Bracket Abstraction Preserves Typability
A formal proof of Diller–algorithm–C in PVS

Sjaak Smetsers and Arjen van Weelden

Institute for Computing and Information Sciences,
Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
S.Smetsers@science.ru.nl, A.vanWeelden@cs.ru.nl

Abstract. Bracket abstraction is an algorithm that transforms lambda expressions into combinator terms. There are several versions of this algorithm depending on the actual set of combinators that is used. Most of them have been proven correct with respect to the operational semantics. In this paper we focus on typability. We present a fully machine verified proof of the property that bracket abstraction preserves types; the types assigned to an expression before and after performing bracket abstraction are identical. To our knowledge, this is the first time that (1) such proof has been given, and (2) the proof is verified by a theorem prover. The theorem prover used in the development of the proof is PVS.

1 Introduction

Pioneer functional programming languages used combinators [1], such as S, K, and I, originally developed by Schönfinkel [2], to define the semantics of expressions. The function definitions are translated via lambda expressions to combinators, this last step has become known as bracket abstraction. The first implementations of interpreters and compilers, e.g., SASL [3], also used combinators to evaluate or compile untyped functional programs. Designing the 'best' set of combinators has temporarily been a competitive sport. The race to construct the fastest evaluators has spawned dozens of additional combinators and complex bracket abstraction algorithms, e.g., Abdali [4], Turner [5], Diller [6], and Bunder [7]. Super combinators, cf. Peyton Jones [8], were later introduced and program optimizations were defined using extensible super combinators instead of a fixed set of combinators. Eventually, combinators were discarded in favor of generating efficient native machine code. Nowadays, compilers for strongly typed functional languages generate code comparable in efficiency to C.

Using one such strongly typed functional programming language, Clean [9], the authors have written an interactive shell [10] that can type check functional command-line expressions before executing them. The translation of command-line expressions, which support all basic functional language constructions, uses a variant of bracket abstraction. We wanted to show that this is possible using merely Dynamics with their apparently limited set of operations. This forced us to do the type inference after bracket abstraction. We are convinced that the algorithm used is sound with respect to the operational semantics, since it has been used innumerably as an implementation of functional languages. However, to our knowledge, nobody has ever shown that this variant of bracket abstraction preserves the principal type.

In this paper we give a formal proof, using the theorem prover PVS [11], which indicates that Diller–algorithm–C [6] without η-conversion preserves typability. One would never attempt such a proof entirely by hand as it contains too many cases, while its reasonable complexity allows it to be rigorously specified in a formal way. All the proofs presented in this paper can be downloaded from the following website: `http://www.cs.ru.nl/A.vanWeelden/bracket/`.

Further on in this paper, we proceed to explain a few things about Dynamics (Sect. 2), the translation from functional expressions to combinators (Sect. 3), and the theorem prover PVS (Sect. 4). We also share some interesting issues of the proof itself (Sect. 5). Related work is discussed in Sect. 6, and we conclude and mention future work in Sect. 7.

2 Dynamics in Clean and the Shell Written in Clean

Clean [9] is a strongly typed, pure, and lazy functional programming language, much like Haskell [12], and not entirely unlike strict functional languages. Such languages are based on the concept of functions and consist of expressions (usually without side effects), function definitions, algebraic data type definitions, pattern matching, and (data) recursion. They usually feature complex static type systems and checkers, including type inference.

Clean features a hybrid type system, where run-time type checking is integrated into the static (compile-time) type system. The system is based on the theories of Abadi [13] and Pil [14]. Any expressions can be wrapped, together with their static (polymorphic) type, into an object with the static type *Dynamic*. Those expressions are, as usual, compiled and statically type checked with respect to the functions and types of the program that defines them. The example below shows the definition of the factorial function, which is wrapped in a dynamic and extracted again using a type pattern match.

```
fac 0 = 1                                   // factorial function
fac n = fac (n - 1) * n

dynamicFac = dynamic fac :: Int → Int       // wrap in a dynamic

matchDynamic (f :: Int → Int) = f           // unwrap by matching

example = matchDynamic dynamicFac 10        // apply; yields 3628800
```

Dynamics can be serialized (written to disk or over a network) while preserving sharing and cycles. The expressions inside those dynamics contain data and code, i.e., functions and closures. Therefore, their serialization will often contain references to the compiled code of the defining program. Dynamics can also be deserialized (read from disk) in other programs. Any necessary code will be automatically and lazily linked into the reading program as implemented by Vervoort [15]. Once the Dynamic object exists inside a running program, the program can pattern match on the original static type of the expression contained within the Dynamic.

```
dynamicApply dynf dynx = case (dynf , dynx) of
      (f :: a → b , x :: a) → dynamic f x :: b
      (g , y) → abort "Cannot unify formal and actual argument"
```

In the example above, we show a more complex example where we apply one Dynamic to another at run time in a type-safe way. The type checker can statically check the usage of type pattern variables a and b in the application of a function that matches the type pattern $a \to b$ on an argument that matches type a. Obviously, this results in something that matches some type b. At run time an attempt is made to unify the type of the argument of f (from the first Dynamic) with the type of x (from the second Dynamic). If it succeeds, the substitution required for that unification is also applied to the type of the result of f, which is used as the resulting type of the application. The result after application can only be stored in a Dynamic again, since the actual type at run-time is unknown to the static type checker.

