Verifying the CPS transformation in Isabelle/HOL

Yasuhiko Minamide
University of Tsukuba
and
PRESTO, JST

Koji Okuma
University of Tsukuba

Abstract

We verified two versions of the CPS transformation in Isabelle/HOL: one by Plotkin and one by Danvy and Filinski. We adopted first order abstract syntax 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 introduce parameterized first order abstract syntax implemented as a polymorphic datatype. The verification of Danvy and Filinski's transformation requires us to reformulate the transformation in several respects. We also need to consider α-equivalence of terms for the verification. To make automatic theorem proving possible to some extent, we reformulate α-equivalence as a syntax-directed deductive system.

1 Introduction

Recent compilers apply various sophisticated program transformations to achieve high performance and to implement various advanced features of programming languages. It is not straightforward to show correctness of those transformations. As transformations are getting more and more sophisticated, correctness of hand-written proofs are getting more and more unreliable. Thus, it is is important to verify correctness of program transformations in theorem provers.

In this paper, we verified two versions of the CPS transformation in Isabelle/HOL: one by Plotkin [10] and one by Danvy and Filinski [2]. As representation of lambda terms we choose first order abstract syntax by the following reasons. First, it is simple and close to representation of lambda terms in hand-written proofs. Secondly, the first order abstract syntax is very close to intermediate languages used in compilers. Thus, verification based on first order abstract syntax may help verification of actual compilers in future research.

During our verification, we identified several problems in proving correctness of program transformations in Isabelle/HOL and developed formalization to overcome the problems.

The first problem is about fresh variables introduced by the transformation. Both the CPS transformations introduce some fresh variables, which must be different to the variables occurring in a term. To carry out verification based on simple first abstract syntax, this restriction must be maintained explicitly. Lemmas need explicit restrictions about variables and that makes it difficult to prove them in Isabelle/HOL. To overcome this problem, we introduce parameterized first order abstract syntax implemented as a polymorphic datatype. By adopting this abstract syntax, it is not necessary to maintain the restriction explicitly.

The verification of Danvy and Filinski's transformation was much more difficult than that of Plotkin's transformation. Danvy and Filinski's transformation is defined by two level specification [2]. The two level specification seems naturally formalized by using higher order functions in Isabelle/HOL. However, renaming of dynamic variables required for static level reduction cannot be directly formalized by functions of Isabelle. Further, the higher

order specification make it difficult to reason about the transformation. Thus we needed to reformulate the transformation in several respects.

The last problem is about treatment of α-equivalence of lambda terms. Our verification is based on the proof by Danvy and Filinski. However, it turns out that their proof relies on α-equivalence of lambda terms. However, the standard definition of α-equivalence is difficult to reason automatically. Thus, we reformulate α-equivalence as a syntax-directed deductive system. Based on this α-equivalence, we proved the lemmas concerning α-equivalence which are necessary to show correctness of Danvy and Filinski's transformation.

This paper is organized as follows. We start with a review of Isabelle/HOL and show how to formalize call-by-value lambda calculus in Isabelle/HOL. In Section 3 we outline verification of Plotkin's CPS transformation and introduce parameterized abstract syntax. In Section 4 we explain our formalization of Danvy and Filinksi's transformation and introduce reformulation of α-equivalence used in the verification. Finally we review related work and presents the conclusions.

2 Call-by-value lambda calculus in Isabelle/HOL

In this section we review Isabelle/HOL and show how to formalize call-by-value λ-calculus based on first order abstract syntax.

Isabelle is a generic, interactive theorem prover which can be instantiated with several different object logic [7]. Isabelle/HOL is an instantiation of Isabelle to Church's higher order logic and has features similar to Gordon's HOL system. In the rest of this paper We refer Isabelle/HOL just as Isabelle.

Isabelle uses normal logical operator like: $\forall, \exists, \wedge, \vee, \rightarrow$. There are also meta-logic operator $\equiv, \Rightarrow, \bigwedge$. The type system is similar to that of ML and contains ML-style polymorphic types. Types follows syntax for ML-types except that the function arrow is $\Rightarrow$. Isabelle supports generic inductive definitions [8]. Sets can be defined inductively with a keyword `inductive`. The inductive definition package provides definition of algebraic datatypes and primitive recursive functions.

In the rest of this section, we explain basic features of Isabelle by showing how to formalize call-by-value lambda calculus and its reduction semantics.

