Verifying CPS Transformations in Isabelle/HOL

Yasuhiko Minamide
University of Tsukuba
and
PRESTO, JST
minamide@is.tsukuba.ac.jp

Koji Okuma
University of Tsukuba
okuma@score.is.tsukuba.ac.jp

ABSTRACT

We have verified several versions of the CPS transformation in Isabelle/HOL. In our verification we adopted first-order abstract syntax with variable names so that the formalization is close to that of hand-written proofs and compilers. To simplify treatment of fresh variables introduced by the transformation, we introduced abstract syntax parameterized with the type of variables. We also found that the standard formalization of α-equivalence was cumbersome for theorem provers and reformulated α-equivalence as a syntax-directed deductive system. To simplify verification of the CPS transformation on the language extended with let-expressions, it was essential to impose that variables are uniquely used in a program.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers,optimization*; F.1.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*mechanical theorem proving*

General Terms

Languages, Verification

Keywords

Program transformation, theorem proving, correctness proofs

1. INTRODUCTION

Recently developed compilers apply various sophisticated program transformations to achieve high performance and to implement various advanced features of programming languages. As transformations become more and more sophisticated, their hand-written correctness proofs become less and less reliable. Thus, it is desirable to verify the correctness of program transformations with theorem provers.

We verified several versions of the CPS transformation in Isabelle/HOL. The CPS transformation has been used in compilers of functional programming languages to make explicit the control flow of a program and to implement continuations as first-class objects [17, 2]. The transformation translates programs into a special form called the continuation-passing style.

As a representation of lambda terms, we chose a first-order abstract syntax with variable names for the following reasons. It is simple and close to the representation of lambda terms in hand-written proofs. Even if variable names are used in abstract syntax, there is no need to rename them for the definition of reduction semantics, because we only consider the evaluation of closed programs and an expression inside a lambda abstraction is not evaluated. Furthermore, abstract syntax with variable names is very close to the intermediate languages used in compilers. Thus, verification based on abstract syntax with variable names may help the verification of actual compilers in future research.

We verified several versions of the CPS transformation, those of Plotkin [16], Danvy and Filinski [4], and Danvy and Nielsen [5]. Plotkin's transformation is simple, but introduces redundant redexes. The other two are optimized transformations and it is more difficult to show their correctness. We wrote our proof scripts in the proof language Isabelle/Isar [19]. By using Isabelle/Isar we could write human-readable structured proofs. The proof scripts of our verification can be obtained from `http://www.score.is.tsukuba.ac.jp/~minamide/cps/`. There are several difficulties in our verification related to variable names.

The first problem relates to fresh variables introduced by the transformation. All the CPS transformations introduce some fresh variables, which must be different from the variables occurring in an original term. This restriction must be maintained explicitly: lemmas require explicit restrictions about variables. However, this makes it difficult to prove them in Isabelle/HOL. To solve this problem, we introduced an abstract syntax parameterized with the type of variables. With this abstract syntax, it is possible to implicitly impose the restriction.

The second problem relates to the treatment of the α-equivalence of lambda terms. Our verification of Danvy and Nielsen's and Danvy and Filinski's transformations rely on the α-equivalence of lambda terms. However, we found that the standard formalization of α-equivalence was cumbersome for theorem provers. Thus, we reformulated α-equivalence as a syntax-directed deductive system.

The last problem is encountered in verification of Danvy and Nielsen's transformation on the language extended with let-expressions. For the verification of this transformation,

MERλIN'03, August 26, 2003, Uppsala, Sweden

we imposed that let-bound variables are uniquely used in a program. Without this restriction, formalization of this transformation requires explicit renaming of let-bound variables and that makes verification difficult to conduct.

This paper is organized as follows. In Section 2 we introduce the language we consider and review the CPS transformation. We also explain why we choose an abstract syntax with variable names. In Section 3 we present the verification of Plotkin's transformation and introduce a parameterized abstract syntax. In Section 4 we present the verification of Danvy and Nielsen's transformation. Our deductive system of α-equivalence is discussed in this section. We also extend our verification to the language with let-expressions. In Section 5 we outline our formalization of Danvy and Filinksi's transformation. Finally we review related work and present our conclusions.

2. LANGUAGE AND TRANSFORMATION

We describe the syntax and semantics of the language we will consider in this paper and review Plotkin's CPS transformation. We also clarify why we chose an abstract syntax with variable names instead of an abstract syntax based on de Bruijn indexes.

The syntax of the language we consider in this paper is defined as follows:

$$M ::= x \mid \lambda x.M \mid MM$$

We consider a call-by-value operational semantics for this language. The operational semantics of the language is defined based on the following β-reduction rule.

$$(\lambda x.M)V \rightarrow M[V/x]$$

where V is a value, either a variable or an abstraction. We consider only the evaluation of closed programs. Thus, for evaluation of programs, the substitution $M[V/x]$ is required only for closed V. By considering this restriction, the substitution $M[N/x]$ is defined without considering the renaming of bound variables as follows:

$$(\lambda x.M)[N/y] \quad = \quad \begin{cases} \lambda x.M & (\text{if } x = y) \\ \lambda x.(M[N/y]) & (\text{if } x \neq y) \end{cases}$$

This is well-defined without considering the α-equivalence of lambda terms, and thus the operational semantics of the language is also well-defined without the α-equivalence of lambda terms. This makes it possible to formalize the operational semantics based on abstract syntax with variable names directly without problems of renaming. This is the first reason why we chose abstract syntax with variable names.