Using Dynamics to type only applications, in the manner done in the examples above, our shell [10] is capable of type checking/inferring any expression of a simple functional language. This is enabled by existing translation schemes, cf. Peyton Jones [8], that can transform any language construct in a functional programming language to lambda expressions with explicit letrec-sharing, cf. Hindley [16]. The source language used throughout this paper contains, therefore, only applications, lambda expressions, variables and letrec expressions.

$$\begin{aligned} Expr &::= Expr\ Expr \mid \lambda\ Var\ .\ Expr \mid Var \mid \textbf{letrec}\ Var = Expr\ \textbf{in}\ Expr \\ Var &::= x \mid y \mid z \mid \cdots \end{aligned}$$

Using a syntax tree, we can infer the types of applications and constants:

```
:: Expr = Con Dynamic | App Expr Expr | Lam Var Expr | Var Var
:: Var = Identifier String

type (Con dyn)    = dyn
type (App e1 e2) = case (type e1 , type e2) of
                        (f :: a → b , x :: a) → dynamic f x :: b
type (Lam v b)    = case type b of (e :: a) → dynamic λx.e :: ?
type (Var v)      = abort "cannot type a single variable"
```

However, static type inference of lambda expressions is problematic. The type of the body of the lambda expression (b in the code above) cannot be inferred because it is an open expression (contains the variable v). To infer the type using Dynamics, we need the run-time value of the variable, which is not available at compile time. We solved this problem using bracket abstraction, which removes all variables from an expression. This forced us to do type inference after bracket abstraction.

$$\begin{array}{rcl}
I\ x \Rightarrow & x & : \alpha \rightarrow \alpha \\
K\ x\ y \Rightarrow & x & : \alpha \rightarrow \beta \rightarrow \alpha \\
S\ f\ g\ x \Rightarrow & f\ x\ (g\ x) & : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\
B\ f\ g\ x \Rightarrow & f\ (g\ x) & : (\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta \\
C\ f\ g\ x \Rightarrow & f\ x\ g & : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \alpha \rightarrow \gamma \\
S'\ h\ f\ g\ x \Rightarrow & h\ (f\ x)\ (g\ x) & : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\delta \rightarrow \alpha) \rightarrow (\delta \rightarrow \beta) \rightarrow \delta \rightarrow \gamma \\
B'\ h\ f\ g\ x \Rightarrow & h\ f\ (g\ x) & : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow (\delta \rightarrow \beta) \rightarrow \delta \rightarrow \gamma \\
C'\ h\ f\ g\ x \Rightarrow & h\ (f\ x)\ g & : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\delta \rightarrow \alpha) \rightarrow \beta \rightarrow \delta \rightarrow \gamma \\
Y\ f \Rightarrow & f\ (Y\ f) & : (\alpha \rightarrow \alpha) \rightarrow \alpha
\end{array}$$

Fig. 1. Combinator reduction rules and their types.

3 From Expression to Combinators

The combinators used in our target language, both in this paper and in the shell[1], are from the algorithm–C by Diller [6] and are shown in Fig. 1. The target language consists of variables, applications, and constants, which are the combinators and Dynamics. Of course, bracket abstraction should remove all variables from a closed expression. The implementation of the shell uses only Dynamics as constants. The combinators can easily be expressed using lambda expressions and be put in a Dynamics with their static types. In contrast, the algorithm used in the proof does not use Dynamics as constants. This does not influence typability because Dynamics are already typed and not altered in any way by bracket abstraction. One could easily add any Dynamic to the set of combinator constants, much like the super-combinator approach.

$$\begin{array}{rcll}
[\![e_1\ e_2]\!] &=& [\![e_1]\!][\![e_2]\!] & \text{(application)} \\
[\![\lambda x.e]\!] &=& [\![[\![e]\!]]\!]_x & \text{(abstraction)} \\
[\![x]\!] &=& x & \text{(variable)} \\
[\![\mathit{let}\ x = e_1\ \mathit{in}\ e_2]\!] &=& [\![\lambda x.e_2]\!]\ [\![e_1]\!] & \text{(non-recursive let)} \\
[\![\mathit{letrec}\ x = e_1\ \mathit{in}\ e_2]\!] &=& [\![\lambda x.e_2]\!]\ (Y\ [\![\lambda x.e_1]\!]) & \text{(monomorphic let)} \\
[\![\mathit{letrec}\ x = e_1\ \mathit{in}\ e_2]\!] &=& [\![e_2]\!][Y[\![\lambda x.e_1]\!]/x] & \text{(polymorphic let)}
\end{array}$$

Fig. 2. Translation from expressions to combinators via lambda expressions.

