being semantically equivalent to e; otherwise, we know that $C(e)$ is incorrect and some errors in the compiler C need to be fixed.

In a typed programming language, a well-typed program possesses some properties guaranteed by the type system. Assume that the source language of a compiler C is typed. Then it seems natural to expect that a type system for the target language of the compiler can be designed to guarantee similar properties and C can then be constructed to always compile a well-typed source program e into a well-typed target program $C(e)$. We call such a C a type-preserving compiler and we are interested in constructing a type-preserving compiler for Dependent ML (DML).

DML is a functional programming language that enriches ML with a restricted form of dependent types (Xi and Pfenning 1999; Xi 1998). This enrichment allows for specification and inference of significantly more precise type information, facilitating program error detection at compile-time. The reader can find various programming examples in DML on-line (Xi 1999). For instance, the following is an implementation of the factorial function in DML.

```
fun fact (x) = if x = 0 then 1 else fact (x-1)
withtype {a:nat} int(a) -> [b:int] int(b)
```

The novelty here is the `withtype` clause, which assigns the dependent type $\Pi a : nat.int(a) \to \Sigma b : int.int(b)$ to the function $fact$. We have refined the usual integer type int into infinitely many singleton types $int(a)$ such that only expressions with integer value a can be of type $int(a)$, where a ranges over integers. The type of $fact$ indicates that this is a function that takes a natural number(or literally, a value of type $int(a)$ for some natural number a) and returns an integer. After elaboration, $fact$ is transformed into the following explicitly typed expression e_{fact} in Church typing style.

$$\textbf{fix}\ fact : \Pi a : nat.int(a) \to int.\lambda_i\, a : nat.\lambda_e\, x : int(a).\textbf{if}(= [a][0](x, 0), 1, fact[a-1](-[a][1](x, 1)))$$

Note that we have assigned $=$ and $-$ the following types, respectively.

$$\Pi a : int.\Pi b : int.int(a) * int(b) \to int(eq(a, b)) \text{ and } \Pi a : int.\Pi b : int.int(a) * int(b) \to int(a - b)$$

We use $eq(a, b)$ for the function that returns 1 and 0 depending on whether a equals b. In order to type-check e_{fact}, we need to derive $a : nat, eq(a, 0) = 0 \vdash a - 1 : nat$, that is, to show that $a - 1$ is a natural number under the assumption that a is a natural number and a is not 0. This involves proving the following,

$$\forall a : int.a \geq 0 \supset (a \neq 0 \supset a - 1 \geq 0)$$

which we call a constraint. In general, type-checking in DML often involves a great deal of constraint constraint solving.

Suppose that we are to construct a type-preserving compiler C for DML. Given $e : \tau$, that is, program e is of type τ, in DML, we need to translate e and τ into $C(e)$ and $C(\tau)$, respectively, such that $C(e) : C(\tau)$ can be derived in the (dependent) type system of the target language of C. In our experiment, we observe that type-checking $C(e) : C(\tau)$ then involves solving constraints that are essentially the same as those solved during type-checking $e : \tau$. As constraint solving can be time-consuming, a natural question is whether we can avoid solving the same constraints repeatedly by preserving the proofs of these constraints. This question prompts us to study the possibility of compiling the type derivation of a program instead of the program itself.

Given a type derivation $\mathcal{D}$ of $e : \tau$ in DML, we intend to represent $\mathcal{D}$ as $\ulcorner \mathcal{D} \urcorner$ in the logical framework LF (Harper, Honsell, and Plotkin 1993) and then build a compiler that translates $\ulcorner \mathcal{D} \urcorner$ into $C(encode\mathcal{D})$, which represents a type derivation of $C(e) : C(\tau)$ in the target language of C. This practice offers two immediate advantages.

- It is no longer necessary to implement a type-checker for the target language as the type-checker for LF suffices. This is particularly attractive when a compiler consists of a number of phases and we want to verify that each phase is type-preserving.
- The proofs of constraints encountered during type-checking $e : \tau$ are encoded into $\ulcorner \mathcal{D} \urcorner$ and are then carried into $C(\ulcorner \mathcal{D} \urcorner)$, obviating the need to solve the same constraints repeatedly.

Many compilers such as TIL (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996) (and its successor TILT) and FLINT (Shao 1997) for function programming languages start with a continuation-passing style (CPS) transform. In this paper, we demonstrate how CPS transform for $\mathrm{ML}_0^{\Pi,\Sigma}$, a core of DML that excludes effects, can be lifted from the level of expressions to the level of type derivations, presenting a concrete example of compiling type derivations. This serves as the first step in our attempt to build a type-preserving compiler for DML.

The main contribution of the paper lies in our forming the notion of compiling type derivations, which we believe suggests a promising novel approach to building type-preserving compilers. The technical contribution of the paper consists the formation of the CPS transform for $\mathrm{ML}_0^{\Pi,\Sigma}$ and a formalization of the transform in Twelf (Schürmann and Pfenning 1998).

The rest of paper is organized as follows. We form a language $\mathrm{ML}_0^{\Pi,\Sigma}$ in Section 2, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types, developed in DML. We then present the CPS transform for $\mathrm{ML}_0^{\Pi,\Sigma}$ in Section 3 and lift to the level of type derivations. In Section 4, we briefly explain how the CPS transform can be formalized in Twelf. We then mention some related work and conclude. The reader can find omitted details on-line (Xi and Schürmann 2001).

2 $\mathrm{ML}_0^{\Pi,\Sigma}$

We start with a language $\mathrm{ML}_0^{\Pi,\Sigma}$, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types and (general) recursion. The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$ is given in Figure 1.