CPS transformations have been used in compilers of functional programming languages to make explicit the control flow of a program and to implement continuation as first-class objects [17, 2]. The transformation translates programs into a special form called the continuation-passing style. The following is Plotkin's transformation [16]:

$$\begin{array}{rcl} [\![x]\!] & = & \lambda k.kx \\ [\![\lambda x.M]\!] & = & \lambda k.k(\lambda x.[\![M]\!]) \\ [\![MN]\!] & = & \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnk)) \end{array}$$

where k, m and n are fresh variables. The translated program is evaluated with the initial continuation $\lambda x.x$. Plotkin showed that M and $[\![M]\!](\lambda x.x)$ are evaluated to equivalent values.

The transformation is defined by taking advantage of variable names. If we adopt an abstract syntax based on de Bruijn indexes [6], we encounter difficulties in formalizing the transformation. Let us consider the following transformation of $x\,x$ by Plotkin's CPS transformation.

$$[\![x\,x]\!] = \lambda k.(\lambda k.kx)(\lambda m.(\lambda k.kx)(\lambda n.mnk)))$$

The transformation of x does not depend on the contexts where it appears. If we represent x by 0 of de Bruijn indexes, the transformation is revised as follows:

$$[\![0\ 0]\!] = \lambda(\lambda\ 0\ 2)(\lambda(\lambda\ 0\ 3)(\lambda\ 1\ 0\ 2))$$

where the transformation of 0 is context-sensitive: $(\lambda\ 0\ 2)$ or $(\lambda\ 0\ 3)$. As a result, the CPS transformation of terms represented by de Bruijn indexes is context-sensitive and seems to be more difficult to manage. This is another reason we chose an abstract syntax with variable names. We think that de Bruijn indexes are not suitable for the formalization of many program transformations because they are often defined by taking advantage of variable names.

2.1 Language and Semantics in Isabelle/HOL

We review Isabelle/HOL and show how we formalized the language and operational semantics based on abstract syntax with variable names. Isabelle is a generic interactive theorem prover that can be instantiated with several different object logics [13] and Isabelle/HOL is an instantiation of Isabelle to Church's higher-order logic [12]. We refer to Isabelle/HOL as Isabelle in the rest of this paper. The type system of Isabelle is similar to that of ML and has ML-style polymorphic types. Types follow the syntax of ML-types except that the function arrow is `=>`.

As discussed earlier, we chose first-order abstract syntax with variable names as the representation of lambda terms. The abstract syntax of lambda terms is defined by the `datatype` definition as follows:

```
datatype lt = Var nat
            | Abs nat lt
            | "$" lt lt (infixl 200)
```

where variable names are represented by natural numbers: `nat`. The keyword `infixl` means that the operator `$` can be used in infix notation with priority 200. For example, $\lambda x.\lambda y.xy$ is represented by the following term in Isabelle.

```
Abs 0 (Abs 1 (Var 0 $ Var 1))
```

This representation of terms is almost the same as those of intermediate languages of most compilers.

The observation in the previous section simplifies the formalization of substitution and reduction. Substitution is defined as a primitive recursive function in Isabelle as follows:

```
primrec
  (Var y)[V/x] = (if y = x then V else Var y)
  (Abs y M)[V/x] = (if (y = x) then Abs y M
                               else Abs y (M[V/x]))
  (M $ N)[V/x] = M[V/x] $ N[V/x]
```

Since the substitution `M[V/x]` is applied only to a closed term `V`, there is no need to rename variable `y` in the definition for abstraction. With substitution the reduction relation is defined as an inductively-defined relation.

```
consts  CPS :: lt => lt ("([_])" [250] 250)
primrec
 ([Var x]) = Abs 0 (Var 0 $ Var x)
 ([Abs x M]) = Abs 0 (Var 0 $ Abs x ([M]))
 ([M $ N])  = Abs 0 (([M]) $ (Abs 1 (([N]) $ (Abs 2 (Var 1 $ Var 2 $ Var 0)))))
```

Figure 1: Plotkin's CPS transformation: naive formalization

```
consts  CPS :: 'a lt => 'a variable lt ("([_])" [250] 250)
primrec
 ([Var x]) = Abs k (Var k $ Var (x~))
 ([Abs x M]) = Abs k (Var k $ (Abs (x~) ([M]) ))
 ([M $ N]) = Abs k (([M]) $ (Abs m (([N]) $ (Abs n (Var m $ Var n $ Var k)))))
```

Figure 2: Plotkin's CPS transformation: revised formalization

.

```
consts
  eval :: (lt * lt) set
inductive eval
  intros
  [isValue V; closed V] ⇒
                    ((Abs x M) $ V, M[V/x])∈eval
  [isValue V; (M,M')∈eval] ⇒ (V $ M, V $ M')∈eval
  (M,M')∈eval ⇒ (M $ N, M' $ N)∈eval
```

This definition introduces a relation `eval` between `lt`. To ensure that substitution is applied only if `V` is closed, the rule for the β-axiom is restricted to the case where `V` is closed.