Translation from the source language to combinators uses lambda expressions as an intermediate step. The translation $[\![e]\!]$ of an expression e in the source language is shown in Fig. 2. We show translations for both non-recursive *let* and recursive *letrec*s, for which there is a monomorphic and a polymorphic variant. The shell implements the monomorphic *letrec* and it writes polymorphic function definitions to disk using Dynamics. By reading them back in when they are used

[1] The shell written in Clean actually implements the fix-point combinator by the recursive binding `let x = f x in x` to construct efficient cycles.

in other expression, it achieves the same effect as the polymorphic *letrec*. The expression inside a Dynamic is shared, not substituted, but the (polymorphic) type can be instantiated multiple times.

$$\begin{array}{rll}
[\![\, x \,]\!]_x &= I & \\
[\![\, e \,]\!]_x &= K\ e & \text{if } x \notin FV(e) \\
[\![\, (e_1\ e_2)\ e_3 \,]\!]_x &= B'\ e_1\ e_2\ [\![\, e_3 \,]\!]_x & \text{if } x \notin FV(e_1) \wedge x \notin FV(e_2) \\
[\![\, (e_1\ e_2)\ e_3 \,]\!]_x &= C'\ e_1\ [\![\, e_2 \,]\!]_x\ e_3 & \text{if } x \notin FV(e_1) \wedge x \notin FV(e_3) \\
[\![\, (e_1\ e_2)\ e_3 \,]\!]_x &= S'\ e_1\ [\![\, e_2 \,]\!]_x\ [\![\, e_3 \,]\!]_x & \text{if } x \notin FV(e_1) \\
[\![\, e_1\ e_2 \,]\!]_x &= B\ e_1\ [\![\, e_2 \,]\!]_x & \text{if } x \notin FV(e_1) \\
[\![\, e_1\ e_2 \,]\!]_x &= C\ [\![\, e_1 \,]\!]_x\ e_2 & \text{if } x \notin FV(e_2) \\
[\![\, e_1\ e_2 \,]\!]_x &= S\ [\![\, e_1 \,]\!]_x\ [\![\, e_2 \,]\!]_x &
\end{array}$$

Fig. 3. Our variant of bracket abstraction, based on priority from the top down.

The bracket abstraction algorithm $[\![\,\cdot\,]\!]_x$, over a variable x used in the shell and throughout this paper is defined in Fig. 3. It is almost the same as algorithm–C and also very similar to Abs/Dash/4 by Joy et al. [17], who show that it produces good code and that improvements are hard and yield little effect. In contrast to algorithm–C, we do not use η-conversion because it obviously does not preserve the principal type, as indicated by the following example:

$$\lambda x.\lambda y.x\,y : (\alpha \to \beta) \to \alpha \to \beta \rightsquigarrow_\eta \lambda x.x : \alpha \to \alpha \rightsquigarrow_{[\![\ ]\!]} I : \alpha \to \alpha.$$

4 PVS

As a short introduction to PVS, we will briefly recall the basics (see also [11]). PVS (*P*rototype *V*erification *S*ystems) offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. Both the use of basic types, like integers, booleans and reals, and compound types (built with type constructors such as records, tuples, and function types) are permitted. New, possibly recursive, data types can be introduced via algebraic data type definitions. As an example of a user defined data type, consider the following parameterized definition of a binary tree:

```
BinTree[V : TYPE] : DATATYPE BEGIN
  leaf   : leaf?
  node(el:V, left, right: BinTree) : node?
END BinTree
```

The data type has two *constructors*, `leaf` and `node`, with which trees can be built. In addition, two *recognizers* `leaf?` and `node?` are defined (observe that PVS allows question marks as constituents of identifiers), which can be used as predicates to test whether or not a tree object starts with the corresponding constructor. The field names `el`, `left` and `right` can be used as *accessors* to extract these components from a node. However, for extraction purposes, it is often more convenient to use the built-in pattern matching via **CASES** expressions. Consider, for example, the following function `tree2List` that collects all elements of a tree and places them in a list.

```
tree2List(t:BinTree) : RECURSIVE list[V]  =
    CASES t OF
        leaf: null,
        node(e,l,r): append(tree2List(l),cons(e,tree2List(r)))
    ENDCASES
MEASURE size(t)
```