2.1 Syntax

We fix an integer domain and restrict type index expressions, namely, the expressions that can be used to index a type, to this domain. This is a sorted domain and subset sorts can be formed. For instance, we use

index constants	c_I	$::=$	$\cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$
index expressions	i	$::=$	$a \mid c_I \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1 / i_2 \mid$ $lt(i_1,i_2) \mid gt(i_1,i_2) \mid lte(i_1,i_2) \mid gte(i_1,i_2) \mid eq(i_1,i_2) \mid neq(i_1,i_2)$
index propositions	P	$::=$	$i_1 < i_2 \mid i_1 \leq i_2 \mid i_1 > i_2 \mid i_1 \geq i_2 \mid i_1 = i_2 \mid i_1 \neq i_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2$
index sorts	γ	$::=$	$int \mid \{a : \gamma \mid P\}$
index variable contexts	ϕ	$::=$	$\cdot \mid \phi, a : \gamma \mid \phi, P$
index constraints	Φ	$::=$	$P \mid P \supset \Phi \mid \forall a : \gamma . \Phi$
types	τ	$::=$	$\delta(i) \mid \tau_1 \to \tau_2 \mid \Pi a : \gamma . \tau \mid \Sigma a : \gamma . \tau$
contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau \mid \Gamma, f : \tau$
constants	c	$::=$	$true \mid false \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \cdots$
expressions	e	$::=$	$c \mid x \mid f \mid \mathbf{if}(e, e_1, e_2) \mid \lambda_i\, a : \gamma . v \mid \lambda_e\, x : \tau . e \mid e_1(e_2) \mid$ $\mathbf{fix}\ f : \tau . v \mid e[i] \mid \langle i \mid e \rangle \mid \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2$
values	v	$::=$	$c \mid x \mid \lambda_i\, a : \gamma . v \mid \lambda_e\, x : \tau . e \mid \langle i \mid v \rangle$

Figure 1: The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$

nat for the subset sort $\{a : int \mid a \geq 0\}$. We use $\delta(i)$ for a base type indexed with an index expression i, which may be empty. For instance, $\texttt{bool}(0)$ and $\texttt{bool}(1)$ are types for boolean values *false* and *true*, respectively; for each integer i, $\texttt{int}(i)$ is the singleton type for integer expressions with value equal to i.

We use $\Pi a : \gamma.\tau$ and $\Sigma a : \gamma.\tau$ for the usual dependent function and sum types, respectively. We also introduce λ-variables and ρ-variables in $\mathrm{ML}_0^{\Pi,\Sigma}$ and use x and f for them, respectively. A lambda-abstraction can only be formed over a λ-variable while recursion (via fixed point operator) must be formed over a ρ-variable. A λ-variable is a value but a ρ-variable is not.

We use λ_i for abstracting over index variables, λ_e for abstracting over variables, and **fix** for forming recursive functions. Note that the body after either λ_e or **fix** must be a value. We use $\langle i \mid e \rangle$ for packing an index i with an expression e to form an expression of a dependent sum type, and **open** for unpacking an expression of a dependent sum type.

2.2 Constraint Domain

Unlike in general dependent type theory, type index expressions in DML can only be drawn from a given constraint domain C, which is fixed to be the integer domain in this paper.

We use $\phi \models P$ for a satisfaction relation, which means P holds under ϕ, that is, the formula $(\phi)P$, defined below, is satisfied in the domain of integers.

$$(\cdot)\Phi = \Phi \qquad (\phi, a : int)\Phi = (\phi)\forall a : int.\Phi$$
$$(\phi, a : \{a : \gamma \mid P\})\Phi = (\phi, a : \gamma)(P \supset \Phi)$$
$$(\phi, P)\Phi = (\phi)(P \supset \Phi)$$

For instance, the satisfaction relation

$$a : nat, a \neq 0 \models a - 1 \geq 0$$

holds since the following formula is true in the integer domain.

$$\forall a : int.a \geq 0 \supset (a \neq 0 \supset a - 1 \geq 0)$$

Note that the decidability of the satisfaction relation depends on the constraint domain. For the integer constraint domain we use here, the satisfaction relation is decidable (as we do not accept nonlinear integer constraints). The sorting rules for the constraint domain are given below. We omit the rules for forming legal sorts and legal index variable contexts.

$$\frac{\phi(a) = \gamma}{\phi \vdash a : \gamma} \qquad \frac{\phi \vdash i : \{a : \gamma \mid P\}}{\phi \vdash i : \gamma} \qquad \frac{\phi \vdash i : \gamma \quad \phi \models P[a \mapsto i]}{\phi \vdash i : \{a : \gamma \mid P\}}$$

2.3 Static Semantics

We omit the rules for forming legal types and contexts, which are standard.

index substitutions	θ_I	$::=$	$[] \mid \theta_I[a \mapsto i]$
substitutions	θ	$::=$	$[] \mid \theta[x \mapsto e] \mid \theta[f \mapsto e]$

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \vdash \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2} \ \textbf{(tp-eq)}$$

$$\frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau} \ \textbf{(tp-}\lambda\textbf{-var)} \qquad \frac{\Gamma(f) = \tau}{\phi;\Gamma \vdash f : \tau} \ \textbf{(tp-}\rho\textbf{-var)}$$

$$\frac{\phi, a : \gamma;\Gamma \vdash v : \tau}{\phi;\Gamma \vdash \lambda_i\, a : \gamma.v : \Pi a : \gamma.\tau} \ \textbf{(tp-ilam)}$$

$$\frac{\phi;\Gamma \vdash e : \Pi a : \gamma.\tau \quad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau[a \mapsto i]} \ \textbf{(tp-iapp)}$$

$$\frac{\phi;\Gamma, f : \tau \vdash v : \tau}{\phi;\Gamma \vdash \textbf{fix}\ f : \tau.v : \tau} \ \textbf{(tp-fix)}$$

$$\frac{\phi;\Gamma \vdash e : \texttt{bool}(i) \quad \phi, i = 1;\Gamma \vdash e_1 : \tau \quad \phi, i = 0;\Gamma \vdash e_2 : \tau}{\phi;\Gamma \vdash \textbf{if}(e, e_1, e_2) : \tau} \ \textbf{(tp-if)}$$

$$\frac{\phi;\Gamma, x : \tau_1 \vdash e : \tau_2}{\phi;\Gamma \vdash \lambda_e\, x : \tau_1.e : \tau_1 \to \tau_2} \ \textbf{(tp-lam)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2} \ \textbf{(tp-app)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \phi, a : \gamma;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \textbf{open}\ e_1\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_2 : \tau_2} \ \textbf{(tp-open)}$$

$$\frac{\phi \vdash i : \gamma \quad \phi;\Gamma \vdash e : \tau[a \mapsto i]}{\phi;\Gamma \vdash \langle i \mid e\rangle : \Sigma a : \gamma.\tau} \ \textbf{(tp-pack)}$$