3. PLOTKIN'S CPS TRANSFORMATION

We outline our verification of Plotkin's CPS transformation in this section. The key to our verification is to introduce abstract syntax parameterized with the type of variables. The parameterized abstract syntax is implemented as a polymorphic datatype.

Plotkin's transformation can be formalized as a primitive recursive function in Isabelle if we fix the representation of the variables k, m and n to some natural numbers. Figure 1 shows a formalization of the transformation where k, m and n are represented by the natural numbers 0, 1 and 2, respectively. The notation ([M]) is introduced for `(CPS M)` [1]. We first attempted to verify the correctness of the transformation based on this formalization, but found that the formalization was cumbersome for verification. The transformation is valid only if the variables represented by 0, 1 and 2 do not occur in a program. This condition must be maintained in every lemma, which makes it difficult to prove lemmas, especially with automated theorem-proving tactics.

This problem occurs because the distinction between the variables in an original term and those introduced by the transformation is not clear from the formalization. To overcome this problem we introduce an abstract syntax parameterized with the set of variables. The abstract syntax is implemented by the following polymorphic datatype definition.

```
datatype 'a lt = Var 'a
               | Abs 'a "'a lt"
               | $ "'a lt" "'a lt" (infixl 200)
```

[1]We use notation ([M]) instead of [M] because [M] conflicts with Isabelle's notation.

The type of terms `'a lt` is parameterized with the type `'a` of variables. The type `'a` can be any type: our definition of substitution does not require renaming and thus a type with a finite number of values can be used for variables.

In this representation, the source and target languages with different sets of variables can share the same abstract syntax. In the CPS transformation, we represent the set of variables of the target language by the following datatype.

```
datatype 'a variable = Orig 'a ("_~")| k | m | n
```

The variables introduced by the transformation are represented by constants: `k`, `m` and `n`. A variable `x` in the source language is translated into `x~`, that is, an abbreviation of `Orig x`.

Using this abstract syntax, the CPS transformation is defined as a polymorphic primitive recursive function. The transformation can then be obtained by refining the previous definition and is shown in Figure 2. By this formalization, freshness of the variables k, m and n is imposed implicitly and we do not have to maintain the restriction explicitly. Then, proofs of many lemmas are greatly simplified. For example, the following is the substitution lemma for the CPS transformation.

$$[\![M]\!][\Psi(V)/x] = [\![M[V/x]]\!]$$

where $\Psi(V)$ is defined as follows: $\Psi(x) = x$ and $\Psi(\lambda x.M) = \lambda x.[\![M]\!]$. This is one of the key lemmas in Plotkin's proof, and its proof there requires 23 lines. The following is the proof of the lemma in Isabelle. It is proved by induction on the structure of `M`, followed by the automatic theorem-proving tactics `auto`.

```
lemma [isValue V; closed V] ⟹
                          ([M[V/x]]) = ([M])[Ψ(V)/(x~)]
  by(induct M,auto)
```

If we adopt the unparameterized abstract syntax, the proof gets more complicated. The statement of the lemma must explicitly mention that `M` does not contain `0`, `1` and `2` as variables.

The main part of our verification is based on Plotkin's proof. Plotkin introduced the auxiliary transformation $M{:}K$, called the colon transformation, and showed the correctness of the CPS transformation by the following properties.

1. $[\![M]\!]K \rightarrow^* M{:}K$ (if K is a closed value).

2. If $M \to N$ then $M{:}K \to^* N{:}K$ (if K is a closed value).

Plotkin's proofs of these properties require 17 and 25 lines, respectively. Our proof scripts of these lemmas consist of 82 and 90 lines. It was not very difficult to translate Plotkin's proofs into the proofs in Isabelle.

4. DANVY AND NIELSEN'S CPS TRANSFORMATION

4.1 Formalization of the transformation

We verified an optimized version of the CPS transformation by Danvy and Nielsen [5]. This verification was much more difficult than our verification of Plotkin's CPS transformation. In addition to the problem we described in the previous section, we encountered several other problems in formalizing and proving the correctness of the transformation. We outline our formalization mostly without using the syntax of Isabelle. The concrete formalization in Isabelle can be obtained by translating the material in this section.

Danvy and Nielsen's CPS transformation avoids introducing redundant β-redexes. For comparison, let us consider the following Plotkin's transformation:

$$[\![x_1 x_2]\!](\lambda y.y) = (\lambda k.(\lambda k.k x_1)(\lambda m.(\lambda k.k x_2)(\lambda n.mnk)))(\lambda y.y)$$

where many redundant redexes are introduced by the transformation. Danvy and Nielsen's transformation is one of the CPS transformations that avoid introducing these redundant redexes. Their transformation is defined as the following mutually recursive transformations:

$$\begin{array}{rcl} \Psi(x) & = & x \\ \Psi(\lambda x.M) & = & \lambda x.\lambda k.[\![M]\!]k \end{array}$$

$$\begin{array}{rcl} [\![V]\!]K & = & K\Psi(V) \\ [\![V_0 V_1]\!]K & = & \Psi(V_0)\Psi(V_1)K \\ [\![V_0 M_1]\!]K & = & [\![M_1]\!](\lambda a_1.\Psi(V_0)a_1 K) \\ [\![M_0 V_1]\!]K & = & [\![M_0]\!](\lambda a_0.a_0\Psi(V_1)K) \\ [\![M_0 M_1]\!]K & = & [\![M_0]\!](\lambda a_0.[\![M_1]\!](\lambda a_1.a_0 a_1 K)) \end{array}$$

where $\Psi(V)$ and $[\![M]\!]$ are transformations of values and terms, respectively. A continuation K is given as an argument of the transformation $[\![M]\!]$. For example, $x_1 x_2$ with continuation $\lambda y.y$ is transformed into the following term:

$$[\![x_1 x_2]\!](\lambda y.y) = x_1 x_2 (\lambda y.y)$$

where no redundant redex is introduced.