The **MEASURE** specification is mandatory when defining a recursive function, such as the `tree2List` function shown above. In PVS, it is required that all functions are total. This measure is used to show that the function terminates. This is realized by generating a proof obligation (a so-called *TCC*, *T*ype *C*orrectness *C*ondition) indicating that the measure strictly decreases at each recursive call. Obviously, in this case the size of the tree fulfills this property. But how can an appropriate measure be given for the function `size` itself? A solution is to have data type definitions, which cause PVS to generate a number of functions and axioms, which can be used freely in a theory importing that data type. Among them, one example is the ordering $\ll$ on trees indicating whether the first argument tree is a subtree of second one. In a measure specification, $\ll$ can be used as follows:

```
size(t:BinTree) : RECURSIVE int  =
    CASES t OF
        leaf: 0,
        node(e,l,r): size(l) + size (r) + 1
    ENDCASES
MEASURE t BY <<
```

PVS specifications are organized into parameterized theories that may contain declarations of functions, axioms, theorems, etc.. The PVS language provides the customary arithmetic and logical operators, function application, lambda abstraction, and quantifiers. Names may be overloaded, including those of built-in operators such as $<$ and $+$.

Higher-order logic forms the base for the theorem prover of PVS, i.e., PVS admits to quantify over predicates. For example, consider the following lemma that enables the use of course–of–values induction on the size of trees in a proof.

```
tree_size_induction : LEMMA
    (∀ (p:pred[BinTree]): (∀ (g:BinTree): (∀ (s:BinTree):
        size(s) < size(g) ⇒ p(s)) ⇒ p(g)) ⇒ (∀ (c:BinTree): p(c)))
```

This lemma can be proven with the predefined induction principle `NAT_induction`. In order to facilitate the development of proofs, PVS provides a collection of proof commands and predefined combinations of proof commands, so-called *proof* *strategies*. During the construction of a proof, PVS constructs and maintains a proof tree. The goal of a proof is to apply proof commands/strategies such that all the leaves of the proof tree are recognized as true. Therefore, the proof itself is just a sequence of proof strategies that converts the initial proof tree into a complete one. Such proof sequences are available in textual form making them easy to edit or display, or even to rerun. Usually, they are kept in a separate proof file and can be inspected at any time during a proof session.

5 The Proof in PVS

In this section we will use PVS to prove that bracket abstraction preserves typability. We will first concentrate on the *monomorphic* case.

5.1 Bracket Abstraction

We initiate by introducing two data types to represent the source language `FUNC` and destination language `COMB` of our translation.

Our source language is essentially the lambda calculus enriched with a `letr` construct to specify single recursive definitions. The addition of multiple recursion and pattern matching is straightforward, but is left out for reasons of simplicity.

```
FUNC [V : TYPE] : DATATYPE BEGIN
  vari (id:V)                          : vari?
  appl (fun, arg:FUNC)                 : appl?
  lamb (l_var:V, l_body:FUNC)          : lamb?
  letr (b_var:V, b_def, b_body:FUNC): letr?
END FUNC
```

The destination language consists of combinator expressions. Our representation is parametric in both variables and combinators.

```
COMB[V:TYPE, C:TYPE] : DATATYPE BEGIN
    c_var   (v_id: V) : c_var?
    c_const (c_id: C) : c_const?
    c_appl  (c_fun, c_arg: COMB) : c_appl?
END COMB
```

The concrete combinators are introduced via an enumeration type (see Fig. 1).

```
SKI: TYPE = {S,K,I,Y,B,C,S1,B1,C1}
```

We shall now present the bracket abstraction algorithm `lam2Ski` (see Fig. 2):

```
lam2Ski(e:FUNC) : RECURSIVE COMB =
    CASES e OF
        vari(v)      : c_var(v),
        appl(f, a)   : c_appl(lam2Ski(f),lam2Ski(a)),
        lamb(v,e)    : abstr(lam2Ski(e),v),
        letr (v,d,e) : c_appl(abstr(lam2Ski(e),v),
                              c_appl(c_const(Y),abstr(lam2Ski(d),v)))
    ENDCASES
MEASURE e BY <<
```