Figure 2: Typing Rules for $\mathrm{ML}_0^{\Pi,\Sigma}$

A substitution is a finite mapping and [] represents an empty mapping. We use θ_I for a substitution mapping index variables to index expressions and $\mathbf{dom}(\theta_I)$ for the domain of θ_I. Similar notations are used for substitutions on variables. We write $\bullet[\theta_I]$ ($\bullet[\theta]$) for the result from applying θ_I (θ) to $\bullet$, where $\bullet$ can be a type, an expression, etc. The standard definition is omitted. The following rules are for judgments of form $\phi \vdash \theta_I : \phi'$, which roughly means that θ_I has "type" ϕ'.

$$\frac{}{\phi \vdash [] : \cdot} \ \textbf{(sub-i-empty)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \vdash i : \gamma[\theta_I]}{\phi \vdash \theta_I[a \mapsto i] : \phi', a : \gamma} \ \textbf{(sub-i-var)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \models P[\theta_I]}{\phi \vdash \theta_I : \phi', P} \ \textbf{(sub-i-prop)}$$

We write $\mathbf{dom}(\Gamma)$ for the domain of Γ, that is, the set of variables declared in Γ. Given substitutions θ_I and θ, we say $\phi;\Gamma \vdash (\theta_I;\theta) : (\phi';\Gamma')$ holds if $\phi \vdash \theta_I : \phi'$ and $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ and $\phi;\Gamma \vdash \theta(x) : \Gamma'(x)[\theta_I]$ for all $x \in \mathbf{dom}(\Gamma')$.

We write $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i = j$ from index expressions to types, determined by the following rules. It is the application of these rules that generates constraints during type-checking.

$$\frac{\phi \models i = j}{\phi \models \delta(i) \equiv \delta(j)} \qquad \frac{\phi \models \tau_1' \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

$$\frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Pi a : \gamma.\tau \equiv \Pi a : \gamma.\tau'} \qquad \frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Sigma a : \gamma.\tau \equiv \Sigma a : \gamma.\tau'}$$

We present the typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$ in Figure 2. Some of these rules have obvious side conditions, which are omitted. For instance, in the rule **(tp-ilam)**, a cannot have free occurrences in Γ. The following lemma plays a pivotal rôle in proving the subject reduction theorem for $\mathrm{ML}_0^{\Pi,\Sigma}$, whose standard proof is available in (Xi 1998).

$$
\begin{array}{rcl}
\|\delta(i)\|_* & = & \delta(i) \\
\|\tau_1 \rightarrow \tau_2\|_* & = & \|\tau_1\|_* \rightarrow \|\tau_2\| \\
\|\Pi a : \gamma.\tau\|_* & = & \Pi a : \gamma.\|\tau\|_* \\
\|\Sigma a : \gamma.\tau\|_* & = & \Sigma a : \gamma.\|\tau\|_* \\
\|\tau\| & = & (\|\tau\|_* \rightarrow ans) \rightarrow ans \\
\|c\|_* & = & c \\
\|x\|_* & = & x \\
\|f\|_* & = & f \\
\|\lambda_e\, x.e\|_* & = & \lambda_e\, x.\|e\| \\
\|v\| & = & \lambda_e\, k.k(\|v\|_*) \\
\|\mathbf{if}(e, e_1, e_2)\| & = & \lambda_e\, k.\|e\|(\lambda_e\, x.\mathbf{if}(x, \|e_1\|(k), \|e_2\|(k))) \\
\|e_1(e_2)\| & = & \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\|e_2\|(\lambda_e\, x_2.x_1(x_2)(k))) \\
\|e[i]\| & = & \lambda_e\, k.\|e\|(\lambda_e\, x.x[i](k)) \\
\|\lambda_i\, a.e\| & = & \lambda_i\, a.\|e\| \\
\|\mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ e_2\| & = & \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ \|e_2\|(k)) \\
\|\langle i \mid e\rangle\| & = & \langle i \mid \|e\|\rangle \\
\|\mathbf{fix}\ f.v\| & = & \lambda_e\, k.(\mathbf{fix}\ f.\|v\|)(k)
\end{array}
$$

Figure 3: CPS transform for types and expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$

Lemma 2.1 *Assume* $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ *is derivable and* $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ *holds. Then we can derive* $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I]$.

2.4 Dynamic Semantics

We present the dynamic semantics of $\mathrm{ML}_0^{\Pi,\Sigma}$ through the use of evaluation contexts defined below. Certainly, there are other possibilities for this purpose, which we do not explore here.

$$\text{evaluation contexts}\quad E \quad ::= [\,] \mid \mathbf{if}(E, e_1, e_2) \mid E[i] \mid E(e) \mid v(E) \mid \langle i \mid E\rangle \mid \mathbf{open}\ E\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ e$$

We write $E[e]$ for the expression resulting from replacing the hole $[\,]$ in E with e. Note that this replacement can *never* result in capturing free variables.

Definition 2.2 *A redex is defined below.*

- $\mathbf{if}(c, e_1, e_2)$ *are redexes for* $c = true, false$, *which reduce to* e_1 *and* e_2, *respectively.*
- $(\lambda_e\, x : \tau.e)(v)$ *is a redex, which reduces to* $e[x \mapsto v]$.
- *Let* e *be* $\mathbf{fix}\ f : \tau.v$. *Then* e *is a redex, which reduces to* $\lambda_i\, a : \gamma.v[f \mapsto e]$.
- $(\lambda_i\, a : \gamma.v)[i]$ *is a redex, which reduces to* $v[a \mapsto i]$.
- $\mathbf{open}\ \langle i \mid v\rangle\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ e$ *is a redex, which reduces to* $e[a \mapsto i][x \mapsto v]$.

We use r *for a redex and write* $r \hookrightarrow e$ *if* r *reduces to* e. *If* $e_1 = E[r]$, $e_2 = E[e]$ *and* $r \hookrightarrow e$, *we write* $e_1 \hookrightarrow e_2$ *and say* e_1 *reduces to* e_2 *in one step.*

Let $\hookrightarrow^*$ be the reflexive and transitive closure of $\hookrightarrow$. We say e_1 reduces to e_2 (in many steps) if $e_1 \hookrightarrow^* e_2$. We omit the standard proof for the following subject reduction theorem, which uses Lemma 2.1.