As we discussed in the introduction, we choose first order abstract syntax as representation of lambda terms. First order abstract syntax of lambda terms is defined by `datatype` definition as follows:

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

In this definition we choose to represent variables by natural numbers. Keyword `infixl` means 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 that of the intermediate language of many compilers. However, it turns out that it is not convenient to prove correctness of transformations later in this paper.

In this paper, we prove correctness of the CPS transformation as a transformation used in compilers of call-by-value languages. Thus, we consider only evaluation of closed programs. This simplifies our definition of substitution and reduction. The 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 for a closed term `V`, there is no need to rename variable `y` in the definition for abstraction.

We define the operational semantics of the calculus based on the following rule.

$$(\lambda x.M)V \to M[V/x] \quad \frac{M \to M'}{MN \to M'N} \quad \frac{N \to N'}{VN \to VN'}$$

This reduction relation can be easily defined as inductively defined relation in Isabelle as follows:

```
consts
  eval :: (term * term) 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 `term`. An expression, $[\![A_1; \ldots; A_n]\!] \Rightarrow A$ is abbreviation for $A_1 \Rightarrow \ldots A_n \Rightarrow A$. To ensure that substitution is applied only if `V` is closed, the rule for β-axiom is restricted to the case where `V` is closed.

3 The CPS transformation of Plotkin

We outline verification of Plotkin's CPS transformation in this section. The key to our verification is to introduce parameterized first order syntax implemented as a polymorphic datatype.

We first review Plotkin's CPS transformation. The CPS transformation transforms programs into a special form called continuation-passing style. The CPS transformation is used in several compilers of call-by-value functional languages. The transformation is defined as follows:

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

where k, m and n are fresh variables. A whole program is translated by the initial continuation $\lambda x.x$ as follows: $[\![M]\!](\lambda x.x)$.

This transformation can be formalized as a primitive recursive function in Isabelle if we fix the variables k, m and n. The following is formalization of the transformation where k, m, and n are represented by the natural numbers 0, 1, and 2 respectively. The definition introduces the postfix notation `M*` for `CPStrans M`.

```
consts
 CPStrans:: term => term               ("_*" [300] 300)
primrec
 (Const n)* = Abs 0 (Var 0 $ Const n)
 (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)))))
```

However, this formalization is difficult to treat: the transformation is valid only if the variables 0, 1, and 2 do not occur in a program. This condition must be maintained in every lemma. That makes it difficult to prove various lemmas, especially to prove lemmas with automatic theorem proving tactics.

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

```
datatype 'a term = Var 'a
                 | Abs 'a "'a term"
                 | $ "'a term" "'a term" (infixl 100)
```

The type of terms `'a term` is parameterized by the type of the variables `'a`. The type of variables can be any type: our definition of substitution does not include any renaming and thus the type with a finite number of values can also 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 case of the CPS transformation we represent the set of variables of the target language the following data type.

```
datatype 'a variable = Orig 'a  ("_~" [150] 150)
                     | k
                     | m
                     | n
```

The variables introduce by the transformation are represented by `k`, `m` and `n`. A variable `x` in the source langauge is translated into `x~` that is abbreviation of `Orig x`.

Using this abstract syntax, the CPS transformation is defined as a primitive recursive function with the following type.

```
'a term => ('a variable) term
```

The definition of the transformation can be obtained easily by refining that for the previous abstract syntax.

Using this abstract syntax, the following substitution lemma for the CPS transformation can be proved almost automatically.

Lemma 1 $[\![M]\!][\Psi(V)/x] = [\![M[V/x]]\!]$

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 Ploktin's proof and its Plotkin's proof is 23 lines in length. The following is the proof of the lemma in Isabelle. It is proved by induction on structure of `M`. The lemma `value_CPS` proved earlier is used in the proof.