where abstr is a recursive function that builds up the combinator expression for the distribution of a parameter v (see Fig. 3). CAppl2 and CAppl3 are just helper functions introduced to improve readability.

```
CAppl2(c:SKI, a1,a2: COMB)    : COMB =
c_appl(c_appl(c_const(c),a1),a2) CAppl3(c:SKI, a1,a2,a3: COMB) :
COMB = c_appl(CAppl2(c,a1,a2),a3)

abstr(e:COMB,v:V) : RECURSIVE COMB =
     IF fvs(e)(v)
     THEN CASES e OF
              c_var(w)     : c_const(I),
              c_appl(f,a)  :
                  IF c_appl?(f) ∧  ¬fvs(c_fun(f))(v)
                  THEN LET g = c_fun(f), b = c_arg(f) IN
                      IF ¬fvs(b)(v)
                      THEN CAppl3 (B1, g, b, abstr(a,v))
                      ELSIF ¬fvs(a)(v)
                      THEN CAppl3 (C1, g, abstr(b,v), a)
                      ELSE CAppl3 (S1, g, abstr(b,v), abstr(a,v))
                      ENDIF
                  ELSIF ¬fvs(f)(v)
                  THEN CAppl2 (B, f, abstr(a,v))
                  ELSIF ¬fvs(a)(v)
                  THEN CAppl2 (C, abstr(f,v),a)
                  ELSE CAppl2 (S,abstr(f,v),abstr(a,v))
                  ENDIF
          ENDCASES
     ELSE c_appl(c_const(K),e)
     ENDIF
MEASURE e BY <<
```

The standard subterm order of FUNC and COMB are used as measures in lam2Ski and in abstr, respectively. In both cases these orders are denoted by $\ll$. The predicate fvs indicates whether a given variable freely occurs in an expression. It is defined using the reduce function for the COMB data type. This (fold-like) operation is internally generated by PVS and can often be used as a substitute for recursion.

```
fvs:[COMB → PRED[V]] = reduce(singleton, λ(c:C):∅,
∪)
```

The equivalence between lam2Ski and abstr and the specifications of these transformations in Sect. 3, is self-evident.

5.2 Typing

In order to represent types for both combinators and lambda expressions, we introduce the following data type.

```
TYPES[V : TYPE] : DATATYPE BEGIN
    t_var  (t_var:V)            : t_var?
    t_arr  (t_arg, t_res:TYPES) : t_arr?
END TYPES
```

This definition is self-explanatory, as well as the definitions of *substitution* and *(substitution) instance*. The latter is denoted as a binary predicate $\leq$ on types. Here subst is the customary lifting of substitutions to types. It can be easily expressed in terms of reduce.

```
Substitution : TYPE = [V → TYPES] subst(s:Substitution): [TYPES →
TYPES] = reduce(s,t_arr);

t1, t2: VAR TYPES ≤(t1, t2) : bool = ∃(s:Substitution) : t2 =
subst(s)(t1)
```

The type system for FUNC terms is a straightforward extension of simple Curry typing. We make use of the possibility in PVS to define *inductive predicates*. The expression typableE(b)(e,t) should be read as "In the context of a base b, the expression e has type t".

```
BASE : TYPE = [X → TYPES] typableE(b:BASE)(e:FUNC, tr:TYPES) :
INDUCTIVE bool =
    CASES e OF
        vari(v)     : b(v) = tr,
        appl(f, a)  : ∃(ta:TYPES): typableE(b)(f,t_arr(ta,tr)) ∧
                                   typableE(b)(a,ta),
        lamb(v,e)   : t_arr?(tr) ∧
                       typableE(b WITH [v := t_arg(tr)])(e,t_res(tr)),
        letr (v,d,e): ∃(ft:TYPES): LET nb = b WITH [v := ft] IN
                          typableE(nb)(d,ft) ∧ typableE(nb)(e,tr)
    ENDCASES
```

Observe that the type system is *monomorphic*: in the term letr (v,d,e), all occurrences of v in e should have identical types. In Sect. 5.3 we describe how to extend the system with *polymorphic* letr constructs.

For typing COMB expressions, we assume that combinator symbols are supplied with a type (or actually a type scheme) by a so-called *type environment*. The type system is defined as a PVS theory parameterized with that environment.

```
typingCOMB [V, X, C:TYPE, % type and term variables, and combinator symbols
    (IMPORTING TYPES[V]) env:[C → TYPES[V]]] : THEORY % type environment
BEGIN
  typableC(b:BASE)(e:COMB, t:TYPES) : INDUCTIVE bool =
      CASES e OF
          c_var(w)     : b(w) = t,
          c_const(c)   : env(c) ≤ t,
        c_appl(f, a) : ∃(t1:TYPES):
                           typableC(b)(f,t_arr(t1,t)) ∧ typableC(b)(a,t1)
     ENDCASES
END typingCOMB
```

We can now formulate our main theorem that relates the typing of FUNC to the typing of COMB.

```
type_preserving : THEOREM   ∀(e:FUNC,b:BASE,t:TYPES):
                              typableE(b)(e,t) ⇔ typableC(b)(lam2Ski(e),t)
```

