Implementing Typeful Program Transformations*

Chiyan Chen and Hongwei Xi
Computer Science Department
Boston University
{chiyan, hwxi}@cs.bu.edu

ABSTRACT

The notion of program transformation is ubiquitous in programming language studies on interpreters, compilers, partial evaluators, etc. In order to implement a program transformation, we need to choose a representation in the meta language, that is, the programming language in which we construct programs, for representing object programs, that is, the programs in the object language on which the program transformation is to be performed. In practice, most representations chosen for typed object programs are typeless in the sense that the type of an object program cannot be reflected in the type of its representation. This is unsatisfactory as such typeless representations make it impossible to capture in the type system of the meta language various invariants in a program transformation that are related to the types of object programs. In this paper, we propose an approach to implementing program transformations that makes use of a first-order typeful program representation formed in Dependent ML (DML), where the type of an object program as well as the types of the free variables in the object program can be reflected in the type of the representation of the object program. We introduce some programming techniques needed to handle this typeful program representation, and then present an implementation of a CPS transform function where the relation between the type of an object program and that of its CPS transform is captured in the type system of DML. In a broader context, we claim to have taken a solid step along the line of research on constructing certifying compilers.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Theory

Keywords

Typeful Program Transformation, Dependent Types, Dependent ML, DML, Continuation-Passing Style, CPS

*Partially supported by the NSF Grants No. CCR-0224244 and No. CCR-0229480

1. INTRODUCTION

The notion of program transformation is frequently encountered in various programming language studies on interpreters, compilers, partial evaluators, etc. When implementing a program transformation in a meta language that acts on programs in an object language, we immediately face the question as to what representation needs to be chosen in the meta language for representing object programs. In the case where the meta language is a functional programming language such as Standard ML (SML) [10] or Haskell [14], we often choose to declare a datatype for representing object programs. For instance, we may declare the following datatype *EXP* in SML for representing pure simply typed λ-expressions.

```
datatype EXP =
  EXPvar of string
| EXPlam of string * EXP
| EXPapp of EXP * EXP
```

As an example, the following simply typed λ-expression (with Church typing):

$$\lambda x : int.\lambda y : int \rightarrow int.y(x)$$

is represented as follows:

$$EXPlam(\text{"}x\text{"}, EXPlam(\text{"}y\text{"}, EXPapp(EXPvar\ \text{"}y\text{"}, EXPvar\ \text{"}x\text{"})))$$

Unfortunately, such a representation contains some serious problems that may adversely affect its use in implementing program transformations. For instance, this is a typeless representation for simply typed λ-expressions as all typing information in a simply typed λ-expression is completely lost in the type of its representation, which is always *EXP*. Also, there are more values of type *EXP* than simply typed λ-expressions. As an example, the following value:

$$EXPlam(\text{"}x\text{"}, EXPapp(EXPvar\ \text{"}x\text{"}, EXPvar\ \text{"}x\text{"}))$$

does not correspond to any simply typed λ-expression. Furthermore, it is impossible to tell from the type of the representation of an expression whether the expression contains

.

```
fun subst0 x sub e =
  case e of
    Var x' => if x = x' then sub else e
  | Lam (x, e') =>
    if x = x' then e else Lam (x', subst0 x sub e')
  | App (e1, e2) =>
    App (subst0 x sub e1, subst0 x sub e2)
withtype string -> EXP -> EXP -> EXP
```

Figure 1: A typeless implementation of a substitution function

free variables. These problems arise when we try to implement a function $subst_0$ whose intended use is to substitute a *closed* simply typed λ-expression M_0 for a given variable x in another simply typed λ-expression M, where the type of M_0 is required to be the same as the type assigned to x. In Figure 1, we give a possible implementation of $subst_0$ in SML. The `withtype` clause is a slight addition to the syntax of SML, which we use to supply a type annotation. For instance, the type annotation in Figure 1 indicates that the function $subst_0$ is assigned the type $string \to EXP \to EXP \to EXP$. Obviously, when $subst_0(x)(e_0)(e)$ is called, we can enforce neither the requirement that e_0 represent a closed expression nor the one that the type of the expression represented by e_0 be the same as the type assigned to the variable represented by x. Certainly, we can insert some checks to enforce these requirements at run-time, but this solution is unsatisfactory as such checks, which may cause potential efficiency concerns, can offer no assistance in capturing program errors at compile-time.

In this paper, we propose an approach to implementing program transformations that makes use of a first-order typeful program representation in which program variables are replaced with deBruijn indexes [6]. With this approach, we can encode the type of an object program as well as the types of the free variables in the object program into the type of the representation of the object program. As a result, we can expect to assign more informative types to functions that transform programs, thus capturing more invariants in program transformations. We first form a datatype in Dependent ML (DML) for representing simply typed λ-expressions in a typeful manner. We specifically choose first-order abstract syntax (f.o.a.s.) over higher-order abstract syntax (h.o.a.s.) [15] as program transformations may often need to be performed on open programs, that is, programs containing free variables, and we are currently unclear about how to make h.o.a.s. interact with free program variables in a satisfactory manner. We also develop some programming techniques to facilitate the implementation of program transformations, which constitute the main contribution of the paper. In particular, we present a typeful implementation of a CPS transform function, where the type assigned to the implementation captures the relation between the type of an object program and that of its transform.

