Implementing Typeful Program Transformations[*][†]

Chiyan Chen
Computer Science Department
Boston University
Boston, MA 02215, USA
chiyan@cs.bu.edu

Rui Shi
Computer Science Department
Boston University
Boston, MA 02215, USA
shearer@cs.bu.edu

Hongwei Xi
Computer Science Department
Boston University
Boston, MA 02215, USA
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

Address for correspondence: Computer Science Department, Boston University, Boston, MA 02215, USA
[*]Partially supported by the NSF Grants No. CCR-0224244 and No. CCR-0229480
[†]A previous version of the paper [6] appears in the Proceedings of ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics Based Program Manipulation (PEPM '03)

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, 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.

Keywords: Typeful Program Transformation, Dependent Types, Applied Type System, ATS, Continuation Passing Style, CPS, Closure Conversion

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) [15] or Haskell [22], we often choose to declare a datatype for representing object programs. For instance, we may declare the following datatype $\mathbf{EXP}$ 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):

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

can be represented as follows:

$$EXPlam("x", EXPlam("y", EXPapp(EXPvar\ "y",\ EXPvar\ "x")))$$

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 $\mathbf{EXP}$. Also, there are more values of type $\mathbf{EXP}$ than simply typed λ-expressions. As an example, the following value:

$$EXPlam("x",\ EXPapp(EXPvar\ "x", EXPvar\ "x"))$$

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 free variables. These problems

```
fun nf0 (e: EXP): EXP =
  case e of
    | EXPvar _ => e
    | EXPlam (x, e) => EXPlam (x, nf0 e)
    | EXPapp (e1, e2) =>
        (case nf0 e1 of
           | EXPlam (x, e) => nf0 (subst0 e2 x e)
           | e1 => EXPapp (e1, nf0 e2))
```

Figure 1. A typeless implementation of normalization

arise when we try to implement a function nf_0 whose intended use is to reduce a simply typed λ-expression into normal form (NF). Clearly, nf_0 can be regarded as a form of program transformation. In Figure 1, we give a possible implementation of nf_0 in ATS, a programming language with its type system rooted in the framework *Applied Type System* ($\mathcal{ATS}$) [30]. In particular, ATS subsumes the previously designed programming language *Dependent ML* (DML) [33, 28], which extends ML with a restricted form of dependent types. However, we emphasize that this paper makes no direct use of the framework $\mathcal{ATS}$. In particular, the underlying type theory for the paper is completely within DML and a moderate amount of knowledge of DML is sufficient for understanding the typeful programming techniques we are to present, which are largely in line with the typeful programming paradigm advocated by Cardelli [5].

The syntax involved in ATS should be readily accessible to those who are familiar with Standard ML [15]. We use $subst_0$ for a function that implements the usual substitution on λ-terms. The implementation of $subst_0$, which is omitted here, is not completely trivial as we need to prevent free variables from being captured and thus some form of α-renaming needs to be implemented as well.

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 de Bruijn indexes [4]. 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 ATS 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.) [23] 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 interesting 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 call-by-value CPS transform function, where the type assigned to the implementation captures the relation between the type of an object program and that of its CPS 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 de Bruijn indexes. We then show in Section 3 how substitution can be implemented in a typeful manner with this program representation. Also, we give in Section 4 a typeful implementation of the call-by-value evaluation for simply typed λ-calculus, where a typeful representation for closures is employed.

In Section 5, we present a typeful implementation of a call-by-value CPS transform function, and in Section 6, we briefly mention a typeful implementation of closure conversion. We also show how the second-order polymorphism can be handled in Section 7. 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 simply typed λ-calculus (with Curry typing) can be readily given as follows, where we use x for variables and b for some base types.

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

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.