```
lemma   "[| isValue V; closed V |] ==> ((M[V/x])* = (M* )[Psi(V)/(x~)])"
proof(induct M,auto)
  assume "isValue V"
  thus "V* = Abs k (Var k $ Psi(V))"
    by (rule value_CPS)
qed
```

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

Lemma 2

- $[\![M]\!]K \rightarrow^{+} M{:}K$
- *If* $M \rightarrow N$ *then* $M{:}K \rightarrow N{:}K$.

Plotkin's proofs of these properties are 17 and 25 lines in length respectively. Our proof scripts consist of 88 and 110 lines. It was not so difficult to translate Plotkin's proofs into the proofs in Isabelle.

4 The CPS transformation of Danvy and Filinski

We also verified Danvy and Filinski's CPS transformation in Isabelle/HOL. The verification was much more difficult than that of Plotkin's CPS transformation. In addition to the problem we described in the previous section, we encountered three problems in formalizing and proving the correctness of the transformation: treatment of fresh variables introduced by the transformation, two level specification, and α-equivalence. In this section, we outline our formalization of the transformation without using Isabelle/HOL. The concrete formalization in Isabelle/HOL can be obtained directly by translating the material in this section.

We first review Danvy and Filinski's CPS transformation, which eliminates redundant β-redexes introduced in the transformation of Plotkin. The transformation is formalized by two level lambda calculus as follows:

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

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

The overlined λ and @ correspond to functional abstractions and applications in the translation program. 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 $\overline{@}[\![M]\!](\overline{\lambda} m.m)$. The following is a translation of term xyz.

$$\overline{@}[\![x\,y\,z]\!](\overline{\lambda} m.m) = x\,y\,(\lambda a.a\,z\,(\lambda a.a))$$

The first problem about this transformation is about fresh variables introduced in the transformation. For Danvy and Filinski's CPS transformation it is not enough to introduce a fixed number of new variables. Many instances of the variable a may be necessary simultaneously. The following example clarifies the problem.

$$\overline{@}[\![(x_1x_2)(x_3x_4)]\!](\lambda y.y) = (x_1x_2)(\lambda a_1.x_3x_4(\lambda a_2.a_1a_2(\lambda y.y)))$$

During the computation of x_3x_4, the value of x_1x_2 must be preserved. Thus a_1 and a_2 must be different variables. This is not clear from the definition based on the two-level lambda calculus and we first thought that it was enough to introduce a fix number of variables as Plotkin's CPS transformation. In the specification based on the two level lambda calculus, it is assumed that crashes of variable names are eliminated by renaming of bound variables and during the translation of a program, dynamic variables are renamed if necessary. Since this renaming cannot be performed if we formalize static abstractions as Isabelle functions, it is not possible to formalize the specification above directly in Isabelle.

To eliminate a need of implicit renaming of dynamic variables, we take an approach of generating new variables explicitly. Most compilers achieves the hygiene condition on variables in this approach. To formalize this approach, we revise the definition of the transformation so that the translation of $[\![\cdot]\!]$ takes a natural number as an argument. The natural number is used to generate fresh variables. The following is the definition of the revised transformation:

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

$$[\![M_1 M_2]\!]'' = \overline{\lambda} k.[\![M_1]\!]_0\overline{@}\overline{\lambda} m.[\![M_2]\!]_1\overline{\lambda} n.mnk$$

where we assume there are fresh variables a_i indexed by a natural number i. This specification can be formalized directly in Isabelle/HOL.

The second problem is about formalization of the transformation as higher order functions in Isabelle. The automated reasoning tactics of Isabelle/HOL did not work effectively for this specification. Even very simple lemmas cannot be proved automatically. The higher order functions make it difficult to reason about the transformation automatically. Thus it was not feasible to do verification based on this specification.

To overcome this problem, we represent static lambda abstraction as contexts. Context $\mathbb{C}$ is defined as follows:

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

By checking the definition of the transformation, it is shown that this form of contexts are enough to represent static continuation. After formalizing the contexts and some operations on them in Isabelle, it was straightforward to reformulate the transformation in this representation of static lambda abstraction and to show that two versions of the transformation are equivalent. However, the definition of the transformation becomes much more complicated and difficult to understand in this representation. Thus we used this representation for simple lemmas which can be proved almost automatically and the previous representation when we need to write proof scripts by ourselves.