The rest of the paper is organized as follows. We first form a first-order typeful program representation in Section 2 for the simply typed λ-calculus, where variables in λ-expressions are replaced with deBruijn indexes. We then show in Section 3 how substitution can be implemented with this program representation. In Section 4, we present a typeful implementation of a CPS transform function *à la* Plotkin, where we assume the existence of some primitive recursive functions on type indexes, and then eliminate the need for such functions in Section 5, making the implementation of the CPS transform typable in DML($\mathcal{L}_{alg}$) for some type index language $\mathcal{L}_{alg}$ consisting of sorted algebraic terms. Lastly, we mention some related work and conclude.

2. A FIRST-ORDER TYPEFUL PROGRAM REPRESENTATION

In this section, we introduce a first-order typeful program representation for simply typed λ-expressions. The syntax for the simple typed λ-calculus (with Curry typing) can be readily given as follows, where we use x for variables and b for some base types.

$$\begin{array}{lrcl} \text{types} & \tau & ::= & b \mid \tau_1 \to \tau_2 \\ \text{expressions} & M & ::= & x \mid \lambda x.M \mid M_1(M_2) \\ \text{contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x:\tau \end{array}$$

We require that a variable be declared at most once in a given context Γ and use $\mathbf{dom}(\Gamma)$ for the set of variables declared in Γ. As usual, we have the following typing rules for the simply typed λ-calculus, where we write $\Gamma(x) = \tau$ to mean that the variable x is assigned the type τ in the context Γ.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textbf{(ty-var)}$$

$$\frac{\Gamma, x:\tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1(M_2) : \tau_2}\ \textbf{(ty-app)}$$

Instead of representing λ-expressions directly, which contain no typing information, we are to represent typing derivations of λ-expressions. For this purpose, we introduce a judgment $\Gamma \vdash_0 x : \tau$ to mean that x is declared to be of the type τ in the context Γ. The rules for deriving such a judgment is given as follows.

$$\frac{}{\Gamma, x:\tau \vdash_0 x : \tau}\ \textbf{(ty-var-one)}$$

$$\frac{\Gamma \vdash_0 x : \tau_1}{\Gamma, x':\tau_2 \vdash_0 x : \tau_1}\ \textbf{(ty-var-shi)}$$

We can now change the rule **(ty-var)** into the following one:

$$\frac{\Gamma \vdash_0 x : \tau}{\Gamma \vdash x : \tau}\ \textbf{(ty-var)}$$

In the rest of the paper, we assume that *int* is the only base type we have. The assumption is solely for simplifying the presentation, and it is straightforward to handle additional base types.

In Dependent ML [21, 19], a language schema is presented for extending ML with a restricted form of dependent types where type index expressions are required to be drawn from a given constraint domain or a type index language as we now call it. We use DML($\mathcal{L}$) for such an extension, where $\mathcal{L}$ denotes the given type index language. In the rest of the paper, we use $\mathcal{L}_{alg}$ for a type index language consisting of sorted algebraic terms. In particular, we assume the following sorts *ty* and *env* are declared in $\mathcal{L}_{alg}$.

```
datatype VAR (env, ty) =
  | {G:env, t:ty}. VARone (t :: G, t)
  | {G:env, t1:ty, t2:ty}.
    VARshi (t2 :: G, t1) of VAR (G, t1)

datatype EXP (env, ty) =
  | {G:env, t:ty}. EXPvar (G, t) of VAR (G, t)
  | {G:env, t1:ty, t2:ty}.
      EXPlam (G, t1 -> t2) of EXP (t1 :: G, t2)
  | {G:env, t1:ty, t2:ty}.
      EXPapp (G, t2) of
        EXP (G, t1 -> t2) * EXP (G, t1)
```

Figure 2: Two datatypes for representing typing derivations of the simply typed λ-expressions

```
infixr -> ::
sort ty = int | -> of (ty, ty)
sort env = nil | :: of (ty, env)
```

The concrete syntax simply indicates that

- $\rightarrow$ and :: are infix operators with right associativity, and
- a type index i is of sort ty in $\mathcal{L}_{alg}$ if it is int or $i_1 \rightarrow i_2$ for some type indexes i_1 and i_2 of sort ty, and
- a type index i is of sort env in $\mathcal{L}_{alg}$ if it is nil or $i_1 :: i_2$ for some type indexes i_1 and i_2 of sorts ty and env, respectively.

Intuitively, we use type indexes of sorts ty and env to represent types and contexts, respectively, in the simply typed λ-calculus. For instance, $int :: (int \rightarrow int) :: nil$ represents a context in which the first and the second variables are assigned the type int and $int \rightarrow int$, respectively. In general, given a context $\Gamma = \emptyset, x_1 : \tau_1, \ldots, x_n : \tau_n$, we use $t_n :: \ldots :: t_1 :: nil$ to represent Γ, assuming t_i represents τ_i for each $1 \leq i \leq n$.