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

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

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

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 are 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. We first declare two sorts ty and env in Figure 2. The concrete syntax indicates that

- $\twoheadrightarrow$ and :: are infix operators with right associativity, and

```
infixr ->> ::

datasort ty = int | ->> of (ty, ty)
datasort env = nil | :: of (ty, env)

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. Types for representing typing derivations of the simply typed λ-expressions

- a type index t is of sort ty if it is int or $t_1 \twoheadrightarrow t_2$ for some type indexes t_1 and t_2 of sort ty, and
- a type index G is of sort env if it is nil or $t :: G$ for some type indexes t and G 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, $int :: (int \twoheadrightarrow int) :: nil$ represents a context in which the first and the second variables are assigned the types int and $int \to int$, 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 next declare in Figure 2 two dependent datatypes for representing typing derivations of simply typed λ-expressions, and we provide some explanation as follows. We use curly braces to denote universal quantification. The type constructor $\mathbf{VAR}$ takes a type index G of sort env and a type index t of sort ty to form a type $\mathbf{VAR}(G, t)$ 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 $\mathbf{VAR}$ are assigned the following types, respectively:

$$\forall G : env. \forall t : ty. \mathbf{VAR}(t :: G, t)$$
$$\forall G : env. \forall t_1 : ty. \forall t_2 : ty. \mathbf{VAR}(G, t_1) \to \mathbf{VAR}(t_2 :: G, t_1)$$

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 de Bruijn indexes for variables: *VARone* stands for the de Bruijn index 1, referring to the first variable in a context, and *VARshi* stands for the shift operator that increases a given de Bruijn index by 1. For example, *VARshi*(*VARshi*(*VARone*)) represents the de Bruijn index 3, referring to the third variable in a context.

Like $\mathbf{VAR}$, the type constructor $\mathbf{EXP}$ also takes a type index G of sort env and a type index t of sort ty to form a type $\mathbf{EXP}(G, t)$ 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 $\mathbf{EXP}$, which are assigned the following types respectively:

$$\begin{array}{rcl} \mathit{EXPvar} & : & \forall G : env.\forall t : ty.\mathbf{VAR}(G,t) \rightarrow \mathbf{EXP}(G,t) \\ \mathit{EXPlam} & : & \forall G : env.\forall t_1 : ty.\forall t_2 : ty.\mathbf{EXP}(t_1 :: G, t_2) \rightarrow \mathbf{EXP}(G, t_1 \twoheadrightarrow t_2) \\ \mathit{EXPapp} & : & \forall G : env.\forall t_1 : ty.\forall t_2 : ty.(\mathbf{EXP}(G, t_1 \twoheadrightarrow t_2), \mathbf{EXP}(G, t_1)) \rightarrow \mathbf{EXP}(G, t_2) \end{array}$$

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,

$$\mathit{EXPlam}(\mathit{EXPlam}(\mathit{EXPapp}(\mathit{EXPvar}(\mathit{VARone}), \mathit{EXPvar}(\mathit{VARshi}(\mathit{VARone})))))$$

where the representation can be assigned the following type:

$$\forall G : env.\mathbf{EXP}(G, int \twoheadrightarrow (int \twoheadrightarrow int) \twoheadrightarrow int).$$

Note that explicit applications to type indexes are omitted here. 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. In particular, it can be readily observed that there is a natural 1-to-1 correspondence between a typing derivation $\mathcal{D}$ of $\Gamma \vdash M : \tau$ and a value v of type $\mathbf{EXP}(G,t)$, where G and t are the representations of Γ and τ, respectively. For brevity, we omit a straightforward formalization of this observation. We are now ready to present some techniques for implementing program transformations with this typeful representation.

3. Implementing Substitution and Normalization

Substitution plays a key rôle in implementing program transformations. In this section, we present a typeful implementation of substitution for simply typed λ-calculus and then use it to implement normalization. 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 now write $\mathbf{SUB}(G_1, G_2)$ to represent the following type:

$$\forall t : ty.\mathbf{VAR}(G_1, t) \rightarrow \mathbf{EXP}(G_2, t)$$

where G_1 and G_2 are type indexes of sort env. For instance, the functions $idSub$ and $shiSub$ defined below correspond to the standard substitutions id and $\uparrow$, respectively, in the framework of explicit substitution [1]:

```
fun idSub {G:env}: SUB (G, G) = lam x => EXPvar (x)
fun shiSub {G:env, t:ty}: SUB (G, t :: G) = lam x => EXPvar (VARshi x)
fun subst {G1:env,G2:env,t:ty}
   (sub: SUB(G1,G2)) (e: EXP(G1,t)): EXP(G2,t) =
  case e of
    | EXPvar x => sub (x)
    | EXPlam e => EXPlam (subst (subLam sub) e)
    | EXPapp (e1, e2) => EXPapp (subst sub e1, subst sub e2)

and subLam {G1:env,G2:env,t:ty} (sub: SUB (G1,G2)): SUB (t::G1,t::G2) =
  lam v =>
     case v of
       | VARone => EXPvar (VARone)
         // the special syntax {...} is designed for type-checking; it
         // means that (implicit) static application needs to be performed
       | VARshi x' => subst (shiSub {...}) (sub x')
```

Figure 3. Implementing substitution

Note that the types assigned to id and $\uparrow$ are:

$$\forall G : \mathit{env}.\mathbf{SUB}(G, G) \text{ and } \forall G : \mathit{env}.\forall t : \mathit{ty}.\mathbf{SUB}(G, t :: G)$$

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 de Bruijn's notation [4], whose syntax is given as follows.

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

For instance, the de Bruijn'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:

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

Given an expression N and a substitution σ, we write $N \cdot \sigma$ for the substitution σ' such that $\sigma'(1) = N$ and $\sigma'(n + 1) = \sigma(n)$ for each index n. Also, given two substitutions σ_1 and σ_2, we write $\sigma_1 \circ \sigma_2$ for the the composition σ_3 of σ_1 and σ_2 such that $\sigma_3(n) = \sigma_1(n)[\sigma_2]$ for each index n.

For the functions subst and subLam in Figure 3, we have that $\mathit{subst}(\mathit{sub})(e)$ and $\mathit{subLam}(\mathit{sub})$ 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

```
fun subPre {G1:env,G2:env,t:ty} (sub: SUB(G1, G2)) (e: EXP (G2, t))
  : SUB (t :: G1, G2) =
  lam x => case x of VARone => e | VARshi x => sub x

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

Figure 4. Two functions on substitutions

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

Figure 5. A typeful implementation of normalization

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. The types of the functions $subst$, $subLam$, $subPre$ and $subComp$ can be formally written as follows:

$$\begin{array}{rcl}
subst & : & \forall G_1 : env.\forall G_2 : env.\forall t : ty.\mathbf{SUB}(G_1, G_2) \rightarrow \mathbf{EXP}(G_1, t) \rightarrow \mathbf{EXP}(G_2, t) \\
subLam & : & \forall G_1 : env.\forall G_2 : env.\forall t : ty.\mathbf{SUB}(G_1, G_2) \rightarrow \mathbf{SUB}(t :: G_1, t :: G_2) \\
subPre & : & \forall G_1 : env.\forall G_2 : env.\forall t : ty.\mathbf{SUB}(G_1, G_2) \rightarrow \mathbf{EXP}(G_2, t) \rightarrow \mathbf{SUB}(t :: G_1, G_2) \\
subComp & : & \forall G_1 : env.\forall G_2 : env.\forall G_3 : env.\mathbf{SUB}(G_1, G_2) \rightarrow \mathbf{SUB}(G_2, G_3) \rightarrow \mathbf{SUB}(G_1, G_3)
\end{array}$$

As an interesting example, we implement a function nf in Figure 5 to compute the normal form of a given simply typed λ-expression. The following type is assigned to the function nf,

$$\forall G : env.\forall t : ty.\mathbf{EXP}(G, t) \rightarrow \mathbf{EXP}(G, t)$$

which indicates that nf is type-preserving.

4. Implementing Evaluation

In this section, we present a typeful implementation of the call-by-value evaluation for simply typed λ-calculus. Instead of applying substitution to λ-expressions directly, which is prohibitively inefficient in practice, we form closures during evaluation.

```
datatype VAL (ty) =
  | {G:env,t1:ty,t2:ty} VALclo (t1 ->> t2) of (ENV G, EXP (t1 :: G, t2))

and ENV (env) =
  | ENVnil (nil)
  | {G:env,t:ty} ENVcons (t :: G) of (VAL t, ENV G)

fun evalVar {G:env,t:ty} (env: ENV G) (x: VAR (G, t)): VAL t =
  case x of
    | VARone => let val ENVcons (v, _) = env in v end
    | VARshi x => let val ENVcons (_, env) = env in evalVar env x end

fun eval {G:env,t:ty} (env: ENV G) (e: EXP (G, t)): VAL t =
  case e of
    | EXPvar x => evalVar env x
    | EXPlam (e) => VALclo (env, e)
    | EXPapp (e1, e2) =>
      let
         val VALclo (env', body) = eval env e1
         val v = eval env e2
      in
         eval (ENVcons (v, env')) body
      end

fun evaluate {t:ty} (e: EXP (nil, t)): VAL t = eval ENVnil e
```