The first problem with this transformation concerns fresh variables introduced in the transformation. For Danvy and Nielsen's CPS transformation, it is not enough to introduce a fixed number of fresh variables. In the specification, there are three variables introduced by the transformation: k, a_0 and a_1. However, many instances of the variable a_0 are required simultaneously. The following example clarifies the problem.

$$\begin{array}{l} [\![x_1 x_2 (x_3 x_4 x_5)]\!](\lambda y.y) = \\ \qquad x_1 x_2 (\lambda a_0.x_3 x_4 (\lambda a_0'.a_0' x_5 (\lambda a_1.a_0 a_1 (\lambda y.y)))) \end{array}$$

During the computation of $x_3 x_4$, the value of $x_1 x_2$ must be preserved. Thus, a_0 and a_0' must be different variables. This is not clear from the definition and we first thought that it was enough to introduce the three variables a_0, a_1 and k as in Plotkin's CPS transformation. However, in the specification, it is assumed that clashes of variable names are resolved by renaming bound variables. Since this renaming is not performed automatically, it is not possible to formalize the specification, above, directly in Isabelle.

To eliminate implicit renaming of the variable a_0, we take the approach of generating fresh variables explicitly. Most compilers obtain fresh variables in this approach. We revise the definition of the transformation so that the transformation $[\![\cdot]\!]_i$ is indexed by a natural number, which is used to generate fresh variables. The following is the definition of the revised transformation:

$$\begin{array}{rcl} [\![V]\!]_i K & = & K\Psi(V) \\ [\![V_0 V_1]\!]_i K & = & \Psi(V_0)\Psi(V_1)K \\ [\![V_0 M_1]\!]_i K & = & [\![M_1]\!]_i(\lambda a_i.\Psi(V_0)a_i K) \\ [\![M_0 V_1]\!]_i K & = & [\![M_0]\!]_i(\lambda a_i.a_i\Psi(V_1)K) \\ [\![M_0 M_1]\!]_i K & = & [\![M_0]\!]_i(\lambda a_i.[\![M_1]\!]_{i+1}(\lambda a_{i+1}.a_i a_{i+1} K)) \end{array}$$

where we assume there are fresh variables a_i indexed by a natural number i.

We first formalized Danvy and Nielsen's transformation based on this specification. As Plotkin's CPS transformation, the source and target languages are formalized by the same abstract syntax parameterized with the type of variables. We represented the variables after the translation with the following datatype:

```
datatype 'a variable = Orig 'a   ("_~")
                     | Avar nat  ("_~~")
                     | k
```

where `x~` and `i~~` represent the variable x in the source language and a_i introduced by the transformation, respectively. Then the specification can be formalized as functions with the following types:

```
consts
  Psi:: 'a lt => 'a lt'
  CPS:: ['a lt, nat, 'a lt'] => 'a lt'
```

where `'a lt'` is an abbreviation of `'a variable lt`.

The second problem concerns formalization of the transformation $[\![\cdot]\!]_i$ as a higher-order function: the transformation of a term is not a term, but a function from terms to terms. The automated theorem-proving tactics of Isabelle do not work effectively for this specification, so that even very simple lemmas cannot be proved automatically. Thus, it was not feasible to do verification based on this specification.

To solve this problem, we reformulated the transformation as a function from terms into contexts. To represent the transformation we introduced context $\mathbb{C}$ defined as follows:

$$\mathbb{C} ::= [\,] \mid \lambda a_i.\mathbb{C} \mid M\mathbb{C} \mid \mathbb{C}M$$

The following is the reformulation of the transformation: we write $(\!|M|\!)_i$ for the transformation based on contexts.

$$\begin{array}{rcl} (\!|V|\!)_i & = & [\,]\Psi(V) \\ (\!|V_0 V_1|\!)_i & = & \Psi(V_0)\Psi(V_1)[\,] \\ (\!|V_0 M_1|\!)_i & = & (\!|M_1|\!)_i \circ (\lambda a_i.\Psi(V_0)a_i[\,]) \\ (\!|M_0 V_1|\!)_i & = & (\!|M_0|\!)_i \circ (\lambda a_i.a_i\Psi(V_1)[\,]) \\ (\!|M_0 M_1|\!)_i & = & (\!|M_0|\!)_i \circ (\lambda a_i.[\,]) \circ (\!|M_1|\!)_{i+1} \circ (\lambda a_{i+1}.a_i a_{i+1}[\,]) \end{array}$$

where $\mathbb{C}_1 \circ \mathbb{C}_2$ is the composition of contexts $\mathbb{C}_1$ and $\mathbb{C}_2$. This reformulation is possible because K appears exactly once in $[\![M]\!]K$. We showed that two definitions are equivalent in Isabelle: $[\![M]\!]_i K = (\!|M|\!)_i[K]$ where the operation $\mathbb{C}[N]$ fills

the hole of $\mathbb{C}$ with N. Then we used the formalization based on contexts in our verification.