In order to be able to prove this theorem, it is necessary to have a concrete type environment for SKI. We simply choose natural numbers as names for the type variables appearing in the type schemes .

```
 alp : TYPES = t_var(0)
 bet : TYPES = t_var(1)
 gam : TYPES = t_var(2)
 del : TYPES = t_var(3)

 TArr2(t1,t2,t3: TYPES)        : TYPES = t_arr(t1,t_arr(t2,t3))
 TArr3(t1,t2,t3,t4: TYPES)     : TYPES = t_arr(t1,t_TArr2(t2,t3,t4))
 TArr4(t1,t2,t3,t4,t5: TYPES)  : TYPES = t_arr(t1,t_TArr3(t2,t3,t4,t5))

 env_ski(c:SKI): TYPES =
     CASES c OF
         I : t_arr(alp,alp),
         K : TArr2(alp,bet,alp),
         S : TArr3(t_arr(alp,t_arr(bet,gam)),t_arr(alp,bet),gam,bet),
         Y : t_arr(t_arr(alp,alp),alp),
         B : TArr3(t_arr(alp,bet),t_arr(gam,alp),gam,bet),
         C : TArr3(t_arr(alp,t_arr(bet,gam)),bet,alp,gam),
         S1: TArr4(t_arr(alp,t_arr(bet,gam)),t_arr(del,alp),
                 t_arr(del,bet),del,gam),
         B1: TArr4(t_arr(alp,t_arr(bet,gam)),alp,t_arr(del,bet),del,gam),
         C1: TArr4(t_arr(alp,t_arr(bet,gam)),t_arr(del,alp),bet,del,gam)
     ENDCASES
```

It is not difficult to show that each combinator written as a lambda expression is typable according to the typableE predicate, using an arbitrary base and the type given by env_ski. For example, the following properties can be proven in merely a few steps.

```
kterm: FUNC = lamb(0,lamb(1,vari(0))) yterm: FUNC = lamb(0, letr
(1,appl(vari(0),vari(1)))),vari(1)) K_typable: LEMMA
∀(bas:Base): typableE(bas)(kterm,env_ski(K)) Y_typable: LEMMA
∀(bas:Base): typableE(bas)(yterm,env_ski(Y))
```

The proof of our main theorem takes more effort. It requires the following property concerning the typing of abstractions.

```
type_abstr : LEMMA      ∀(e:COMB,b:BASE,ta,tr:TYPES,v:V):
    typableC(b WITH [v:=ta])(e,tr) ⇔ typableC(b)(abstr(e,v),t_arr(ta,tr))
```

This lemma is proven by course-of-values induction on the size of `e`. The proof itself is actually not difficult, but its size is quite extensive. It requires approximately 500 proof steps, as can be seen in the PVS proof files. This example also clearly shows that it is almost impossible to perform such a proof without the assistance of a theorem prover.

5.3 A Polymorphic Type System

In order to express `letr` polymorphism we need quantified types, also known as *type schemes*. These type schemes are of the form $\forall\alpha_1,\ldots,\alpha_n : \sigma$ in which the α_i are type variables and σ is a type. Instead of formalizing a type scheme as a pair consisting of a set of type variables and a type, we use a more explicit representation as for instance can be found in Naraschewski and Nipkow [18]. As such, the type scheme is formalized as an algebraic data type containing separate constructors for free and bound variables.

```
SCHEME[V : TYPE]: DATATYPE BEGIN
    ts_bv  (bv:V): ts_bv?
    ts_fv  (fv:V): ts_fv?
    ts_arr (arg, res:SCHEME): ts_arr?
END SCHEME

discard(x:V): PRED[V] = ∅

bvs:[SCHEME → PRED[V]] = reduce(singleton,discard,∪)
fvs:[SCHEME → PRED[V]] = reduce(discard,singleton,∪)
```

The predicates `bvs` and `fvs` determine the set of bound and free type variables for a given scheme, respectively. We will use several conversions between types and type schemes. A type can be obtained from a scheme by means of *instantiation*. This operation replaces the bound variables of a scheme with types and leaves the free variables unaltered.

```
inst(s:Substitution): [SCHEME → TYPES] =
reduce(s,λ(v:V):t_var(v),t_arr)
```

A type can be converted into a scheme via *generalization*. The result of a generalization step depends on the context in which this operation is performed, in particular on the type variables appearing in the used base: only type variables not appearing free in a base can be universally quantified. Observe that in the polymorphic case a base associates term variables with schemes rather than with types.

```
BASE : TYPE = [X → SCHEME]

fvs(b:BASE): PRED[V] = { v:V | ∃(x:X) : fvs(b(x))(v) }
gen(b:BASE): [TYPES → SCHEME] =
        reduce(λ(v:V):IF fvs(b)(v)THEN ts_fv(v) ELSE ts_bv(v) ENDIF,ts_arr)
```