We are now ready to declare in Figure 2 two dependent datatypes for representing typing derivations of simply typed λ-expressions. The keyword `datatype` is used to declare dependent datatypes. The type constructor *VAR* takes a type index G of sort env and a type index t of sort ty to form a type $VAR(G, t)$ for values representing typing derivations of $\Gamma \vdash_0 x : \tau$, where we assume G and t represent Γ and τ, respectively. The two value constructors *VARone* and *VARshi* associated with *VAR* are assigned the following two types, respectively:

$$\Pi G : env.\Pi t : ty.\, VAR(t :: G, t)$$
$$\Pi G : env.\Pi t_1 : ty.\Pi t_2 : ty.\, VAR(G, t_1) \rightarrow VAR(t_2 :: G, t_1)$$

Note that *VARone* and *VARshi* correspond to the typing rules **(ty-var-one)** and **(ty-var-shi)**, respectively. Essentially, we use *VARone* and *VARshi* to form deBruijn indexes for variables: *VARone* stands for the deBruijn index 1, referring to the first variable in a context, and *VARshi* stands for the shift operator that increases a given deBruijn index by 1. For example, $VARshi(VARshi(VARone))$ represents the deBruijn index 3, referring to the third variable in a context.

Like *VAR*, the type constructor *EXP* also takes a type index G of sort env and a type index t of sort ty to form a type $EXP(G, t)$ for values representing typing derivations of $\Gamma \vdash M : \tau$, where we assume G and t represent Γ and τ, respectively. There are three value constructors associated with *EXP*, which are assigned the following types respectively.

$$\begin{array}{rcl} EXPvar & : & \Pi G : env.\Pi t : ty.\, VAR(G, t) \rightarrow EXP(G, t) \\ EXPlam & : & \Pi G : env.\Pi t_1 : ty.\Pi t_2 : ty. \\ & & \quad EXP(t_1 :: G, t_2) \rightarrow EXP(G, t_1 \rightarrow t_2) \\ EXPapp & : & \Pi G : env.\Pi t_1 : ty.\Pi t_2 : ty \\ & & \quad EXP(G, t_1 \rightarrow t_2) * EXP(G, t_1) \rightarrow EXP(G, t_2) \end{array}$$

The value constructors *EXPvar*, *EXPlam* and *EXPapp* correspond to the typing rules **(ty-var)**, **(ty-lam)**, and **(ty-app)**, respectively. For instance, the expression:

$$\lambda x : int.\lambda y : int \rightarrow int.y(x)$$

can be represented as follows,[1]

$$\begin{array}{l} EXPlam(EXPlam(\\ \quad EXPapp(EXPvar(\, VARone), \\ \qquad EXPvar(\, VARshi(\, VARone))))) \end{array}$$

where the value is assigned the following type:

$$EXP(nil, int \rightarrow (int \rightarrow int) \rightarrow int).$$

In contrast to the typeless representation mentioned in Section 1, we have now formed a typeful representation for simply typed λ-expressions as the type of a simply typed λ-expression can be reflected in the type of its representation. We next develop some techniques for implementing program transformations with this typeful representation.

3. IMPLEMENTING SUBSTITUTION

Substitution plays a key rôle in implementing program transformations. In this section, we present a typeful implementation of substitution for simply typed λ-calculus, setting up the machinery for further development.

We first declare a type definition as follows to facilitate presentation.

```
typedef SUB (G1:env, G2:env) =
          {t:ty} VAR(G1,t) -> EXP(G2,t)
```

With this declaration, we can write $SUB(G_1, G_2)$ to represent the type $\Pi t : ty.\, VAR(G_1, t) \rightarrow EXP(G_2, t)$, where G_1 and G_2 are type indexes of sort env. For instance, the functions *idSub* and *shiSub* defined below correspond the standard substitutions id and $\uparrow$ as are described in [1], respectively,

```
fun idSub x = EXPvar (x)
fun shiSub x = EXPvar (VARshi x)
```

and they can be assigned the following types:

$$\Pi G : env.SUB(G, G) \quad \text{and} \quad \Pi G : env.\Pi t : ty.SUB(G, t :: G)$$

respectively.

We implement two functions *subst* and *subLam* mutually recursively in Figure 3. To provide some explanation for these functions, we briefly turn to substitution for untyped λ-expressions in deBruijn's notation [6], whose syntax is given as follows.

$$\begin{array}{lrcl} \text{indexes} & n & ::= & 1 \mid 2 \mid \cdots \\ \text{expressions} & N & ::= & n \mid \lambda.N \mid N_1(N_2) \end{array}$$

[1]We have omitted explicit applications to type indexes.

```
fun subst sub e =
  case e of
      EXPvar x => sub (x)
    | EXPlam e =>
      EXPlam (subst (subLam sub) e)
    | EXPapp (e1, e2) =>
      EXPapp (subst sub e1, subst sub e2)
withtype
  {G1:env,G2:env,t:ty}.
    SUB(G1,G2) -> EXP(G1,t) -> EXP(G2,t)

and subLam sub =
  fn VARone => EXPvar (VARone)
   | VARshi x' => subst shiSub (sub (x'))
withtype
  {G1:env,G2:env,t:ty}.
    SUB (G1,G2) -> SUB (t::G1,t::G2)
```

Figure 3: Implementing substitution

```
fun subPre sub e =
  fn VARone => e
   | VARshi x => sub x
withtype
  {G1:env,G2:env,t:ty}.
    SUB(G1, G2) -> EXP (G2, t) -> SUB (t :: G1, G2)

fun subComp sub1 sub2 =
  fn x => subst sub2 (sub1 x)
withtype
  {G1:env,G2:env,G3:env}.
    SUB(G1,G2) -> SUB(G2,G3) -> SUB(G1,G3)
```