Theorem 2.3 *(Subject Reduction) Assume* $\cdot; \cdot \vdash e : \tau$ *is derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$. *If* $e \hookrightarrow^* e'$, *then* $\cdot; \cdot \vdash e' : \tau$ *is also derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$.

3 CPS Transform for $\mathrm{ML}_0^{\Pi,\Sigma}$

The expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$ are explicitly typed. In the following presentation, we are to use the type derivation of an expression to witness the well-typedness of the expression. Therefore, we are to omit types in expressions.

In Figure 3, we present the CPS transform for types and expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$, where *ans* is assumed to be a newly introduced type. This is a standard call-by-value CPS transform. The treatment of type index expressions in the transform clearly indicates that they have no effect on the operational semantics of expressions.

Theorem 3.1 *Given a type derivation $\mathcal{D}$ for $\phi; \Gamma \vdash e_0 : \tau_0$, we are to construct a type derivation $\|\mathcal{D}\|$ for $\phi; \|\Gamma\| \vdash \|e_0\| : \|\tau_0\|$, where $\|\Gamma\|(x) = \|\Gamma(x)\|_*$ and for $x \in \mathbf{dom}(\Gamma)$ and $\|\Gamma\|(f) = \|\Gamma(f)\|$ for $f \in \mathbf{dom}(\Gamma)$. In other words, we are to lift the transform $\|\cdot\|$ from expressions to their type derivations.*

Proof We present some interesting case. **Case:** The last applied rule is **(tp-eq)**.

$$\frac{\phi; \|\Gamma\| \vdash \|e_0\| : \|\tau_1\| \quad \phi \vdash \|\tau_1\| \equiv \|\tau_0\|}{\phi; \|\Gamma\| \vdash \|e_0\| : \|\tau_0\|}$$

It should be clear that a derivation of $\phi \vdash \|\tau_1\| \equiv \|\tau_0\|$ can be readily transformed from a derivation of $\phi \vdash \tau_1 \equiv \tau_0$.

Case: The last applied rule is **(tp-app)**. Let $e_0 = e_1(e_2)$ and $\tau_0 = \tau_1$. Note that

$$\|e_1(e_2)\| = \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\|e_2\|(\lambda_e\, x_2.x_1(x_2)(k)))$$

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ are type derivations for $\phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\phi; \Gamma \vdash e_2 : \tau_2$, respectively. Let $\Gamma_0 = \|\Gamma\|, k : \|\tau_2\|_* \to ans$, $\Gamma_1 = \Gamma_0, x_1 : \|\tau_1 \to \tau_2\|_*$, and $\Gamma_2 = \Gamma_1, x_2 : \|\tau_1\|_*$. Then $\|\mathcal{D}\|$ can be formed as follows.

$$\dfrac{\begin{array}{c}\|\mathcal{D}_1\|\\ \phi; \Gamma_0 \vdash \|e_1\| : \|\tau_1 \to \tau_2\|\end{array} \quad \dfrac{\dfrac{\begin{array}{c}\|\mathcal{D}_2\|\\ \phi; \Gamma_1 \vdash \|e_2\| : \|\tau_2\|\end{array} \quad \dfrac{\dfrac{\dfrac{\phi; \Gamma_2 \vdash x_1 : \|\tau_1 \to \tau_2\|_* \quad \phi; \Gamma_2 \vdash x_2 : \|\tau_1\|_*}{\phi; \Gamma_2 \vdash x_1(x_2) : \|\tau_2\|} \quad \phi; \Gamma_2 \vdash k : \|\tau_1\|_* \to ans}{\phi; \Gamma_2 \vdash x_1(x_2)(k) : ans}}{\phi; \Gamma_1 \vdash \lambda_e\, x_2.x_1(x_2)(k) : \|\tau_2\|_* \to ans}}{\phi; \Gamma_1 \vdash \|e_2\|(\lambda_e\, x_2.x_1(x_2)(k)) : ans}}{\phi; \Gamma_0 \vdash \lambda_e\, x_1.\|e_2\|(\lambda_e\, x_2.x_1(x_2)(k)) : \|\tau_1 \to \tau_2\|_* \to ans}}{\dfrac{\phi; \Gamma_0 \vdash \|e_1\|(\lambda_e\, x_1.\|e_2\|(\lambda_e\, x_2.x_1(x_2)(k))) : ans}{\phi; \|\Gamma\| \vdash \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\|e_2\|(\lambda_e\, x_2.x_1(x_2)(k))) : \|\tau_2\|}}$$

Case: The last applied rule is **(tp-iapp)**. Let $e_0 = e[i]$ and $\tau_0 = \tau[a \mapsto i]$. Note that

$$\|e[i]\| = \lambda_e\, k.\|e\|(\lambda_e\, x.x[i](k))$$

Let $\mathcal{D}_1$ be the type derivation of $\phi; \Gamma \vdash e : \Pi a : \gamma.\tau$. Let $\Gamma_0 = \|\Gamma\|, k : \|\tau[a \mapsto i]\|_* \to ans$ and $\Gamma_1 = \Gamma_0, x : \|\Pi a : \gamma.\tau\|_*$. Then $\|\mathcal{D}\|$ can be formed as follows.

$$\dfrac{\begin{array}{c}\|\mathcal{D}_1\|\\ \phi; \Gamma_0 \vdash \|e\| : \|\Pi a : \gamma.\tau\|\end{array} \quad \dfrac{\dfrac{\dfrac{\phi; \Gamma_1 \vdash x : \|\Pi a : \gamma.\tau\| \quad \phi \vdash i : \gamma}{\phi; \Gamma_1 \vdash x[i] : \|\tau[a \mapsto i]\|} \quad \phi; \Gamma_1 \vdash k : \|\tau[a \mapsto i]\|_* \to ans}{\phi; \Gamma_1 \vdash x[i](k) : ans}}{\phi; \Gamma_0 \vdash \lambda_e\, x.x[i](k) : \|\Pi a : \gamma.\tau\|_* \to ans}}{\dfrac{\phi; \Gamma_0 \vdash \|e\|(\lambda_e\, x.x[i](k)) : ans}{\phi; \|\Gamma\| \vdash \lambda_e\, k.\|e\|(\lambda_e\, x.x[i](k)) : \|\tau[a \mapsto i]\|}}$$