The third problem is about α-equivalence of terms. Danvy and Filinki proved the correctness of their CPS transformation by showing the following property.

$$M \to N \quad \Rightarrow \quad \overline{@}[\![M]\!]k \to^* \overline{@}[\![N]\!]k$$

However, this property does not hold in our formalism based on first order abstract syntax. In fact, $[\![M]\!]k$ is evaluated not to $[\![N]\!]k$, but a term which is α-equivalent to $[\![N]\!]k$. Thus, we need to show the following property.

$$M \to N \quad \Rightarrow \quad [\![M]\!]k \to^* N' \text{ and } [\![N]\!]k =_\alpha N' \text{ for some } N'$$

In order to prove this property, we first attempted to use the standard α-equivalence of lambda terms. However, it is found that the standard α-equivalence is not convenient for our proof in the following reasons.

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 definition of α-conversion is not sufficient to derive necessary equivalence of terms. For example, if we have only two variables x and y, the following equivalence cannot be derived from rules.

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

Furthermore, it is rather difficult to reason about the standard definition of α-equivalence automatically. The standard definition of α-equivalence contains the rule for transitivity and is not syntax-directed. Thus it is not straightforward to determine whether two terms are α-equivalent or not automatically. In fact, automated reasoning tactics often fails to prove two terms are α-equivalent.

To avoid these problems, we reformulated α-equivalence as a syntax directed deductive system. Let Var be a set of variables. We say $E \subseteq \mathrm{Var} \times \mathrm{Var}$ is renaming if $(x, y) \in E$ and $(x', y') \in E$ implies $x = x' \Leftrightarrow y = y'$. This restriction of renaming ensures that any renaming can be extended to a permutation on the set of variables. The extension of a renaming $E \oplus (x, y)$ is defined as follows:

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

The extension $E \oplus (x, y)$ eliminates renaming related to x and y. Then, we define a deductive system with judgments of the form $E \vdash M = N$, which means M and N are α-equivalent

under renaming E. The rules of the deductive system is defined as follows:

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

We write $\vdash M =_\alpha N$ iff $\mathrm{ID} \vdash M =_\alpha N$ where ID is the identity relation $\{(x,x) \mid x \in \mathrm{Var}\}$. As we discuss later, it is shown that $\vdash M =_\alpha N$ is equivalent $M =_\alpha N$ if the set of variables is infinite.

Based on this α-equivalence we verified Danvy and Filinski's CPS transformation. In our verification we needed to show α-equivalence of expressions below[1].

$$[\![M]\!]_0 =_\alpha [\![M]\!]_1$$

In order to prove this lemma, we first proved the following weaker lemma.

$$\{(a_n, a_{n+1}) \mid n \in N\} \vdash [\![M]\!]_0 =_\alpha [\![M]\!]_1$$

This can be easily proved by induction on M in Isabelle. Then $[\![M]\!]_0 =_\alpha [\![M]\!]_1$ can be obtained from $a_i \notin FV([\![M]\!]_0)$ for any natural number i.

For our verification, we needed the following substitution lemma for α-equivalence.

Lemma 3 (Substitution) *If $E \oplus (x,y) \vdash M =_\alpha M'$ and $\emptyset \vdash V =_\alpha V'$ then $E \vdash M[V/x] =_\alpha M'[V'/y]$.*

The proof of this lemma was difficult than I expected and the development of the theory of α-equivalence requires about 1000 lines of Isabelle code. The development also contains some other basic lemmas including the following properties.

- If $E_1 \subseteq E_2$ and $E_1 \vdash M =_\alpha N$ then $E_2 \vdash M =_\alpha N$.
- $E \vdash M =_\alpha N$ iff $E^{-1} \vdash N =_\alpha M$.
- If $E_1 \vdash L =_\alpha M$ and $E_2 \vdash M =_\alpha N$ then $E_1 \circ E_2 \vdash L =_\alpha N$.

After the verification of the CPS transformation, we also showed that the two version of α-equivalence are equivalent if the set of variables is infinite. We first formalized the standard α-equivalence in Isabelle based on the work by Vestergaard and Brotherston [11]. We define a function $\mathrm{Capt}_x(M)$ which returns the set of all binding variables in M captured that have free occurrences of x in their scope.

$$\begin{array}{rcl} \mathrm{Capt}_x(y) & = & \emptyset \\ \mathrm{Capt}_x(M_1 M_2) & = & \mathrm{Capt}_x(M_1) \cup \mathrm{Capt}_x(M_2) \\ \mathrm{Capt}_x(\lambda y.M) & = & \begin{cases} \{y\} \cup \mathrm{Capt}_x(M) \cup \mathrm{Capt}_x(M_2) & \text{if } x \neq y \text{ and } x \in FV(M) \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

Then the axiom of α-equivalence is defined as follows:

$$\frac{y \notin \mathrm{Capt}_x(M)}{\lambda x.M = \lambda y.M[y/x]}$$

We formalize the standard α-equivalence $M \equiv_\alpha N$ as the transitive and compatible closure relation of this axiom.

After formalizing two versions of α-equivalence, it is not so difficult to show that if $M =_\alpha N$, then $\vdash M =_\alpha N$. The axiom of the standard alpha equivalence is obtained as follows:

$$\frac{\mathrm{ID} \oplus (x,y) \vdash M = M[y/x]}{\mathrm{ID} \vdash \lambda x.M = \lambda y.M[y/x]}$$