Figure 6. Implementing call-by-value evaluation for simply typed λ-calculus

In Figure 6, we first declare two datatype constructors $\mathbf{VAL}$ and $\mathbf{ENV}$. Given indexes t and G of sorts ty and env, respectively, $\mathbf{VAL}(t)$ is the type for values that represent object values and $\mathbf{ENV}(G)$ is the type for values that represent environments, that is, lists of object values. In this case, the only form of an object value is a closure, which consists of an environment paired with the body of a λ-abstraction.

Given an environment and a de Bruijn index, the function $evalVar$ fetches a value from the environment that corresponds to the index. The main function is $eval$ which evaluates a λ-expression under a given environment. Note that the following type is assigned to $eval$:

$$\forall G : env.\forall t : ty.\mathbf{ENV}(G) \to \mathbf{EXP}(G, t) \to \mathbf{VAL}(t)$$

which captures the fact that $eval$ is type-preserving. Lastly, we implement $evaluate$ as a specialized version of $eval$. The type $\forall t : ty.\mathbf{EXP}(nil, t) \to \mathbf{VAL}(t)$ is assigned to $evaluate$, indicating that $evaluate$ can only be applied to a closed λ-expression.

In the body of $evalVar$, we have the following two cases of pattern matching:

`val ENVcons (v, _) = env` and `val ENVcons (_, env) = env` both of which are verified in ATS to be exhaustive. We provide some explanation as follows. Note that the variables env and x (in the definition of $evalVar$) are of types $\mathbf{ENV}(G)$ and $\mathbf{VAR}(G,t)$, respectively. Since $VARone$ is given the type $\forall G : env.\forall t : ty. VAR(t :: G, t)$, we can assume that $G = t' :: G'$ and $t = t'$ for some static variables G' and t' when x matches the pattern $VARone$. In order for env to match the pattern *ENVnil*, which is of type $\mathbf{ENV}(nil)$, we need $G = nil$. Thus, ATS can determine that env cannot match *ENVnil* if x matches $VARone$, i.e., the case of pattern matching `val ENVcons (v, _) = env` in the body of $evalVar$ is exhaustive. The reason for the exhaustiveness of the other case of pattern matching `val ENVcons (_, env) = env` in the body of $evalVar$ is similar. A formal detailed treatment of this issue is also available [29].

5. Implementing CPS Transformation

In this section, we present a typeful implementation of a call-by-value CPS transformation for a source object language that extends the simply typed λ-calculus with two control constructs $callcc$ and $throw$. It should be clear that a typeful implementation of a call-by-name CPS transformation can be done along the same line.

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 $\bot$.