Figure 4: Two functions on substitutions

For instance, the deBruijn's notation for the λ-expression $\lambda x.\lambda y.y(x)$ is $\lambda.\lambda.1(2)$. We use σ for a substitution, which is a function that maps indexes n to expressions N. We use $\uparrow$ for the substitution that maps each index n to the index $n+1$. Given a substitution σ and an expression N, we write $N[\sigma]$ for the result of applying the substitution σ to N, which is defined as follows,

$$\begin{array}{rcl} n[\sigma] & = & \sigma(n) \\ (\lambda.N)[\sigma] & = & \lambda.N[1\cdot(\sigma\circ\uparrow)] \\ (N_1(N_2))[\sigma] & = & N_1[\sigma](N_2[\sigma]) \end{array}$$

where $\sigma\circ\uparrow$, the composition of the substitutions σ and $\uparrow$, yields the substitution σ' such that $\sigma'(n) = \sigma(n)[\uparrow]$.

For the functions *subst* and *subLam* in Figure 3, we have that $subst(sub)(e)$ and $subLam(sub)$ correspond to $N[\sigma]$ and $1\cdot(\sigma\circ\uparrow)$, respectively, where we assume *sub* and e represent the substitution σ and the expression N, respectively. The operator $\cdot$ for prepending a term to a substitution and the operator $\circ$ for composing two substitutions, which are standard in the study on explicit substitutions [1], can be implemented as the functions *subPre* and *subComp*, respectively, in Figure 4. Note that $\circ$ is associative.

As an interesting example, we implement a function *hnf* in Figure 5 to compute head normal form for the simply typed λ-expressions. The following type is assigned to the function *hnf*,

$$\Pi G : env.\Pi t : ty.EXP(G,t) \to EXP(G,t)$$

```
fun hnf e =
  case e of
      EXPvar _ => e
    | EXPlam e => EXPlam (hnf e)
    | EXPapp (e1, e2) =>
        (case hnf e1 of
             EXPlam e =>
               hnf (subst (subPre idSub e2) e)
           | e1 => EXPapp (e1, e2))
withtype {G:env,t:ty}. EXP(G, t) -> EXP (G, t)
```

Figure 5: Computing head normal form

which indicates that *hnf* is type-preserving.

4. IMPLEMENTING CPS TRANSFORMATION

In this section, we present a typeful implementation of CPS transformation for a source object language that extends the simply typed λ-calculus with two control constructs *callcc* and *throw*. We first extend the syntax of the simply typed λ-calculus as follows,

$$\begin{array}{rcl} \tau & ::= & \cdots \mid cont(\tau) \mid \bot \\ M & ::= & \cdots \mid callcc(M) \mid throw(M_1, M_2) \end{array}$$

where $cont(\tau)$ is the type for continuations that only take values of type τ, and $\bot$ is the empty type that contains no closed values. In addition, we introduce the following typing rules for handling *callcc* and *throw*.

$$\frac{\Gamma \vdash M : cont(\tau) \to \tau}{\Gamma \vdash callcc(M) : \tau}\ \textbf{(ty-callcc)}$$

$$\frac{\Gamma \vdash M_1 : cont(\tau_1) \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash throw(M_1, M_2) : \tau_2}\ \textbf{(ty-throw)}$$

We now need to extend the definition of the sort *ty* to accommodate the type constructor *cont* and the type $\bot$.

```
sort ty = ... | cont of ty | bot
```

We also associate more value constructors with the type constructor *EXP*:

```
datatype EXP (env, ty) =
  ...
  | {G:env,t:ty}.
      EXPcal (G, t) of EXP (G, cont(t) -> t)
  | {G:env,t1:ty,t2:ty}.
      EXPthr (G, t2) of
        EXP (G, cont(t1)) * EXP (G, t1)
```

The value constructors *EXPcal* and *EXPthr* correspond to the typing rules **(ty-callcc)** and **(ty-throw)**, respectively.

We define some transform functions in Figure 6. The functions $T_1(\cdot)$ and $T_2(\cdot)$ are defined mutually inductively on the structure of types, while the functions $cps(\cdot)$ and $cpsw(\cdot,\cdot)$ are defined mutually inductively on the structure on λ-expressions. Note that $cps(\cdot)$ is a CPS transform function *à la* Plotkin [16]. We have omitted some obvious restrictions on the bound variables occurring on the right-hand side of some of the equations in the definition of *cps* and *cpsw*. For instance, the variable x in the following equation needs to have no free occurrences in M:

$$cps(M) \quad = \quad \lambda x.cpsw(x, M)$$

$$
\begin{array}{rcl}
T_1(\tau) & = & (T_2(\tau) \to \bot) \to \bot \\
T_2(b) & = & b \\
T_2(\tau_1 \to \tau_2) & = & T_2(\tau_1) \to T_1(\tau_2) \\
T_2(cont(\tau)) & = & T_2(\tau) \to \bot \\
 & & \\
cps(M) & = & \lambda x.cpsw(x, M) \\
cpsw(M_k, x) & = & M_k(x) \\
cpsw(M_k, \lambda x.M) & = & M_k(\lambda x.cps(M)) \\
cpsw(M_k, M_1(M_2)) & = & \\
 & & cpsw(\lambda x_1.cpsw(\lambda x_2.x_1(x_2)(M_k), M_2), M_1) \\
cpsw(M_k, callcc(M)) & = & cpsw(\lambda x.x(M_k)(M_k), M) \\
cpsw(M_k, throw(M_1, M_2)) & = & cpsw(\lambda x.cpsw(x, M_2), M_1)
\end{array}
$$

Figure 6: A CPS Transform

We have the following theorem that relates the typing derivation of an expression in the source object language to the typing derivation of its CPS transform in the target object language [9, 7].