[1]This is not precise because $[\![M]\!]_0$ and $[\![M]\!]_1$ themselves are not terms. We proved some extensions of the property in our verification.

theory name	description	lines	lemmas
Lt	abstract syntax, reduction	136	9
Plotkin	Plotkin's CPS	358	17
Alpha	standard α-equivalnce	550	24
NewAlpha	our α-equivalnce	943	41
AlphaEq	equivalence of two versions of α-equivalnce	536	13
Danvy	Danvy and Filinski's CPS	1142	75

Table 1: Theory files

where $\mathrm{ID} \oplus (x, y) \vdash M = M[y/x]$ is shown by induction on structure of M. This implication holds even if the set of variables is finite.

On the other hand, the following inverse implication was difficult to show in Isabelle. The proof required about 370 lines of code and the reasoning using a fact that some set is infinite was difficult in Isabelle.

Lemma 4 *Let the set of variables is infinite. If $M =_\alpha N$, then $M \equiv_\alpha N$.*

5 Related work

In this paper, we have chosen first order abstract syntax as representation of terms by the reason described in the introduction. Recently, the Church-Rosser property of the lambda calculus was shown in Isabelle/HOL based on first order abstract syntax [11]. However, most previous work on program languages took other approaches in representation of terms.

One approach which is often taken is using de Bruijn index. This approach has an advantage that it can represent bindings without variable names, so problem of variable names such as new name generation, or occurrence check can be avoided. This approach is taken by several works formalizing theory of programming languages in Isabelle/HOL [6, 1] However, we think that de Bruijn index sometimes make it difficult to formalize and reason about program transformations. For example, if Danvy and Filinski's CPS is formalized based on de Bruijn index, renaming of dynamic variables is necessary when static application is reduced.

Another approach is using higher-order abstract syntax[9]. It represents abstractions with meta level abstractions. This approach often simplifies proofs in theorem provers and is taken for theorem provers based on a logical framework [5]. However, higher order abstract syntax is difficult to treat in theorem-provers like Isabelle/HOL and Coq [4].

Our verification of Danvy and Filinski's CPS transformation is based on their proof. There is another proof by Danvy and Nielsen which explicitly treats static level reduction [3]. Their proof is based on the Church-Rosser and SN properties of static level reduction. Since these properties seems difficult to verify in theorem prover, their proof will be more difficult to formalize in Isabelle/HOL.

6 Conclusions

We have verified two versions of the CPS transformations. Table 6 summarizes the files we developed in Isabelle/HOL. For each theory file, the figure shows the number of lines and the number of lemmas in it. These files can be obtained from the following URL.

`http://www.score.is.tsukuba.ac.jp/~minamide/cps-isabelle/`

In the verfication of the CPS transformations, `Plotkin` and `Danvy`, about 2/3 of lemmas are proved almost automatically in one line of proof script. We think that this is achivied by our parametrized abstract syntax.

The other part of verification of Plotkin's transformation was not difficult, either. The proofs of non-trivial lemmas almost directly corresponds to the proofs by Plotkin.

On the other hand, verification of Danvy and Filiski's transformation required much more time: most difficulties come from its two level specification. We think that we need to develop better treatment of two level specification in Isabelle/HOL. The theory of α-equivalence also has important rule in verification of the transformation. In order to make verfication of program transformations easier we think that we need to develop basic theory of programming languages before verifying of program transformations.

Acknowledgements

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.

References

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

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

[3] O. Danvy and L. R. Nielsen. A higher-order colon translation. In *Proceedings of the 5th International Symposium on Functional and Logic Programming*, LNCS, pages 78–91, 2001.

[4] J. Despeyroux and A. Hirschowitz. Higher-order syntax and induction in Coq. In F. Pfenning, editor, *proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, volume 822, pages 159–173. Springer-Verlag LNAI, July 1994. Also appears as INRIA Research Report RR-2292 (June 1994).

[5] J. Hannan and F. Pfenning. Compiler verification in LF. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[6] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Jounarl of Automated Reasoning*, pages 51–66, 2001.

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

[8] L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 187–211. MIT Press, 2000.

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

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

[11] 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 A. Middeldorp, editor, *Proceedings of RTA-12*, volume 2051 of *LNCS*. Springer-Verlag, 2001.