Case: The last applied rule is **(tp-open)**. Let $e_0 = \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2$ and $\tau_0 = \tau_2$. Note that

$$\|\mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2\| = \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ \|e_2\|(k))$$

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the type derivations of $\phi;\Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1$ and $\phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2$, respectively. Let $\Gamma_0 = \|\Gamma\|, k : \|\tau_2\|_* \to ans$,$\Gamma_1 = \Gamma_0, x_1 : \|\Sigma a : \gamma.\tau_1\|_*$,and $\Gamma_2 = \Gamma_1, x : \|\tau_1\|_*$. Then $\|\mathcal{D}\|$ can be formed as follows.

$$
\dfrac{\begin{array}{c}\|\mathcal{D}_1\| \\ \phi;\Gamma_0 \vdash \|e_1\| : \|\Sigma a : \gamma.\tau_1\|\end{array} \qquad \dfrac{\dfrac{\phi;\Gamma_1 \vdash x_1 : \|\Sigma a : \gamma.\tau_1\|_* \qquad \dfrac{\begin{array}{c}\|\mathcal{D}_2\| \\ \phi, a : \gamma; \Gamma_2 \vdash e_2 : \|\tau_2\|\end{array} \qquad \phi, a : \gamma; \Gamma_2 \vdash k : \|\tau_2\|_* \to ans}{\phi, a : \gamma; \Gamma_2 \vdash e_2(k) : ans}}{\phi;\Gamma_1 \vdash \mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ \|e_2\|(k) : ans}}{\phi;\Gamma_0 \vdash \lambda_e\, x_1.\mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ \|e_2\|(k) : \|\Sigma a : \gamma.\tau_1\|_* \to ans}}{\dfrac{\phi;\Gamma_0 \vdash \|e_1\|(\lambda_e\, x_1.\mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ \|e_2\|(k)) : ans}{\phi;\|\Gamma\| \vdash \lambda_e\, k.\|e_1\|(\lambda_e\, x_1.\mathbf{open}\ x_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ \|e_2\|(k)) : \|\tau_2\|}}
$$

Case: The last applied rule is **(tp-lam)**. Let $e_0 = \lambda_e\, x.e$ and $\tau_0 = \tau_1 \to \tau_2$. Note that

$$\|\lambda_e\, x.e\| = \lambda_e\, k.k(\lambda_e\, x.\|e\|)$$

Let $\mathcal{D}_1$ be the type derivation of $\phi;\Gamma, x : \tau_1 \vdash e : \tau_2$. Then $\|\mathcal{D}\|$ can be formed as follows.

$$
\dfrac{\dfrac{\phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash k : \|\tau_0\|_* \to ans \qquad \dfrac{\begin{array}{c}\|\mathcal{D}_1\| \\ \phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans, x : \|\tau_1\|_* \vdash \|e\| : \|\tau_2\|\end{array}}{\phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash \lambda_e\, x.\|e\| : \|\tau_0\|_*}}{\phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash k(\lambda_e\, x.\|e\|) : ans}}{\phi;\|\Gamma\| \vdash \lambda_e\, k.k(\lambda_e\, x.\|e\|) : \|\tau_0\|}
$$

Case: The last applied rule is **(tp-fix)**. Let $e_0 = \mathbf{fix}\ f.v$. Note that

$$\|\mathbf{fix}\ f.v\| = \lambda_e\, k.(\mathbf{fix}\ f.\|v\|)(k)$$

Let $\mathcal{D}_1$ be the type derivation of $\phi;\Gamma, f : \tau_0 \vdash v : \tau_0$. Then $\|\mathcal{D}\|$ can be formed as follows.

$$
\dfrac{\dfrac{\dfrac{\begin{array}{c}\|\mathcal{D}_1\| \\ \phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans, f : \|\tau_0\| \vdash \|v\| : \|\tau_0\|\end{array}}{\phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash \mathbf{fix}\ f.\|v\| : \|\tau_0\|} \qquad \phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash k : \|\tau_0\|_* \to ans}{\phi;\|\Gamma\|, k : \|\tau_0\|_* \to ans \vdash (\mathbf{fix}\ f.\|v\|)(k) : ans}}{\phi;\|\Gamma\| \vdash \lambda_e\, k.(\mathbf{fix}\ f.\|v\|)(k) : \|\tau_0\|}
$$

Theorem 3.2 *Assume $\phi;\Gamma \vdash e : \tau$ is derivable in $\mathrm{ML}_0^{\Pi,\Sigma}$. Then $\phi;\|\Gamma\| \vdash \|e\| : \|\tau\|$ is also derivable.*

Proof This immediately follows from the lifted transform $\|\cdot\|$ for type derivations. ■

We can simply transform $\mathrm{ML}_0^{\Pi,\Sigma}$ into a language ML_0 by erasing all syntax related to type index expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$. Then ML_0 basically extends simply typed λ-calculus with recursion. Let $|e|_i$ be the erasure of expression e. We have e_1 reducing to e_2 in $\mathrm{ML}_0^{\Pi,\Sigma}$ implies $|e_1|_i$ reducing to $|e_2|_i$ in ML_0, where $|\cdot|_i$ is the erasure function. Please find more details on this issue in (Xi and Pfenning 1999; Xi 1998). Clearly, we have that the erasure of the CPS transform of an expression is the same as the CPS transform of the erasure of the expression. This again indicates that type index expressions have no effect on the operational semantics of expressions.

4 Formalization in LF

In our design, proofs of type preservation properties are treated as first class objects. They are being generated, manipulated, and transformed to other proofs. The proof of Theorem 2.3 (the subject reduction theorem of DML) in Section 2, for example, states how a valid typing derivation of source expressions can be transformed into valid typing derivations for the result of the dynamic semantics. Below however we discuss the formalization of the proof of Theorem 3.1 (CPS conversion), and show how valid typing derivations can be transformed into valid typing derivations of the the outcome of CPS conversion.