The adjustment of the `typableE` predicate leads to a type system almost equivalent to the system presented by Naraschewski and Nipkow [18]. There is only one small difference: in the term `letr(v,b,e)` we distinguish between two cases depending on whether or not v occurs in e. If v is present in e then the whole term is treated as usual. If not, the subterm b is ignored. It doesn't matter whether b is typable or not if it is not used in e. We will explain the reason for this refinement later. The test for the presence of v in e is done via `fvs`.

```
fvs:[FUNC → PRED[V]] = reduce(singleton, ∪, remove,
           λ(v:V,b,e:PRED[V]):IF e(v) THEN remove(v,∪(b,e)) ELSE e ENDIF)

type2Scheme: [TYPES  → SCHEME] = reduce(ts_fv,ts_arr)

typableE(b:BASE)(e:FUNC, tr:TYPES) : INDUCTIVE bool =
    CASES e OF
        vari(v)    : ∃(s:Substitution): inst(s)(b(v)) = tr,
        appl(f, a) : ∃(ta:TYPES): typableE(b)(f,t_arr(ta,tr)) ∧
                                      typableE(b)(a,ta),
        lamb(v,e)  : t_arr?(tr) ∧ typableE(b
                        WITH [v := type2Scheme(t_arg(tr))])(e,t_res(tr)),
        letr(v,d,e): IF fvs(e)(v)
                     THEN ∃(t:TYPES) :
                          typableE(b WITH [v := type2Scheme(t)])(d,t) ∧
                          typableE(b WITH [v := gen(b)(t)])(e,tr)
                     ELSE typableE(b)(e,tr)
                     ENDIF
    ENDCASES
```

The operation `type2Scheme` converts a type into a fully monomorphic scheme, i.e., a scheme with no bound variables.

The question remains of how to change the type system for combinators such that it can handle multiple occurrences of a recursive function introduced by a `letr`. Normally, (the combinator version of) this function is distributed over the corresponding expression via the S combinator. However, the polymorphic usage of arguments requires polymorphism of a higher rank. Observe that the schemes we have introduced are essentially of rank 1, as well as the types provided by the type environment, which is used for assigned types to combinators. Instead of allowing universal quantifiers at arbitrary levels, we adjust the transformation rule for `letr` expressions in the following way:

```
lam2Ski(e:FUNC) : RECURSIVE COMB =
    CASES e OF
        vari(v)    : c_var(v),
        appl(f, a) : c_appl(lam2Ski(f),lam2Ski(a)),
        lamb(v,e)  : abstr(lam2Ski(e),v),
        letr(v,d,e): subst(single(v,c_appl(c_const(Y),
                          abstr(lam2Ski(d),v)))) (lam2Ski(e))
    ENDCASES
MEASURE e BY <<

single(v:V,e:COMB) : [V → COMB] =
    λ(w:V): IF v = w THEN e ELSE c_var(v) ENDIF
```

```
subst(s:[V → COMB]) : [COMB → COMB] = reduce(s,c_const,c_appl)
```

Here `single` and `subst` are substitutions on combinator terms. The disadvantage of the transformation is of course, that if the function is used more than once, it will be duplicated.

If the function is not used at all, it will disappear due to the substitution. For this reason we made the case distinction in `typableE` for the `letr` construct. In this way we are able to preserve typability in all cases and are not obliged to make an exception for cases that probably rarely occur.

The only part of the type system for combinators, `typableC`, that needs to be adjusted is the rule for variables. Instead of using the base type of the variable, we now allow instantiation of the scheme provided by the base. Since this adjustment is self-evident we do not show it here.

The main goal of this section is to prove the following theorem again:

```
type_preserving : THEOREM   ∀(e:FUNC,b:BASE,t:TYPES):
                            typableE(b)(e,t) ⇔ typableC(b)(lam2Ski(e),t)
```

At first sight, the extension of our system with type schemes seems to have low impact. However, as already has been noticed by Naraschewski and Nipkow [18], reasoning about type schemes is much more subtle than reasoning about (monomorphic) types. The proof of the theorem depends on the following two properties concerning term substitutions:

```
type_subst1 : LEMMA    ∀(e1,e2:COMB,b:BASE,t1,t2:TYPES, v:V):
        typableC(b)(e1,t1) ∧ typableC(b WITH [v := gen(b)(t1)])(e2, t2)
            ⇒ typableC(b)(subst(single(v,e1))(e2),t2)

type_subst2 : CONJECTURE
∀(e1,e2:COMB,b:BASE,t2:TYPES,v:(fvs(e2))):
        typableC(b)(subst(single(v,e1))(e2),t2) ⇒
            ∃(t1:TYPES): typableC(b)(e1,t1) ∧
                         typableC(b WITH [v := gen(b)(t1)])(e2, t2)
```

The first one is used to prove the ⇒ part of the theorem; the second one to prove the ⇐ part. Momentarily we have finished the proof of `type_subst1`. This proof is performed by induction on the structure of the term `e2`, see the proof files. Currently we are working on a full formal proof of `type_subst2`.