For the specification based on contexts, it was possible to show basic properties of the transformation with automated theorem-proving tactics. We first extended basic operations on terms such as the set of free variables and substitution to contexts. Since the definitions of the most operations are straightforward, we show only the definition of substitution on contexts:

$$\begin{array}{rcl} [][N/x] & = & [] \\ (\lambda a_i.\mathbb{C})[N/x] & = & \lambda a_i.\mathbb{C}[N/x] \\ (M\mathbb{C})[N/x] & = & M[N/x]\mathbb{C}[N/x] \\ (M\mathbb{C})[N/x] & = & \mathbb{C}[N/x]M[N/x] \end{array}$$

where the variable x is restricted to a variable of the source language. Then the following properties of the transformation are verified with automated theorem-proving tactics.

- $FV(M) = FV((\!|M|\!)_i)$
- $(\!|M[V/x]|\!)_i = (\!|M|\!)_i[\Psi(V)/x]$

The third problem in verification of the transformation concerns the α-equivalence of terms. Danvy and Nielsen proved the correctness of their CPS transformation by the following property.

$$M \to N \quad \Rightarrow \quad [\![M]\!]K \to^* [\![N]\!]K$$

However, this property does not hold in our formalization based on abstract syntax with variable names. In fact, $(\!|M|\!)_i[K]$ is not reduced to $(\!|N|\!)_i[K]$, but to a term that is α-equivalent to $(\!|N|\!)_i[K]$. Thus, we must show the following property.

$$\begin{array}{rcl} M \to N & \Rightarrow & (\!|M|\!)_i[K] \to^* N' \text{ and} \\ & & (\!|N|\!)_i[K] =_\alpha N' \text{ for some } N' \end{array}$$

Verification of this property requires formalization of α-equivalence.

4.2 Formalization of α-equivalence

We found that the standard formalization of α-equivalence was not natural for our formalization and cumbersome for automated theorem proving.

The operational semantics itself can be formalized without considering α-conversion and thus the set of variables can be finite. However, if the set of variables is finite, then the standard formalization of α-equivalence is not sufficient to derive the equivalence of terms. For example, if we have only two variables x and y, the following equivalence cannot be derived.

$$\lambda x.\lambda y.xy =_\alpha \lambda y.\lambda x.yx$$

Furthermore, it is rather difficult to derive α-equivalence of terms automatically in the standard formalization, because the standard formalization of α-equivalence contains the rule of transitivity and is not syntax-directed. Thus, it is not straightforward to determine whether two terms are α-equivalent automatically.

To overcome these issues, we formalized α-equivalence as a syntax-directed deductive system. Let Var be the set of variables. We say $E \subseteq \mathrm{Var} \times \mathrm{Var}$ is a renaming if $(x, y) \in E$ and $(x', y') \in E$ implies $x = x' \Leftrightarrow y = y'$. Any renaming can be extended to a permutation on the set of variables if the set of variables is finite. However, a renaming that cannot be extended to a bijection is also sometimes useful. The following relation is a renaming for the set of variables $\{x_i \mid i \in \mathbb{N}\}$.

$$\{(x_i, x_{i+1}) \mid i \in \mathbb{N}\}$$

The extension of a renaming $E \oplus (x, y)$ is defined as follows:

$$E \oplus (x, y) = \{(a, b) \mid (a, b) \in E \wedge a \neq x \wedge b \neq y\} \cup \{(x, y)\}$$

We define a deductive system with judgments of the form $E \vdash M =_\alpha N$, which means M and N are α-equivalent under the renaming E. The deductive system is defined as follows:

$$\frac{(x, y) \in E}{E \vdash x =_\alpha y} \qquad \frac{E \oplus (x, y) \vdash M =_\alpha M'}{E \vdash \lambda x.M =_\alpha \lambda y.M'}$$

$$\frac{E \vdash M =_\alpha M' \quad E \vdash N =_\alpha N'}{E \vdash MN =_\alpha M'N'}$$

We write $\vdash M =_\alpha N$ if $\mathrm{ID} \vdash M =_\alpha N$ where ID is the identity relation on the set of variables.

4.3 Alpha-equivalence in verification

Based on the deductive system of α-equivalence we verified Danvy and Nielsen's CPS transformation. In our verification, it was necessary to show the following α-equivalence:

$$\vdash (\!|M|\!)_i[K] =_\alpha (\!|M|\!)_{i+1}[K]$$

where $FV(K) \cap \{a_i \mid i \in \mathbb{N}\} = \emptyset$. Please note the set of variables in the target language of the transformation is the union of the set of variables of the source language $\mathrm{Var}_{\mathrm{src}}$ and the set of variables introduced by the transformation $\{a_i \mid i \in \mathbb{N}\} \cup \{k\}$.