Conveniently, the result of executing CPS conversion can be read out of the result of the proof transformation. Consequently, this transformation algorithm bears many advantages over standard CPS conversion, because it does not only yield the same result, but simultaneously produces a valid typing derivation of the type correctness of this result. Unlike standard CPS conversion schemes, where separate type checking is necessary, this algorithm preserves information about well-typedness, and hence this transformation algorithm provides an implementable compilation algorithm for DML. The proof of Theorem 3.1 can in fact be interpreted as a type-preserving compiler! It is this property that makes our research on compilation with explicit proofs attractive, and we consider it the main contribution of this work.

The quintessential question is how to represent proofs in general, and typing derivations in particular. Those proofs are complex objects, and it is a challenging question of how to represent them elegantly and concisely while preserving adequacy. In fact, what is most challenging is the encoding of side conditions such as the newness of hypotheses and the freshness of parameters. For instance, such assumptions are associated with **(tp-ilam)**, **(tp-lam)**, and **(tp-fix)**.

Among different possibilities we have chosen the logical framework LF (Harper, Honsell, and Plotkin 1993) for representation and its implementation — the meta-logical framework Twelf (Schürmann and Pfenning 1998) — for experimentation. Twelf allows concise and elegant higher-order encodings of many inference systems including their side conditions, such as natural deduction, sequent calculi, and most importantly type systems, operational semantics, compilers, etc. It draws its expressive power from dependent types together with higher-order representation techniques both of which directly support common concepts in deductive systems such as variable binding, capture-avoiding substitutions, parametric and hypothetical judgments and substitution properties.

Alternatively, typing derivations and CPS conversion can also be represented in logical frameworks such as Coq (Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner 1993) and Isabelle (Lawrence 1994), however only at the expense of conciseness. In general, the inference systems defined above cannot be easily cast into inductively defined types while taking advantage of higher-order encodings because they often violate the fundamental positivity condition (Paulin-Mohring 1993). As a consequence, all properties of substitutions that are evoked in a proof (for example of Theorem 2.3 and Theorem 3.1) must be made explicitly and concrete formalization of substitutions must be defined explicitly which are otherwise implicitly provided in LF. Thus, this kind of representation of typing derivations is not easily amenable to manipulation by functions, and has therefore not been considered in the past.

The function that realizes the proof that CPS conversion preserves well-typedness must be defined with respect to the higher-order nature of the encoding of typing derivations. However, only few languages support this feature, and hence it is best implemented as a total function (realizer) in the meta-logic $\mathcal{M}_2^+$ (Schürmann 2000) of type:

$$\forall E : \mathrm{exp}. \forall T : \mathrm{tp}. \forall D : \mathrm{of}\ E\ T. \exists C : \mathrm{exp}. \exists T' : \mathrm{tp}. \exists R : E \stackrel{\mathrm{exp}}{\Longrightarrow} C. \exists R^* : T \stackrel{\mathrm{tp}^*}{\Longrightarrow} T'. \exists Q : \mathtt{of}\quad C\ T'. \top$$

where the participating judgments are represented in LF as types.

$$\begin{array}{rcllcl}
\ulcorner\text{is expression }\urcorner & = & \mathrm{exp} & \mathrm{exp} & : & \mathrm{type} \\
\ulcorner\text{is index expression }\urcorner & = & \mathrm{iexp} & \mathrm{iexp} & : & \mathrm{type} \\
\ulcorner\text{is type }\urcorner & = & \mathrm{tp} & \mathrm{tp} & : & \mathrm{type} \\
\ulcorner\text{is index prop }\urcorner & = & \mathrm{prop} & \mathrm{prop} & : & \mathrm{type} \\
\ulcorner\text{is index sort}\urcorner & = & \mathrm{sort} & \mathrm{sort} & : & \mathrm{type} \\
\ulcorner\phi \vdash i : \gamma\urcorner & = & \mathrm{ofi}\ulcorner i\urcorner\ulcorner\gamma\urcorner & \mathrm{ofi} & : & \mathrm{idx} \to \mathrm{sort} \to \mathrm{type} \\
\ulcorner\phi; \Gamma \vdash e : \tau\urcorner & = & \mathrm{of}\ulcorner e\urcorner\ulcorner\tau\urcorner & \mathrm{of} & : & \mathrm{exp} \to \mathrm{tp} \to \mathrm{type} \\
\ulcorner\|e\| = e'\urcorner & = & \ulcorner e\urcorner \stackrel{\mathrm{exp}}{\Longrightarrow} \ulcorner e'\urcorner & \stackrel{\mathrm{exp}}{\Longrightarrow} & : & \mathrm{exp} \to \mathrm{exp} \to \mathrm{type} \\
\ulcorner\|e\|_* = e'\urcorner & = & \ulcorner e\urcorner \stackrel{\mathrm{exp}^*}{\Longrightarrow} \ulcorner e'\urcorner & \stackrel{\mathrm{exp}^*}{\Longrightarrow} & : & \mathrm{exp} \to \mathrm{exp} \to \mathrm{type} \\
\ulcorner\|\tau\| = \tau'\urcorner & = & \ulcorner \tau\urcorner \stackrel{\mathrm{tp}}{\Longrightarrow} \ulcorner \tau'\urcorner & \stackrel{\mathrm{tp}}{\Longrightarrow} & : & \mathrm{tp} \to \mathrm{tp} \to \mathrm{type} \\
\ulcorner\|\tau\|_* = \tau'\urcorner & = & \ulcorner \tau\urcorner \stackrel{\mathrm{tp}^*}{\Longrightarrow} \ulcorner \tau'\urcorner & \stackrel{\mathrm{tp}^*}{\Longrightarrow} & : & \mathrm{tp} \to \mathrm{tp} \to \mathrm{type}
\end{array}$$

From this list, only 'of' and 'ofi' deserve special attention. They are hypothetical judgments which means that the contexts $(\phi; \Gamma)$ and ϕ are represented respectively by the means of the LF context:

$$\begin{array}{rcl}
\ulcorner\phi, a : \gamma\urcorner & = & \ulcorner\phi\urcorner, a : \mathrm{idx}, u_a : \mathrm{ofi}\ a\ \ulcorner\gamma\urcorner \\
\ulcorner\phi, P\urcorner & = & \ulcorner\phi\urcorner, \ulcorner P\urcorner : \mathrm{prop} \\
\ulcorner\phi; \Gamma, x : \tau\urcorner & = & \ulcorner\phi; \Gamma\urcorner, x : \mathrm{exp}, u_x : \mathrm{of}\ x\ \ulcorner\tau\urcorner \\
\ulcorner\phi; \Gamma, f : \tau\urcorner & = & \ulcorner\phi; \Gamma\urcorner, f : \mathrm{exp}, u_f : \mathrm{of}\ f\ \ulcorner\tau\urcorner
\end{array}$$