6 Related Work

In his book, Hindley [19] writes about *strong type-invariance* and shows that certain sets of combinators form a *typable basis*. He informally proves that a set of lambda expressions imitating combinators preserves typability. This is done by construction, using small variations on common bracket abstraction algorithms. This approach is not unlike ours. However, we use a larger set, a more complex algorithm, and Hindley's proof is neither formal nor are all the details shown.

The type inference algorithm $\mathcal{W}$ by Damas and Milner [20] has been formally and mechanically proven by Nazareth and Nipkow [21] (monomorphic), Naraschewski and Nipkow [18] (polymorphic), bot;h using Isabelle/HOL, and by Dubois and Ménissier-Morain [22] (polymorphic) using Coq. Although we prove equivalence of type inference before and after bracket abstraction in PVS instead of type inference itself, the process of proving has a lot in common. Everybody runs into issues with alpha conversion. Furthermore, using a proof assistant/checker forces one to formalize everything explicitly and prove every minute detail, where one may adversely use hand-waving to sweep it under the rug in an informal proof.

Hindley [23] and, almost simultaneously, Curry [24] showed that one can derive a principal type inference algorithm on a system of combinators, using reduction on type combinators. This already shows that one can do type inference using only application, since the combinators can be imitated using lambda calculus. Their (informal) proof is based on the S and K combinator, while we needed proof that it works for algorithm-C and that it infers the exact same types as functional languages usually do (using variants of algorithm $\mathcal{W}$).

7 Conclusions

We have shown how to specify type derivation and complex bracket abstraction in a rigorously formal way using PVS, at least for the monomorphic case. This enabled us to formally prove that Diller–algorithm–C without η-conversion preserves typability. Our approach to derive types after bracket abstraction in a type checking command-line shell, requires this property. This proof also confirms that we were correct to assume that a type checker/inferrer can really be constructed using merely Dynamics of the functional programming language Clean. We now conclude that the seemingly limited interface of Dynamics is powerful enough for type inference such as done by our shell.

For future aspirations, it rests to complete the component pertaining to the $\Leftarrow$ part of the proof for the polymorphic case.

References

1. Haskell B. Curry and Robert Feys. *Combinatory Logic.* North Holland, 1958.
2. M. Schönfinkel. Über die Bausteine der mathematischen Logik. In *Mathematische Annalen*, volume 92, pages 305–316. 1924.
3. David A. Turner. *SASL language manual.* St. Andrews University, Scotland, 1976.
4. S. Kamal Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41:222–224, 1976.
5. David A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, June 1979.
6. Antoni Diller. *Compiling functional languages.* John Wiley & Sons, 1988.
7. Martin W. Bunder. Some improvements to Turner's algorithm for bracket abstraction. *Journal of Symbolic Logic*, 55(2):656–669, June 1990.
8. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, 1987.

9. Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report (version 2.1)*. Radboud University Nijmegen, November 2002. The Clean website: `http://www.cs.ru.nl/∼clean/`.
10. Rinus Plasmeijer and Arjen van Weelden. A functional shell that operates on typed and compiled applications. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming, 5th International Summer School, AFP 2004, University of Tartu, Revised Lectures*, volume 3622 of *LNCS*, pages 245–272, Tartu, Estonia, August 2004. Springer.
11. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.
12. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. The Haskell website: `http://www.haskell.org/`.
13. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.
14. Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Antony J. T. Davie, and Chris Clack, editors, *IFL '98: Selected Papers from the 10th International Workshop on Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 169–185. Springer, 1999.
15. Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Pena and Thomas Arts, editors, *IFL '02: Selected Papers from the 14th International Workshop on Implementation of Functional Languages*, volume 2670 of *LNCS*, pages 101–117. Springer, 2003.
16. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. London Mathematical Society Student Texts. Cambridge University Press, 1986.
17. Michael S. Joy, Vic J. Rayward-Smith, and F. Warren Burton. Efficient combinator code. *Journal of Computer Languages*, 10(3/4):211–224, 1985.
18. Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm $\mathcal{W}$ in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.
19. J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge tracts in theoretical computer science*. Camebridge University Press, 1997.
20. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.
21. Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm $\mathcal{W}$: The monomorphic case. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 331–346. Springer, 1996.
22. Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas–Milner within Coq. *Journal of Automated Reasoning*, 23:319–346, 1999.
23. J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transaction of the American Mathematical Society*, 146:29–60, December 1969.
24. Haskell B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.