In order to prove the property above, we first extended α-equivalence to contexts as follows:

$$E \vdash [] =_\alpha [] \qquad \frac{E \vdash \mathbb{C} =_\alpha \mathbb{C}' \quad (a_i, a_j) \in E}{E \vdash \lambda a_i.\mathbb{C} =_\alpha \lambda a_j.\mathbb{C}'}$$

$$\frac{E \vdash M =_\alpha M' \quad E \vdash \mathbb{C} =_\alpha \mathbb{C}'}{E \vdash M\mathbb{C} =_\alpha M'\mathbb{C}'}$$

$$\frac{E \vdash M =_\alpha M' \quad E \vdash \mathbb{C} =_\alpha \mathbb{C}'}{E \vdash \mathbb{C}M =_\alpha \mathbb{C}'M'}$$

This deductive system is designed so that the following property holds.

- If $E \vdash \mathbb{C} =_\alpha \mathbb{C}'$ and $E \vdash M =_\alpha M'$ then $E \vdash \mathbb{C}[M] =_\alpha \mathbb{C}'[M']$.

Then we proved the α-equivalence above in the following manner.

1. It is sufficient to show the following property:

   $$\mathrm{ID}' \cup \{(a_i, a_{i+1}) \mid i \in \mathbb{N}\} \vdash (\!|M|\!)_i[K] =_\alpha (\!|M|\!)_{i+1}[K]$$

   where ID' is $\{(x, x) \mid x \in \mathrm{Var}_{\mathrm{src}}\} \cup \{(k, k)\}$. It is because $FV((\!|M|\!)_i[K]) \cap \{a_i \mid i \in \mathbb{N}\} = \emptyset$.

2. We decompose the relation into the following relations with the property of α-equivalence on contexts.

   $$\mathrm{ID}' \cup \{(a_i, a_{i+1}) \mid i \in \mathbb{N}\} \vdash (\!|M|\!)_i =_\alpha (\!|M|\!)_{i+1}$$

   $$\mathrm{ID}' \cup \{(a_i, a_{i+1}) \mid i \in \mathbb{N}\} \vdash K =_\alpha K$$

$$x \in \mathrm{BVU} \qquad \frac{M \in \mathrm{BVU}}{\lambda x.M \in \mathrm{BVU}} \qquad \frac{M \in \mathrm{BVU} \quad N \in \mathrm{BVU} \quad FV(MN) \cap \mathrm{LBV}(MN) = \emptyset}{MN \in \mathrm{BVU}}$$

$$\frac{M \in \mathrm{BVU} \quad N \in \mathrm{BVU} \quad FV(\mathsf{let}\ x = M\ \mathsf{in}\ N) \cap \mathrm{LBV}(\mathsf{let}\ x = M\ \mathsf{in}\ N) = \emptyset}{\mathsf{let}\ x = M\ \mathsf{in}\ N \in \mathrm{BVU}}$$

Figure 3: Bound variable uniqueness

3. The first relation is proved by induction on structure of M. The second relation is obtained from $FV(K) \cap \{a_i \mid i \in \mathbb{N}\} = \emptyset$.

There is another important property of α-equivalence to show correctness of Danvy and Nielsen's transformation: α-equivalence and reduction commute.

- If $L =_\alpha M$ and $M \to N$ then $L \to N'$ and $N' =_\alpha N$ for some N'.

This does not hold for the λ-calculus formalized with variable names where reduction of open terms is considered [18]. In our formalization, α-equivalence and reduction commute because V in the reduction $(\lambda x.M)V \to M[V/x]$ is restricted to a closed term. The property is proved from the following substitution lemma on α-equivalence.

- If $E \oplus (x, y) \vdash M =_\alpha M'$ and $\emptyset \vdash V =_\alpha V'$ then $E \vdash M[V/x] =_\alpha M'[V'/y]$.

4.4 Verification for `let`-expressions

We extend our verification of Danvy and Nielsen's transformation to the language with let-expressions. This verification illustrates that it is essential to impose uniqueness of bound variables in a source term in order to simplify verification of some program transformations. We consider the following extended language.

$$M ::= x \mid \lambda x.M \mid MM \mid \mathsf{let}\ x = M\ \mathsf{in}\ M$$

Danvy and Nielsen's transformation on let-expressions is defined as follows:

$$\begin{array}{rcl} [\![\mathsf{let}\ x = V_0\ \mathsf{in}\ M_1]\!]K & = & \mathsf{let}\ x = \Psi(V_0)\ \mathsf{in}\ [\![M_1]\!]K \\ [\![\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1]\!]K & = & [\![M_0]\!](\lambda a_0.\mathsf{let}\ x = a_0\ \mathsf{in}\ [\![M_1]\!]K) \end{array}$$

For this transformation, a clash of variable names may happen if a variable name is not uniquely used in a source term. In the following example, the free variable x in the the source term is made bound by the transformation.

$$[\![(\mathsf{let}\ x = y\ \mathsf{in}\ x)x]\!](\lambda z.z) = \mathsf{let}\ x = y\ \mathsf{in}\ x(\lambda a_0.a_0x(\lambda z.z))$$

Danvy and Filinski solved this clash of variable names by introducing an explicit renaming into their version of the CPS transformation [4]. We first tried their approach, but introducing renaming into the transformation makes it very difficult to prove properties of the transformation.