THEOREM 4.1. *Assume that* $\Gamma \vdash M : \tau$ *is derivable and* Γ' *is a context such that* $\Gamma'(x) = T_2(\Gamma(x))$ *for each* $x \in \mathbf{dom}(\Gamma) \subseteq \mathbf{dom}(\Gamma')$. *Then we have that*

1. $\Gamma' \vdash cps(M) : T_1(\tau)$ *is derivable, and*
2. $\Gamma' \vdash cpsw(M_k, M) : \bot$ *is also derivable if* $\Gamma' \vdash M_k : T_2(\tau) \to \bot$ *is derivable.*

PROOF. We can prove (1) and (2) simultaneously by structural induction on the typing derivation of $\Gamma \vdash M : \tau$. □

We are to present an implementation of the CPS transform *cps* in DML($\mathcal{L}_{alg}$) such that the type assigned to the implementation captures Theorem 4.1.

It may be surprising that the most difficult question we face is to implement the the following equation in the definition of *cpsw*:

$$cpsw(M_k, x) \quad = \quad M_k(x)$$

With a careful inspection of the equation, we observe that the variable x in the left-hand side of the equation is distinct from the one in the right-hand side: the former is in a source object program while the latter is in a target object program, and the types assigned to them are also different. In a setting where named variables are used, these two x's are implicitly related through their common name. How do we then relate two variables represented as deBruijn indexes? To address this issue, we introduce a concept called *variable mapping*, which is closely related to the notion of substitution. We declare a type definition *VM* as follows, where the function T_2 is defined in Figure 6 through primitive recursion on the structure of type indexes of sort *ty*.

```
typedef VM (G1:env, G2:env) =
          {t:ty}. VAR (G1, t) -> VAR (G2, T2(t))
```

Unfortunately, there is a serious problem with this type definition as we currently do not allow primitive recursion in the type index language $\mathcal{L}_{alg}$.[2] Therefore, the code in the rest of this section cannot be handled in DML($\mathcal{L}_{alg}$). However, we will present a method in the next section to eliminate the need for T_2, making the code typable in DML($\mathcal{L}_{alg}$).

We use *vm* for a variable mapping, which is simply a function of type $VM(G_1, G_2)$ for some type indexes G_1 and G_2 of sort *env*. We implement some functions as follows for manipulating variable mappings.

```
fun vmShi vm = fn x => VARshi (vm x)
withtype
  {G1:env,G2:env,t:ty}. VM(G1,G2) -> VM (G1,t::G2)

fun vmLam vm =
  fn VARone => VARone
   | VARshi x => VARshi (vm x)
withtype
  {G1:env,G2:env,t:ty}.
    VM(G1, G2) -> VM (t::G1, T2(t)::G2)
```

Clearly, *vmLam* is just the counterpart of *subLam*.

We declare a datatype *EXP′* as follows for representing expressions in the target object language of the CPS transformation, which is just the simply typed λ-calculus. So *EXP′* is the same as *EXP* before *EXP* is extended, and the functions defined in Section 3 can simply be carried over.

```
datatype EXP' (env, ty) =
  | {G:env, t:ty}. EXPvar' (G, t) of VAR (G, t)
  | {G:env, t1:ty, t2:ty}.
      EXPlam' (G, t1 -> t2) of EXP' (t1 :: G, t2)
  | {G:env, t1:ty, t2:ty}.
      EXPapp' (G, t2) of
        EXP' (G, t1 -> t2) * EXP' (G, t1)
```

We now implement the functions *cpsw* and *cps* in Figure 7. The function *expShi* is needed to increase the deBruijn index of each free variable in a given expression by 1. To provide some explanation for the implementation *cps* and *cpsw*, we again turn to untyped λ-calculus in deBruijn's notation. We use ν for a variable mapping, which maps deBruijn indexes to deBruijn indexes. We can now define *cps* and *cpsw* in Figure 8 for λ-expressions in deBruijn's notation. This time, *cps* and *cpsw* take a variable mapping ν as an extra argument that is needed to relate deBruijn indexes in the source object program to those in the target object program, and it is straightforward to observe a direct correspondence between the implementation of *cps* and *cpsw* in Figure 7 and their definition in Figure 8.

5. ELIMINATING THE NEED FOR FUNCTIONS ON TYPE INDEXES

As mentioned earlier, T_1 and T_2, which are primitive recursive functions on type indexes, are not available in the type index language $\mathcal{L}_{alg}$, for otherwise it is in general undecidable to determine whether two given type indexes are equal. In this section, we show how the need for T_1 and T_2 can be eliminated, making the implementation of a CPS transform in the previous section typable in DML($\mathcal{L}_{alg}$).