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

We also associate more value constructors with the type constructor $\mathbf{EXP}$:

```
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 7, where k ranges over λ-expressions and $FV(M)$ is the set of free variables in M. The functions $T_1(\cdot)$ and $T_2(\cdot)$ are defined mutually inductively on

$$
\begin{array}{rcll}
T_1(\tau) & = & (T_2(\tau) \to \bot) \to \bot & \\
T_2(b) & = & b & \\
T_2(\tau_1 \to \tau_2) & = & T_2(\tau_1) \to T_1(\tau_2) & \\
T_2(cont(\tau)) & = & T_2(\tau) \to \bot & \\
cps(M) & = & \lambda x.cpsw(x, M) & x \notin FV(M) \\
cpsw(k, x) & = & k(x) & \\
cpsw(k, \lambda x.M) & = & k(\lambda x.cps(M)) & \\
cpsw(k, M_1(M_2)) & = & cpsw(\lambda x_1.cpsw(\lambda x_2.x_1(x_2)(k), M_2), M_1) & \\
 & & \multicolumn{2}{r}{x_1 \notin FV(k) \cup FV(M_2) \text{ and } x_2 \notin FV(k))} \\
cpsw(k, callcc(M)) & = & cpsw(\lambda x.x(k)(k), M) & x \notin FV(k) \\
cpsw(k, throw(M_1, M_2)) & = & cpsw(\lambda x.cpsw(x, M_2), M_1) & x \notin FV(k) \cup FV(M_2)
\end{array}
$$

Figure 7. A call-by-value CPS transform

the structure of types, while the functions $cps(\cdot)$ and $cpsw(\cdot, \cdot)$ are defined mutually inductively on the structure of λ-expressions. Note that $cps(\cdot)$ is a CPS transform function [26]. 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 [14, 11].

Theorem 5.1. 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. $\Gamma' \vdash cpsw(k, M) : \bot$ is also derivable if $\Gamma' \vdash k : T_2(\tau) \to \bot$ is derivable.

Proof:
We can establish (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 function cps in ATS such that the type assigned to the implementation captures Theorem 5.1.

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

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

This, for instance, is also the issue faced by Minamide and Okuma [17]. To relate two variables represented as de Bruijn indexes, we introduce a concept called *variable mapping*, which is closely related to

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

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

the notion of substitution. We use ν for a variable mapping, which maps de Bruijn indexes to de Bruijn indexes. We can now define cps and $cpsw$ in Figure 8 for λ-expressions in de Bruijn's notation, where k ranges over λ-expressions in de Bruijn's notation. This time, cps and $cpsw$ take a variable mapping ν as an extra argument that is needed to relate de Bruijn indexes in the source object program to those in the target object program.

We declare a type definition $\mathbf{VM}$ as follows for representing variable mappings, where the function T_2 is defined in Figure 7 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))
```

We now implement some functions as follows for manipulating variable mappings.

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

fun vmLam {G1:env,G2:env,t:ty} (vm: VM (G1, G2)): VM (t::G1, T2(t)::G2) =
  lam x => (case x of VARone => VARone | VARshi x => VARshi (vm x))
```

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

We also declare a datatype $\mathbf{EXP}'$ as follows for representing expressions in the target object language of the CPS transformation, which is just the simply typed λ-calculus. So $\mathbf{EXP}'$ is the same as $\mathbf{EXP}$ before $\mathbf{EXP}$ 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 9. The function $expShi$ is needed to increase the de Bruijn index of each free variable in a given expression by 1. It is straightforward to observe a direct correspondence between the implementation of cps and $cpsw$ in Figure 9 and their definition in Figure 8.

There is yet another significant point that we need to mention here. In order to type-check the code in Figure 9, the definition of the functions T_1 and T_2 needs to be "hard-wired" into the type-checker. While this is not difficult for us to do (in this case), such "hard-wiring" in general is difficult (if not

```
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 {G:env,t1:ty,t2:ty} (e: EXP' (G, t1)): EXP' (t2 :: G, t1) =
  subst shiSub e

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

and cpsw {G1:env,G2:env,t:ty}
   (vm: VM(G1,G2)) (k: EXP'(G2, T2(t) ->> bot)) (e: EXP(G1,t)): EXP'(G2,bot) =
  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
```

Figure 9. Implementing a call-by-value CPS transform function

```
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'))
  | RT2bot (bot, bot)
  | {t:ty,t':ty} RT2cont(cont(t), t' ->> bot) of RT2 (t, t')
```

Figure 10. An encoding of T_1 and T_2

completely impossible) to achieve as it is most likely that the programmer would have no access to the implementation of the type-checker. We have designed an approach in ATS that allows the programmer to provide proofs for constraints involving functions like T_1 and T_2.

The basic idea is to introduce two datatype constructors $\mathbf{RT}_1$ and $\mathbf{RT}_2$, which we declare in Figure 10, to encode the functions T_1 and T_2; given static terms t and t' of sort ty, the type $\mathbf{RT}_1(t, t')$ is inhabited if and only if $T_1(t) = t'$; similarly, the type $\mathbf{RT}_2(t, t')$ is inhabited if and only if $T_2(t) = t'$. The functions cps and $cpsw$ can now be defined (essentially) as follows:

```
fun cps {G1:env,G2:env,t1:ty,t2:ty}
   (pf: RT1 (t1, t2), vm: VM (G1, G2), e: EXP (G1, t1)): EXP' (G2, t2) =
  let prval RT1 (pf) = pf in EXPlam' (cpsw (pf, vmShi vm, EXPone', e)) end

and cpsw {G1:env,G2:env,t1:ty,t2:ty}
   (pf: RT2 (t1,t2), vm: VM (G1,G2), k: EXP' (G2,t2 ->> bot), e: EXP (G1,t1))
  : EXP' (G2, bot) = ...
```

where the extra argument pf of cps (resp. $cpsw$) guarantees that $T_1(t_1) = t_2$ (resp. $T_2(t_1) = t_2$) holds. In ATS, such arguments can be completely erased after type-checking and thus there is no run-time code involved. This is a programming paradigm which we refer to as *programming with theorem proving*. The interested reader may readily find details on this programming paradigm elsewhere [7]. Also, the code of a typeful implementation of CPS that employs programming with theorem proving is available on-line [31].

6. Implementing Closure Conversion

In the phase of CPS transformation, the compiler names all intermediate computations and eliminates the need for a control stack. Therefore, all unconditional control transfers, including function invocation and return, are achieved through function calls. The phase of closure conversion is another critical program transformation which achieves a separation between data and code. While most accounts consider closure conversion as a transformation applied to untyped terms, the one by Minamide et al [16] takes a type-theoretic point of view. We have also succeeded in implementing a typeful closure conversion for simply typed λ-calculus based on the one by Minamide et al. Given that the implementation strategy we use for handling typeful closure conversion is essentially the same as the one for handling CPS transformation, we omit further details. The interested reader can find the code for typeful closure conversion on-line [34].

7. Polymorphism

So far we have only dealt with program transformations on simply typed programs. In this section, we briefly demonstrate that the typeful programming techniques presented above are also applicable to polymorphically typed programs.

Let us first extend the syntax of simply typed λ-calculus as follows:

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \ldots \mid \alpha \mid \forall\alpha.\tau \\ \text{expressions} & M & ::= & \ldots \mid \forall^+(M) \mid \forall^-(M) \end{array}$$

where we use α for type variables. In addition, we have the following typing rules:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \forall^+(M) : \forall a.\tau}\ \textbf{(ty-tlam)}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash \forall^-(M) : \tau[a \mapsto \tau_0]}\ \textbf{(ty-tapp)}$$

The side condition associated with the typing rule **(ty-tlam)** is obvious: α cannot have free occurrences in Γ. We here essentially use $\forall^+$ and $\forall^-$ as markers to indicate the application of the typing rules **(ty-tlam)** and **(ty-tapp)**, respectively.

In Figure 11, we sketch a typeful implementation of the call-by-value evaluation for the second-order polymorphically typed λ-calculus. As the implementation extends the one given in Figure 6, we present only the additional code. We first declare a sort ty for type indexes that represent second-order polymorphic types. Note that this representation makes use of higher-order abstract syntax [23]. For instance, the polymorphic type $\forall\alpha_1.\forall\alpha_2.\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$ is formally represented as follows:

$$\mathit{forall}(\lambda a_1 : ty.\mathit{forall}(\lambda a_2 : ty.a_1 \twoheadrightarrow (a_2 \twoheadrightarrow a_1)))$$

As before, we declare a datatype constructor $\mathbf{EXP}$ in Figure 11 to represent polymorphic λ-expressions (or more precisely, the typing derivations of polymorphic λ-expressions), where the associated value constructors *EXPtlam* and *EXPtapp* correspond to the typing rules **(ty-tlam)** and **(ty-tapp)**, respectively. The types assigned to *EXPtlam* and *EXPtapp* can be formally written as follows:

$$\begin{array}{lcl} \mathit{EXPtlam} & : & \forall G : env.\forall f : ty \rightarrow ty.(\forall t : ty.\mathbf{EXP}(G, f(t))) \rightarrow \mathbf{EXP}(G, \mathit{forall}(f)) \\ \mathit{EXPtapp} & : & \forall G : env.\forall f : ty \rightarrow ty.\forall t : ty.\mathbf{EXP}(G, \mathit{forall}(f)) \rightarrow \mathbf{EXP}(G, f(t)) \end{array}$$

Also, we introduce an additional value constructor $VALtclo$, which is needed for constructing another form of closure that corresponds to λ-abstraction over a type variable (i.e., the marker $\forall^+$). We next extend the definition of $eval$ with two clauses for handling *EXPtlam* and *EXPtapp*.

We point out that the above technique for handling polymorphic programs can also be applied to other program transformations in the paper. In particular, a typeful implementation of CPS transformation for the second-order polymorphic λ-calculus, either call-by-value or call-by-name, can be readily done by following the one given in Figure 9.