This kind of variable name clashes are usually solved by assuming that all bound variables are distinct in compilers. This assumption makes implementation of the transformation much simpler. We basically took this approach, but adopted a weaker assumption on variable names because the property that all bound variables are distinct is not preserved by reduction. We write $\mathrm{LBV}(M)$ for the set of let-bound variables of M defined as follows:

$$\begin{array}{rcl} \mathrm{LBV}(V) & = & \emptyset \\ \mathrm{LBV}(M_0M_1) & = & \mathrm{LBV}(M_0) \cup \mathrm{LBV}(M_1) \\ \mathrm{LBV}(\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1) & = & \{x\} \cup \mathrm{LBV}(M_0) \cup \mathrm{LBV}(M_1) \end{array}$$

Note that let-bound variables inside lambda abstractions are not included in $\mathrm{LBV}(V)$. Based on this definition, we inductively defined the set of terms BVU where let-bound variables are uniquely used. The definition is shown in Figure 3. The following property of BVU ensures that no name clash occurs when a term in BVU is transformed.

- If $M \in \mathrm{BVU}$ then $\mathrm{LBV}(M) \cap FV(M) = \emptyset$.

It is also shown that BVU is closed under substitution.

- If $M, V \in \mathrm{BVU}$ and V is a closed value then $M[V/x] \in \mathrm{BVU}$.

This holds because the let-bound variables inside lambda abstraction are not included in $\mathrm{LBV}(V)$. From this property, it is shown that BVU is preserved by reduction: if $M \in \mathrm{BVU}$ and $M \to N$ then $N \in \mathrm{BVU}$.

For verification of the transformation we showed that the following two key lemmas by assuming that M is in BVU. The proofs cannot be automated as much as those for the language without let-expressions.

- If $M \in \mathrm{BVU}$ then $FV(M) = FV((\![M]\!)_i)$.
- If $M \in \mathrm{BVU}$ then $(\![M[V/x]]\!)_i = (\![M]\!)_i[\Psi(V)/x]$.

With these lemmas, the rest of the verification was almost the same as that of Danvy and Nielsen's transformation without let-expressions.

5. DANVY AND FILINSKI'S CPS TRANSFORMATION

We also verified Danvy and Filinski's CPS transformation. Danvy and Filinski's transformation is defined with the two-level lambda and more complicated than Danvy and Nielsen's transformation. However, we could verify the transformation with the same techniques used for Danvy and Nielsen's transformation. Danvy and Nielsen's transformation was derived from Danvy and Filinski's CPS transformation by unfolding and folding [5].

The transformation is formalized with the two-level lambda

$$
\begin{array}{rcl}
(\!|...|\!)_i & : & \text{syntax} \to \text{context} \times \text{syntax} \\
(\!|x|\!)_i & = & ([\,], x) \\
(\!|\lambda x.M|\!)_i & = & ([\,], \lambda x.\lambda k.(\!|M|\!)'[k]) \\
(\!|M_1 M_2|\!)_i & = & (\mathsf{fst}((\!|M_1|\!)_i) \circ \mathsf{fst}((\!|M_2|\!)_{i+1}) \circ \mathsf{snd}((\!|M_1|\!)_i)\mathsf{snd}((\!|M_2|\!)_{i+1})(\lambda a_i.[\,]), a_i)
\end{array}
$$

$$
\begin{array}{rcl}
(\!|...|\!)' & : & \text{syntax} \to \text{context} \\
(\!|x|\!)' & = & [\,]x \\
(\!|\lambda x.M|\!)' & = & [\,](\lambda x.\lambda k.(\!|M|\!)'[k]) \\
(\!|M_1 M_2|\!)' & = & \mathsf{fst}((\!|M_1|\!)_0) \circ \mathsf{fst}((\!|M_2|\!)_1) \circ \mathsf{snd}((\!|M_1|\!)_0)\mathsf{snd}((\!|M_2|\!)_1)[\,]
\end{array}
$$

Figure 4: Reformulation of Danvy and Filinski's CPS transformation with contexts

calculus as follows:

$$
\begin{array}{rcl}
[\![...]\!] & : & \text{syntax} \to (\text{syntax} \to \text{syntax}) \to \text{syntax} \\
[\![x]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa x \\
[\![\lambda x.M]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa(\lambda x.\lambda k.\overline{@}[\![M]\!]'k) \\
[\![M_1 M_2]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![M_1]\!](\overline{\lambda}m.\overline{@}[\![M_2]\!](\overline{\lambda}n.mn(\lambda a.\overline{@}\kappa a)))
\end{array}
$$

$$
\begin{array}{rcl}
[\![...]\!]' & : & \text{syntax} \to \text{syntax} \to \text{syntax} \\
[\![x]\!]' & = & \overline{\lambda}k.kx \\
[\![\lambda x.M]\!]' & = & \overline{\lambda}k.k(\lambda x.\lambda k.\overline{@}[\![M]\!]'k) \\
[\![M_1 M_2]\!]' & = & \overline{\lambda}k.\overline{@}[\![M_1]\!](\overline{\lambda}m.\overline{@}[\![M_2]\!](\overline{\lambda}n.mnk))
\end{array}
$$

The overlined $\overline{\lambda}$ and $\overline{@}$ correspond to functional abstractions and applications in the translation program and they are reduced at translation time. Thus, only the other abstractions and applications remain after translation. The transformation is formalized as two mutually recursive transformations to preserve tail-calls. A whole program is translated as $\overline{@}[\![M]\!](\overline{\lambda}m.m)$.