The meta-logic $\mathcal{M}_2^+$ is well understood however not yet implemented in Twelf. Therefore we prefer for this presentation to encode the proof as a primitive recursive relation relating the different input and output objects, which can be executed in Twelf as a logic program. Concretely, the type of this relation is

$$\mathrm{transform} : \mathrm{of}\ E\ T \to E \stackrel{\mathrm{exp}}{\Longrightarrow} C \to T \stackrel{\mathrm{tp}^*}{\Longrightarrow} T' \to \mathrm{of}\ C\ T' \to \mathrm{type}$$

All LF encodings presented this paper are adequate, which means that canonical LF objects stand in one-to-one relation with derivations they represent. Following standard practice (Pfenning 1991) we omit all implicit Π-abstractions from types and we take $\beta\eta$-conversion as the notion of definitional equality (Coquand 1991). For brevity reasons we discuss only one case:

$$
\begin{array}{lll}
\text{trans\_app} & : & \text{transform (of\_app } D_1\ D_2\text{) (r\_app } P_1\ P_2\text{) (r* } Q_2'\text{)} \\
& & \qquad\quad \text{(of\_lam } (\lambda k.\, \lambda u.\, \text{of\_app } R_1 \\
& & \qquad\qquad\qquad \text{(of\_lam } (\lambda x_1.\, \lambda u_{x_1}.\, \text{of\_app } R_2 \\
& & \qquad\qquad\qquad\qquad\quad \text{(of\_lam } (\lambda x_2.\, \lambda u_{x_2}.\, \text{of\_app (of\_app } u_{x_1}\ u_{x_2}\text{) } u\text{))))))} \\
& & \leftarrow \text{transform } D_2\ P_2\ \text{(r* } Q_2\text{) } R_2 \\
& & \leftarrow \text{transform } D_1\ P_1\ \text{(r* (r\_arrow } Q_1'\ \text{(r* } Q_2'\text{))) } R_1
\end{array}
$$

where we only assign types to the relevant constants in this case.

$$
\begin{array}{rcl}
\text{r*} & : & T \stackrel{\text{tp*}}{\Longrightarrow} T' \rightarrow T \stackrel{\text{tp}}{\Longrightarrow} \text{(arr (arr } T'\text{ ans) ans)} \\
\text{r\_app} & : & E_1 \stackrel{\text{exp*}}{\Longrightarrow} E_1' \rightarrow E_2 \stackrel{\text{exp*}}{\Longrightarrow} E_2' \\
& & \rightarrow \text{(app } E_1\ E_2\text{)} \stackrel{\text{exp*}}{\Longrightarrow} \text{(lam } (\lambda k.\ \text{app } E_1'\ \text{(lam } (\lambda x_1.\, \text{app } E_2'\text{(lam } \lambda x_2.\, \text{(app (app } x_1\ x_2\text{) } k\text{))))))} \\
\text{r\_arrow} & : & T_1 \stackrel{\text{tp*}}{\Longrightarrow} T_1' \rightarrow T_2 \stackrel{\text{tp}}{\Longrightarrow} T_2' \rightarrow \text{(arr } T_1\ T_2\text{)} \stackrel{\text{tp*}}{\Longrightarrow} \text{(arr } T_1'\ T_2'\text{)} \\
\\
\text{of\_lam} & : & (\Pi x : \text{exp.\,of } x\ T_1 \rightarrow \text{of } (E\ x)\ T_2) \rightarrow \text{of (lam } E\text{) (arr } T_1\ T_2\text{)} \\
\text{of\_app} & : & \text{of } E_1\ \text{(arr } T_2\ T_1\text{)} \rightarrow \text{of } E_2\ T_2 \rightarrow \text{of (app } E_1\ E_2\text{) } T_1
\end{array}
$$

The logical program 'transform' can be executed in Twelf on a derivation $\mathcal{D}$ of a judgment $\phi; \Gamma \vdash e_2 : \tau_1$ by evoking

$$\text{transform } \ulcorner \mathcal{D} \urcorner\ P\ Q\ R$$

Internally, Twelf matches $\ulcorner D \urcorner$ against (of_app D_1 D_2). If successful, it performs two recursive calls, one on D_1 and the other on D_2. The function returns three result objects P, Q, and R after inverting and assembling the result of the recursive call. By adequacy, P and Q correspond to the respective CPS transformations on expressions and types, and R to the proof that the CPS transform is type correct. The following is a detailed example.

Assume that $\mathcal{D}_1$ and $\mathcal{D}_2$ are type derivations for $\phi; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ and $\phi; \Gamma \vdash e_2 : \tau_1$. We can construct D_6 using $\ulcorner \mathcal{D}_1 \urcorner$ and $\ulcorner \mathcal{D}_2 \urcorner$ as follows, where ans is for $\ulcorner ans \urcorner$.