We declare two dependent datatypes RT_1 and RT_2 in Figure 9. If there is a closed value of type $RT_1(t, t')$, then $t' = T_1(t)$. Similarly, if there is a closed value of type $RT_2(t, t')$, then $t' = T_2(t)$. We may use the name *proof term* to refer to a value of type $RT_1(t, t')$ or $RT_2(t, t')$ for some type indexes t and t'. We now change the type definition *VM* to the following one,

[2]It would otherwise make it undecidable or simply intractable to determine whether two type indexes are equal!

```
val EXPone' = EXPvar' (VARone)
withtype {G:env,t:ty}. EXP' (t :: G, t)

val EXPtwo' = EXPvar' (VARshi VARone)
withtype
  {G:env,t1:ty,t2:ty}. EXP' (t1 :: t2 :: G, t2)

fun expShi e = subst shiSub e
withtype
  {G:env,t1:ty,t2:ty} EXP'(G, t1) -> EXP'(t2::G,t1)

fun cps vm e = EXPlam' (cpsw (vmShi vm) EXPone' e)
withtype
  {G1:env,G2:env,t:ty}
    VM(G1,G2) -> EXP(G1,t) -> EXP'(G2,T1(t))

and cpsw vm k e =
  case e of
    EXPvar v => EXPapp' (k, EXPvar' (vm v))
  | EXPlam (e) =>
    EXPapp' (k, EXPlam' (cps (vmLam vm) e))
  | EXPapp (e1, e2) =>
      let
         val k =
           EXPlam' (
             EXPapp' (
               EXPapp' (EXPtwo', EXPone'),
               expShi (expShi k)
             )
           )
         val k = EXPlam' (cpsw (vmShi vm) k e2)
      in
         cpsw vm k e1
      end
   | EXPcal e =>
      let
         val k = expShi k
         val k =
           EXPlam' (
             EXPapp' (EXPapp' (EXPone', k), k)
           )
      in
         cpsw vm k e
      end
   | EXPthr (e1, e2) =>
      let
         val k = cpsw (vmShi vm) EXPone' e2
      in
         cpsw vm k e1
      end
withtype
  {G1:env, G2:env, t:ty}
    VM(G1, G2) -> EXP'(G2, T2(t) -> bot) ->
    EXP(G1, t) -> EXP'(G2, bot)
```

Figure 7: Implementing a CPS transform function

$$
\begin{aligned}
cps(\nu, N) &= \lambda.cpsw(\nu \circ \uparrow, 1, N) \\
cpsw(\nu, N_k, n) &= N_k(\nu(n)) \\
cpsw(\nu, N_k, \lambda.N) &= N_k(\lambda.cps(1 \cdot (\nu \circ \uparrow), N)) \\
cpsw(\nu, N_k, N_1(N_2)) &= \\
&\quad cpsw(\nu, \lambda.cpsw(\nu \circ \uparrow, \lambda.2(1)(N_k[\uparrow][\uparrow]), N_2), N_1) \\
cpsw(\nu, N_k, callcc(N)) &= \\
&\quad cpsw(\nu, \lambda.1(N_k[\uparrow])(N_k[\uparrow]), N) \\
cpsw(\nu, N_k, throw(N_1, N_2)) &= \\
&\quad cpsw(\nu, \lambda.cpsw(\nu \circ \uparrow, 1, N_2), N_1)
\end{aligned}
$$

Figure 8: A CPS Transform for λ-expressions in de-Bruijn's notation

```
datatype RT1 (ty, ty) =
  | {t:ty,t':ty}.
      RT1 (t, (t' -> bot) -> bot) of RT2 (t, t')

and RT2 (ty, ty) =
  | RT2int (int, int)
  | {t1:ty,t2:ty,t1':ty,t2':ty}.
      RT2fun (t1 -> t2, t1' -> t2') of
        RT2(t1, t1') * RT1 (t2, t2')
  | {t:ty,t':ty}.
      RT2cont(cont(t), t' -> bot) of RT2 (t, t')
  | RT2bot (bot, bot)
```

Figure 9: Two dependent datatypes for T_1 and T_2

```
typedef VM (G:env, G':env) =
  {t:ty,t':ty} RT2(t,t') -> VAR(G,t) -> VAR(G',t')
```

and also modify the previous functions *vmShi* and *vmLam* as follows.

```
fun vmShi vm = fn pf => fn v => VARshi (vm pf v)
withtype
  {G1:env,G2:env,t:ty}.
    VM (G1, G2) -> VM (G1, t :: G2)

fun vmLam pf0 vm =
  fn pf =>
    fn VARone => (* a dynamic check is needed *)
       pf0 = pf >> VARone
     | VARshi v => VARshi (vm pf v)
withtype
  {G1:env,G2:env,t1:ty,t2:ty}.
    RT2 (t1, t2) -> VM (G1, G2) ->
    VM (t1 :: G1, t2 :: G2)
```

There is a subtle point in the implementation of *vmLam*. In the body of the function *vmLam*, the syntax $pf_0 = pf \gg VARone$ indicates that *VARone* can be returned only if $pf_0 = pf$ evaluates to the boolean value *true*, and a run-time exception is raised otherwise. To see the reason for such a run-time check, which bears some similarity to typed dynamic typing as is proposed in [4], we give some informal explanation on how *vmLam* is type-checked.