```
datasort ty = ... | forall of (ty -> ty)

datatype EXP (env, ty) =
  | ...
  | {G:env, f:ty -> ty} EXPtlam (G, forall f) of ({t:ty} EXP (G, f t))
  | {G:env, f:ty -> ty, t:ty} EXPtapp (G, f t) of EXP (G, forall f)

datatype VAL (ty) =
  | ...
  | {G:env, f:ty -> ty} VALtclo (forall f) of (ENV G, {t: ty} EXP (G, f t))

fun eval {G:env,t:ty} (env: ENV G) (e: EXP (G, t)): VAL t =
  case e of
    | ...
    | EXPtlam (e) => VALtclo (env, e)
    | EXPtapp (e) =>
      let
          val VALtclo (env, body) = eval env e
      in // the special syntax {...} is designed for type-checking; it
         // means that (implicit) static application needs to be performed
          eval env (body {...})
      end
```

Figure 11. Implementing call-by-value evaluation for polymorphically typed λ-calculus

8. Related Work and Conclusion

In this paper, we propose an approach to implementing program transformations that makes use of a first-order typeful program representation, 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 [24], a theorem prover/logic programming language based on the type theory underlying LF [12], 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 a typeful program representation used in implementing an interpreter by Augustsson and Carlsson [3] 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 functional programming language (not a theorem prover), typeful program representation for potentially open programs, that is, programs containing free variables. A system $\lambda_{H\bigcirc}$ is proposed by Pasalic et all [21], 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 a recent study on meta-programming with typeful object-language representations [20], Pasalic and Linger (who refer to typeful representation as *typed representation*) employ a strategy to represent abstract syntax that is of great similarity to the one given in Figure 2. Also, they give an implementation of a tagless interpreter that is closely related to the typeful implementation of the call-by-value evaluation given in Section 4. Instead of using dependent types, they rely on generalized datatypes [9, 13], which are also called guarded recursive datatypes [32], to simulate dependent types in Haskell. Please see a paper by Xi et al [8] for a critique on the practicality of simulating dependent types in Haskell.

An approach to constructing a higher-order typeful program representation is presented by Danvy et al [10] that makes use of the notion of *phantom types* available in Haskell. With this approach, it is shown that an implementation of the normalization 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 de Bruijn indexes, a higher-order program representation is presented by Nanevski and Pfenning [19] 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 de Bruijn indexes to obviate the need for explicit names, and the several functions on substitutions that we implement can readily find counterparts in λ_σ. For instance, $subPre$ and $subComp$ directly corresponds the operators $\cdot$ and $\circ$ in λ_σ, respectively.

In summary, we have presented an approach to implementing program transformations that makes use of a first-order typeful program representation. Among several presented examples, a call-by-value CPS transform for the simply typed λ-calculus extended with $callcc$ and $throw$ is implemented in such a manner 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 the literature [27, 18], there are some significant benefits when the compiler makes 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., pattern matching) and more program transformations in the future, facilitating the use of typeful program representation in constructing compilers for typed programming languages. Also, given that the typeful program representation presented in the paper is based on abstract syntax with de Bruijn indexes, it is certainly interesting to study whether it is also possible to find a typeful program representation based on abstract syntax with named variables. Some work on nominal logic [25] seems to be of relevance in this direction.

References

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

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

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

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

[5] Cardelli, L.: Typeful Programming, in: *Formal Description of Programming Concepts* (E. J. Neuhold, M. Paul, Eds.), Springer-Verlag, Berlin, 1991, 431–507.