$$
\begin{array}{lcl}
D_1 & = & \text{of\_app(ans)}(\ulcorner \|\tau_2\|_* \rightarrow ans \urcorner)(\ulcorner x_1(x_2) \urcorner)(k)(\text{of\_app}(\ulcorner \|\tau_2\| \urcorner)(\ulcorner \|\tau_1\|_* \urcorner)(x_1)(x_2)(u_{x_1})(u_{x_2}))(u_k) \\
D_2 & = & \text{of\_lam}(\ulcorner \|\tau_1\|_* \urcorner)(\text{ans})(\lambda x_2 : \text{exp.}\ulcorner x_1(x_2)(k) \urcorner)(\lambda x_2 : \text{exp.}\lambda u_{x_2} : \texttt{of}\ x_2\ \ulcorner \|\tau_1\|_* \urcorner.D_1) \\
D_3 & = & \text{of\_app(ans)}(\ulcorner \|\tau_2\|_* \rightarrow ans \urcorner)(\ulcorner e_2 \urcorner)(\ulcorner \lambda_e\, x_2.x_1(x_2)(k) \urcorner)(\ulcorner \mathcal{D}_2 \urcorner)(D_2) \\
D_4 & = & \text{of\_lam}(\ulcorner \|\tau_1 \rightarrow \tau_2\|_* \urcorner)(\text{ans})(\lambda x_1 : \text{exp.}\ulcorner e_2(\lambda_e\, x_2.x_1(x_2)(k)) \urcorner)(\lambda x_1 : \text{exp.}\lambda u_{x_1} : \texttt{of}\ x_1\ \ulcorner \|\tau_1 \rightarrow \tau_2\|_* \urcorner.D_3) \\
D_5 & = & \text{of\_app(ans)}(\ulcorner \|\tau_1 \rightarrow \tau_2\|_* \rightarrow ans \urcorner)(\ulcorner e_1 \urcorner)(\ulcorner \lambda_e\, x_1.e_2(\lambda_e\, x_2.x_1(x_2)(k)) \urcorner)(\ulcorner \mathcal{D}_1 \urcorner)(D_4) \\
D_6 & = & \text{of\_lam}(\ulcorner \|\tau_1 \rightarrow \tau_2\|_* \rightarrow ans \urcorner)(\text{ans}) \\
& & \quad (\lambda k : \text{exp.}\ulcorner e_1(\lambda_e\, x_1.e_2(\lambda_e\, x_2.x_1(x_2)(k))) \urcorner)(\lambda k : \text{exp.}\lambda u_k : \texttt{of}\ k\ \ulcorner \|\tau_1\|_* \rightarrow ans \urcorner.D_5)
\end{array}
$$

5 Related Work and Conclusion

The study on the typing properties of CPS transform was initiated by Meyer and Wand for a call-by-value interpretation of the simply-typed λ-calculus (Meyer and Wand 1985). Subsequently, Harper and Lillibridge studied the typing properties of several CPS transforms in a language that extends F_ω (Girard, J.-Y. 1971) with some control constructs, where explicit polymorphism is supported (Harper and Lillibridge 1993). The CPS transform presented in this paper is most closely related to the ML-CBV CPS transform in (Harper and Lillibridge 1993). However, what is important in our case is probably not the CPS transform itself. Instead, we are interested in lifting the CPS transform from the level of expressions to the level of type derivations, preparing for building a type-preserving compiler for Dependent ML.

People are interested in type-preserving compilation for various reasons (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996; Shao 1997; Tolmach and Oliva 1998; Morrisett et al. 1999), including facilitating compiler debugging, optimizing data representation, leveraging language interoperability, producing certified low-level executable code, etc. We are currently interested in using dependent types in DML to capture important program properties (e.g., memory safety) and then compiling the dependent types into low-level to certify that the generated low-level code possesses the captured program properties. This opens a promising approach to effectively generating proofs for proof-carrying code (Necula 1997).

When studying type-preserving compilation for DML, we notice that the (essentially) same constraints have to be solved at both source and target levels if we follow the current practice of building type-preserving compilers. This is not only annoying but can also be time-consuming. In this paper, we use CPS transform as an example to demonstrate a viable alternative that compiles type derivations of an expression instead of the expression itself, obviating the need for solving the same constraints repeatedly. In future, we will continue the study on compiling type derivations, constructing a type-preserving compiler for DML that can eventually produce certifiable binary code.

References

Coquand, T. (1991). An algorithm for testing conversion in type theory. In G. Plotkin and G. Huet (Eds.), *Logical Frameworks*, pp. 255–279. Cambridge University Press.

Dowek, G., A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner (1993). The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France. Version 5.8.

Girard, J.-Y. (1971). Une Extension de l'Interprétation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types. In J. E. Fenstad (Ed.), *Proceedings of the Second Scandinavian Logic Symposium*, Volume 63 of *Studies in Logic and the Foundations of Mathematics*, Amsterdam, pp. 63–92. North-Holland.

Harper, R. W., F. Honsell, and G. D. Plotkin (1993, January). A framework for defining logics. *Journal of the ACM 40*(1), 143–184.

Harper, R. W. and M. Lillibridge (1993). Explicit polymorphism and cps conversion. In *Conference Record of the Twentieth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 206–219.

Lawrence, P. (1994). *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828.

Meyer, A. and M. Wand (1985). Continuation Semantics in Typed Lambda Calculi (summary). In R. Parikh (Ed.), *Logics of Programs*, pp. 219–224. Springer-Verlag LNCS 224.

Morrisett, G. et al. (1999). Talx86: A realistic typed assembly language. In *Proceedings of Workshop on Compiler Support for System Software.*

Necula, G. (1997). Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM press.

Paulin-Mohring, C. (1993). Inductive definitions in the system Coq: rules and properties. In M. Bezem and J. de Groote (Eds.), *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, Volume 664 of *Lecture Notes in Computer Science*, pp. 328–345.

Pfenning, F. (1991). Logic programming in the lf logical framework. In I. G. Huet and G. Plotkin (Eds.), *Logical Frameworks*, pp. 149–181. Cambridge University Press.

Schürmann, C. (2000). *Automating the Meta-Theory of Deductive Systems*. Ph. D. thesis, Carnegie Mellon University. CMU-CS-00-146.

Schürmann, C. and F. Pfenning (1998). Automated theorem proving in a simple meta-logic for lf. In *Proceedings of the 15th International Conference on Automated Deduction (CADE)*, pp. 286–300. Springer-Verlag LNCS 1421.

Shao, Z. (1997, June). An Overview of the FLINT/ML compiler. In *Proceedings of ACM SIGPLAN Workshop on Types in Compilation (TIC '97).*

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee (1996, June). A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 181–192.

Tolmach, A. and D. P. Oliva (1998, July). From ML to Ada(!?!): Strongly-typed language interoperability via source translation. *Journal of Functional Programming 8*(4), 367–412.

Xi, H. (1998). *Dependent Types in Practical Programming*. Ph. D. thesis, Carnegie Mellon University. pp. viii+189. Available as
`http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

Xi, H. (1999). Dependent ML. Available at
`http://www.ececs.uc.edu/~hwxi/DML/DML.html`.

Xi, H. and F. Pfenning (1999, January). Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, pp. 214–227.

Xi, H. and C. Schürmann (2001). CPS Transform and Type Derivations in Dependent ML. Available as
`http://www.ececs.uc.edu/~hwxi/academic/papers/DMLcps.ps`.