Assume pf_0 and *vm* are assigned the types $RT_2(t_1, t_2)$ and $VM(G_1, G_2)$, respectively, where t_1 and t_2 are type index variables of sort *ty* and G_1 and G_2 are type index variables of sort *env*. We need to show the the body of *vmLam* can be assigned the type $VM(t_1 :: G_1, t_2 :: G_2)$, which expands

```
datatype TY (ty) =
  | TYint (int)
  | {t1:ty,t2:ty}.
      TYfun (t1 -> t2) of TY (t1) * TY (t2)
  | {t: ty} TYcont (cont(t)) of TY(t)
  | TYbot (bot)

datatype EXP (env, ty) =
  | {G:env, t:ty}. EXPvar (G, t) of VAR (G, t)
  | {G:env, t1:ty, t2:ty}.
      EXPlam (G, t1 -> t2) of EXP (t1 :: G, t2)
  | {G:env, t1:ty, t2:ty}.
      EXPapp (G, t2) of
        TY (t1) * EXP (G, t1 -> t2) * EXP (G, t1)
  | {G:env,t:ty}.
      EXPcal (G, t) of EXP (G, cont(t) -> t)
  | {G:env,t1:ty,t2:ty}.
      EXPthr (G, t2) of
        TY (t1) * EXP (G, cont(t1)) * EXP (G, t1)
```

Figure 10: Two dependent datatypes for representing types and expressions in the source language

into the following one:

$$\Pi t : ty.\Pi t' : ty.\\ RT_2(t,t') \rightarrow VAR(t_1 :: G_1, t) \rightarrow VAR(t_2 :: G_2, t')$$

So we assume that pf is assigned the type $RT_2(t,t')$ for type index variables of sort ty. Without the run-time check, we need to assign *VARone* the type $VAR(t_2 :: G_2, t')$ under the assumption that the pattern *VARone* is of the type $VAR(t_1 :: G_1, t)$; unfortunately, this fails as we cannot prove that $t_2 = t'$, which is required in order to assign *VARone* the type $VAR(t_2 :: G_2, t')$. With the run-time test, we can assume that $RT_2(t_1, t_2)$, the type of pf_0, and $RT_2(t,t')$, the type of pf are the same when assigning *VARone* the type $VAR(t_2 :: G_2, t')$; this succeeds as $t_2 = t'$ can be inferred from the assumption $RT_2(t_1, t_2) = RT_2(t,t')$.

In Figure 10, we declare a dependent datatype TY for representing the types in the source and target object languages, and also modify the previously declared dependent datatype *EXP* for representing expressions in the source object language. The need for this modification can be clearly understood in the implementation of *cps* and *cpsw* in the Figure 11, where the functions *rt1OfTy* and *rt2OfTy* are assigned the following types:

$$\begin{array}{rcl} rt1OfTy & : & \Pi t : sty.TY(t) \rightarrow \Sigma t' : ty.RT_1(t,t') \\ rt2OfTy & : & \Pi t : sty.TY(t) \rightarrow \Sigma t' : ty.RT_2(t,t') \end{array}$$

indicating that *rt1OfTy* (*rt2OfTy*) returns a proof term of type $RT_1(t,t')$ ($RT_2(t,t')$) for some type index t' when applied to a value of type $TY(t)$. The entire program in Figure 11 can be readily verified in a prototype implementation of DML [20].

6. RELATED WORK AND CONCLUSION

In this paper, we propose an approach of implementing program transformations that makes use of a first-order typeful program representation formed in DML, where the type of an object program as well as the types of the free variables in the object program can be reflected in the type of the representation of the object program. We also develop some programming techniques to facilitate the use of this typeful representation in implementing program transformations.

The idea of employing dependent types in forming typeful program representation is not new. For instance, with Elf [17], a theorem prover/logic programming language based on the type theory underlying LF [8], we can readily form a typeful representation for the simply typed λ-calculus or even the second-order polymorphic λ-calculus and then establish properties such as type preservation. Also, one can find in [3] a typeful program representation used in implementing an interpreter in Cayenne [2], a functional programming language based on Haskell that supports a dependent type system, where the type assigned to (the implementation of) the interpreter guarantees it preserves types. However, none of these techniques support, in a function programming language (not a theorem prover), typeful program representation for potentially open programs, that is, programs containing free variables. In [13], a system $\lambda_{H\bigcirc}$ is proposed, with which a typeful program representation can be formed to implement a tagless interpreter for simply typed λ-calculus. However, given the complexity involved in $\lambda_{H\bigcirc}$, it needs to be further investigated as to whether $\lambda_{H\bigcirc}$ or a type system similar to it can be effectively used in practical programming.

In [5], an approach to constructing a higher-order typeful program representation is presented that makes use of the notion of *phantom types* available in Haskell. With this approach, it is shown that an implementation of the normalizing function for the simply typed λ-calculus preserves types and always yields $\beta\eta$-long normal form. However, the use of this approach in implementing program transformations is greatly limited as it does not allow the representation of a program to be used as a function argument.

In contrast to the program representation we use in this paper, where program variables are replaced with deBruijn indexes, a higher-order program representation is presented in [12] to support meta-programming, where a notion of names is introduced for handling free program variables. It is yet to see whether this representation also allows a CPS transform to be implemented in a typeful manner.

The way we implement substitutions in this paper is of great similarity to the way in which substitutions are handled in λ_σ-calculus [1]. Both employ deBruijn indexes to obviate the need for explicit names, and the several functions on substitutions that we implement can readily find correspondence in λ_σ. For instance, *subPre* and *subComp* directly corresponds the operators $\cdot$ and $\circ$ in λ_σ, respectively. At this moment, we are looking into the possibility of a typeful implementation of explicit substitutions.