[6] Chen, C., Xi, H.: Implementing Typeful Program Transformations, *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation*, San Diego, CA, June 2003.

[7] Chen, C., Xi, H.: Combining Programming with Theorem Proving, November 2004, Available at: `http://www.cs.bu.edu/~hwxi/ATS/PAPER/CPwTP.ps`.

[8] Chen, C., Zhu, D., Xi, H.: Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell, *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Springer-Verlag LNCS vol. 3057, Dallas, TX, June 2004.

[9] Cheney, J., Hinze, R.: *Phantom Types*, Technical Report CUCIS-TR2003-1901, Cornell University, 2003, Available at
`http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901`.

[10] Danvy, O., Rhiger, M., Rose, K.: Normalization by Evaluation with Typed Abstract Syntax, *Journal of Functional Programming*, **11**(6), 2001, 673–680.

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

[12] Harper, R. W., Honsell, F., Plotkin, G. D.: A Framework for Defining Logics, *Journal of the ACM*, **40**(1), January 1993, 143–184.

[13] Hinze, R.: Fun with Phantom Types, in: *The Fun of Programming* (J. Gibbons, O. de Moor, Eds.), Palgrave Macmillan, 2003, ISBN 1-4039-0772-2 (hardback) 0-333-99285-7 (paperback), 245–262.

[14] Meyer, A., Wand, M.: Continuation Semantics in Typed Lambda Calculi (summary), *Logics of Programs* (R. Parikh, Ed.), Springer-Verlag LNCS 224, 1985.

[15] Milner, R., Tofte, M., Harper, R. W., MacQueen, D.: *The Definition of Standard ML (Revised)*, MIT Press, Cambridge, Massachusetts, 1997, ISBN 0-262-63181-4.

[16] Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion, *Proceedings of 23rd Annual ACM Symposium on Principles of Programming Languages (POPL '96)*, St. Petersburgh, Florida, 1996.

[17] Minamide, Y., Okuma, K.: Verifying CPS transformations in Isabelle/HOL, *Proceedings of Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN)*, 2003.

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

[19] Nanevski, A., Pfenning, F.: Meta-Programming with Names and Necessity, To appear in JFP. A previous version appeared in the *Proceedings of the International Conference on Functional Programming (ICFP 2002)*, pp. 206–217.

[20] Pasalic, E., Linger, N.: Meta-programming with Typed Object-Language Representations., *The 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)* (G. Karsai, E. Visser, Eds.), Springer-Verlag LNCS, 3286, Vancouver, BC, 2004.

[21] Pasalic, E., Walid Taha, T. S.: Tagless Staged Interpreters for Typed Languages, *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, Pittsburgh, PA, October 2002.

[22] Peyton Jones, S., et al.: Haskell 98 – A Non-strict, Purely Functional Language, Available at `http://www.haskell.org/onlinereport/`, February 1999.

[23] Pfenning, F., Elliott, C.: Higher-order abstract syntax, *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, Atlanta, Georgia, June 1988.

[24] Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)* (H. Ganzinger, Ed.), Springer-Verlag LNAI 1632, Trento, Italy, July 1999.

[25] Pitts, A. M.: Nominal Logic, A First Order Theory of Names and Binding, *Information and Computation*, **186**(2), November 2003, 165–193.

[26] Plotkin, G. D.: Call-by-Name, Call-by-Value and the λ-Calculus, *Theoretical Computer Science*, **1**(2), December 1975, 125–159.

[27] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: A Type-Directed Optimizing Compiler for ML, *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1996.

[28] Xi, H.: *Dependent Types in Practical Programming*, Ph.D. Thesis, Carnegie Mellon University, 1998, Pp. viii+189. Available at `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[29] Xi, H.: Dependently Typed Pattern Matching, *Journal of Universal Computer Science*, **9**(8), 2003, 851–872.

[30] Xi, H.: Applied Type System (extended abstract), *post-workshop Proceedings of TYPES 2003*, Springer-Verlag LNCS 3085, 2004.

[31] Xi, H.: Applied Type System, 2005, Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[32] Xi, H., Chen, C., Chen, G.: Guarded Recursive Datatype Constructors, *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM press, New Orleans, LA, January 2003.

[33] Xi, H., Pfenning, F.: Dependent Types in Practical Programming, *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM press, San Antonio, Texas, January 1999.

[34] Xi, H., Shi, R., Chen, C.: Typeful Closure Conversion, January 2005, Available at: `http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/cc.ats`.