We encountered the basically same difficulties in formalizing and verifying this transformation as those in Danvy and Nielsen's transformation. To avoid clashes of variable names, we formalized the transformation as follows:

$$
\begin{array}{rcl}
[\![x]\!]_i & = & \overline{\lambda}\kappa.\overline{@}\kappa x \\
[\![\lambda x.M]\!]_i & = & \overline{\lambda}\kappa.\overline{@}\kappa(\lambda x.\lambda k.\overline{@}[\![M]\!]'k) \\
[\![M_1 M_2]\!]_i & = & \overline{\lambda}\kappa.\overline{@}[\![M_1]\!]_i(\overline{\lambda}m.\overline{@}[\![M_2]\!]_{i+1}(\overline{\lambda}n.mn(\lambda a_i.\overline{@}\kappa a_i)))
\end{array}
$$

$$
\begin{array}{rcl}
[\![x]\!]' & = & \overline{\lambda}k.kx \\
[\![\lambda x.M]\!]' & = & \overline{\lambda}k.k(\lambda x.\lambda k.\overline{@}[\![M]\!]'k) \\
[\![M_1 M_2]\!]' & = & \overline{\lambda}k.\overline{@}[\![M_1]\!]_0(\overline{\lambda}m.\overline{@}[\![M_2]\!]_1(\overline{\lambda}n.mnk))
\end{array}
$$

where we assume there are fresh variables a_i indexed by a natural number i.

Further, we reformulated the transformation with contexts for verification. We could do that because $[\![M]\!]_i$ can always be written $\lambda k.\mathbb{C}[kV]$ for some $\mathbb{C}$ and V. The reformulation of the transformation with contexts is shown in Figure 4. This specification of the transformation is not intuitive, but it was still easier to reason about it with automated theorem-proving tactics than the higher-order specification above.

With this formalization of the transformation, verification of its correctness was analogous to that of Danvy and Nielsen's transformation.

6. RELATED WORK

We have chosen first-order abstract syntax with variable names as representation of terms. McKinna and Pollack also formalized a theory of pure type systems in the theorem prover LEGO, with abstract syntax with variable names [10]. Recently, the Church-Rosser property of the lambda calculus was shown in Isabelle/HOL based on abstract syntax with variable names [18]. Formalization of reduction in these works differs from that of our work in the sense that they must consider β-reduction of open terms.

Another approach to abstract syntax uses de Bruijn indexes to represent bound variables. This approach has the advantage that α-equivalent terms have unique representation. This approach was taken by several works formalizing theory of programming languages in Isabelle/HOL [11, 1]. However, we think that the de Bruijn notation makes it difficult to formalize and reason about program transformations, as we discussed in Section 2. For example, formalization of Danvy and Filinski's CPS transformation based on de Bruijn indexes would be very complicated.

The third approach uses higher-order abstract syntax [15], where object-level abstractions are represented by meta-level abstractions. This approach often simplifies the formalization of binding in theorem provers, and is used for verification in theorem provers based on a logical framework [8]. However, it was reported that higher-order abstract syntax is difficult to treat in theorem provers such as Isabelle/HOL and Coq [7].

Danvy, Dzafic and Pfenning showed some syntactic properties of Danvy and Filinski's transformation [3]. They formalized their proofs in Elf, a constraint logic-programming language based on the logical framework [9, 14]. Their formalization is based on higher-order abstract syntax and thus they do not have the problems of fresh variables and α-equivalence. Instead, they needed to reformulate the transformation as a deductive system.

7. CONCLUSIONS

We have verified several versions of the CPS transformations based on abstract syntax with variable names. The main obstacles in our verification was treatment of variable names. First, we have shown that freshness of variables introduced by a transformation can be imposed by abstract syntax parameterized with the type of variables. Secondly, formalization of α-equivalence as a deductive system makes it possible to show α-equivalence by automated theorem-proving tactics. Finally, uniqueness of let-bound variables is imposed to simplify Danvy and Nielsen's transformation extended for let-expressions. Even though we needed to devise these methods, we think that it is feasible to verify correctness of program transformations based on abstract syntax with variable names.

Acknowledgments

This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Encouragement of Young Scientists of Japan, No. 13780193, 2001.

8. REFERENCES

[1] S. Ambler and R. L. Crole. Mechanized operational semantics via (co)induction. In *Proceedings of 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 221–238, 1999.

[2] A. W. Appel. *Compiling with Continuation*. Cambridge University Press, 1992.

[3] O. Danvy, B. Dzafic, and F. Pfenning. On proving syntactic properties of CPS programs. In *Proceedings of International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *ENTCS*, 1999.

[4] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361 – 391, 1992.

[5] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. In *Proceedings of Foundations of Software Science and Computation Structures, FOSSACS*, volume 2303 of *LNCS*, pages 98–113, 2002.

[6] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–391, 1972.

[7] J. Despeyroux and A. Hirschowitz. Higher-order syntax and induction in Coq. In F. Pfenning, editor, *Proceedings of the Fifth International Conference on Logic Programming and Automated Reasoning (LPAR)*, volume 822 of *LNAI*, pages 159–173, 1994.

[8] J. Hannan and F. Pfenning. Compiler verification in LF. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, 1992.

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

[10] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23:373–409, 1999.

[11] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[13] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[14] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[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, 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] G. L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

[18] R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names (*Barendregt was right after all ... almost*). In *Proceedings of the 12th International Conference Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 306–321, 2001.

[19] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 167–183, 1999.