In summary, we have presented an approach to implementing program transformations that makes use of a first-order typeful program representation formed in Dependent ML (DML). We have both developed some techniques for programming with such a program representation and then implemented a CPS transform for the simply typed λ-calculus extended with *callcc* and *throw* such that the type assigned to the implementation can capture the relation between the type of an expression and that of its CPS transform.

When constructing a compiler for a typed functional programming language, we may need to apply CPS transformation and/or closure conversion. As is argued in [18, 11], there are some significant benefits when the compiler makes

```
fun rt1OfTy t = RT1 (rt2OfTy t)
withtype {t:ty}. TY(t) -> [t':ty] RT1 (t, t')

and rt2OfTy TYint = RT2int
  | rt2OfTy (TYfun (pf1, pf2)) = RTfun (rt2OfTy pf1, rt1OfTy pf2)
  | rt2OfTy (TYcont pf) = RTcont (rt2OfTy pf)
  | rt2OfTy (TYbot) = RTbot
withtype {t:ty}. TY(t) -> [t':ty] RT2 (t, t')

fun cps (RT1 pf) vm e  = EXPlam' (cpsw pf (vmShi vm) EXPone' e)
withtype
  {G1:env,G2:env,t1:ty,t2:ty}. RT1 (t1, t2) -> VM (G1, G2) -> EXP (G1, t1) -> EXP' (G2, t2)

and cpsw pf vm k e =
  case e of
      EXPvar (v) => EXPapp' (k, EXPvar' (vm pf v))
    | EXPlam (e) =>
      let
         val RT2fun (pf1, pf2) = pf
      in
         EXPapp' (k, EXPlam' (cps pf2 (vmLam pf1 vm) e))
      end
    | EXPapp (t1, e1, e2) =>
      let
         val pf1 = rt2OfTy t1
         val k = EXPlam' (EXPapp' (EXPapp' (EXPtwo', EXPone'), expShi (expShi k)))
         val k = EXPlam' (cpsw pf1 (vmShi vm) k e2)
      in
         cpsw (RT2fun (pf1, RT1(pf))) vm k e1
      end
   | EXPcal e =>
      let
         val k = expShi k
         val k = EXPlam' (EXPapp' (EXPapp' (EXPone', k), k))
      in
         cpsw (RT2fun (RT2cont pf, RT1 pf)) vm k e
      end
   | EXPthr (t1, e1, e2) =>
      let
         val pf1 = rt2OfTy t1
         val k = EXPlam' (cpsw pf1 (vmShi vm) EXPone' e2)
      in
         cpsw (RT2cont pf1) vm k e1
      end
withtype
  {G1:env, G2:env, t1:ty, t2:ty}
    RT2 (t1, t2) -> VM (G1, G2) -> EXP' (G2, t2 -> bot) -> EXP (G1, t1) -> EXP' (G2, bot)
```

Figure 11: Implementing a CPS transform function again

use of typed intermediate languages. Suppose a typeless representation is chosen for some typed intermediate language acting as the target language of CPS transformation. Then it is necessary to verify individually whether a program in the intermediate language is well-typed after it is generated by a CPS transform function. With a typeful representation, it becomes possible to implement a typeful CPS transform function such that its type can guarantee that every program it generates is well-typed.

Along the line of research on typeful program transformations, we plan to handle more program language features (e.g., polymorphism and pattern matching) and more program transformations (e.g., closure conversion) in future, facilitating the use of typeful program representation in constructing compilers for typed programming languages.

7. REFERENCES

[1] M. Abadi, L. Cardelli, Curien, P.-L., and Lévy, J.-J. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] L. Augustsson. Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Baltimore, 1998.

[3] L. Augustsson and M. Carlsson. An excercise in dependent types: A well-typed interpreter, 1999. Available as `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`

[4] A. I. Baars and S. D. Swierstra. Typing Dynamic Typing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, Pittsburgh, PA, October 2002.

[5] O. Danvy and M. Rhiger. A Simple Take on Typed Abstract Syntax in Haskell-like Languages. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS '01)*, pages 343–358, Tokyo, Japan, March 2001.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes mathematicae*, 34:381–392, 1972.

[7] T. Griffin. A Formulae-as-Types Notion of Control. In *Conference Record of POPL '90: 17th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, CA, January 1990.

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

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

[10] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[11] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[12] A. Nanevski. Meta-Programming with Names and Necessity. In *Proceedings of the International Conference on Functional Programming (ICFP '02)*, pages 206–217, Pittsburgh, PA, September 2002.

[13] E. Pasalic and T. S. Walid Taha. Tagless Staged Interpreters for Typed Langauges. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, pages 281–229, Pittsburgh, PA, October 2002.

[14] S. Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`, Feb. 1999.

[15] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[16] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[17] C. Schürmann and F. Pfenning. Automated Theorem Proving in a Simple Meta-Logic for LF. In *Proceedings of the 15th International Conference on Automated Deduction (CADE)*, pages 286–300. Springer-Verlag LNCS 1421, 1998.

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

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

[20] H. Xi. Dependent ML. Available at `http://www.cs.bu.edu/~hwxi/DML/DML.html`, 2001.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.