Improving Automation in Interactive Theorem Provers by Efficient Encoding of Lambda-Abstractions

Łukasz Czajka *
University of Innsbruck, Austria
Lukasz.Czajka@uibk.ac.at

NOTE. This is a corrected version of a paper that appeared at CPP 2016. Because of an implementation error in one of the axioms added by combinator translations, a certain number of generated FOF problems was actually inconsistent. This did not visibly affect our experimental evaluation on any of the provers except for the E theorem prover whose results were significantly affected. Consequently, the cumulative results were also significantly affected and the cumulative success rate of combinatory abstraction algorithms was inflated. The corrected experimental results still show that our best algorithm D significantly outperforms the standard Schönfinkel's combinatory abstraction algorithm S, but it no longer ourperforms (improved) lambda-lifting. In fact, the algorithm D and lambda-lifting perform similarly. With the provers CVC4, Z3 and SPASS the algorithm D outperforms lambda-lifting, while lambda-lifting outperforms algorithm D with the provers E and Vampire. In this corrected version the tables with the results and some of the discussion of the experimental evaluation were updated. The author thanks Thibault Gauthier for pointing out the mistake.

Abstract

Hammers are tools for employing external automated theorem provers (ATPs) to improve automation in formal proof assistants. Strong automation can greatly ease the task of developing formal proofs. An essential component of any hammer is the translation of the logic of a proof assistant to the format of an ATP. Formalisms of state-of-the-art ATPs are usually first-order, so some method for eliminating lambda-abstractions is needed. We present an experimental comparison of several combinatory abstraction algorithms for HOL(y)Hammer – a hammer for HOL Light. The algorithms are compared on problems involving non-trivial lambda-abstractions selected from the HOL Light core library and a library for multivariate analysis. We succeeded in developing algorithms which outperform both lambda-lifting and the simple Schönfinkel's algorithm used in Sledgehammer for Isabelle/HOL. This increases the ATPs' success rate on translated problems, thus enhancing automation in proof assistants.

* Supported by FWF grant P26201

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical Theorem Proving

Keywords proof assistants, automation, hammers, bracket abstraction, lambda-lifting

1. Introduction

Hammers are tools for employing external automated theorem provers (ATPs) to improve automation in formal proof assistants. A strong hammer may save a tremendous amount of human labour in developing formal mathematics. The state-of-the-art hammers manage to fully automatically prove around 40% of theorems in the libraries of Mizar or HOL Light/Flyspeck, and a comparable amount for HOL4 [12] and Isabelle/HOL [5, 15]. An essential part of a hammer is a translation of the logic of an interactive theorem prover (ITP) to a format handled by automatic theorem provers (ATPs). Typically, ATPs are based on untyped first-order logic, while ITPs use higher-order formalisms. Therefore, a translation of an ITP formalism needs to involve a translation of higher-order features into first-order logic, in particular the elimination of lambda-abstractions. Two commonly used methods are lambda-lifting [19] and combinatory abstraction algorithms (also called bracket abstraction algorithms). Combinatory abstraction algorithms translate lambda-expressions to first-order applicative expressions built from *combinators* – special constants with associated axioms allowing to represent any function respresentable using lambda-abstractions. Sledgehammer [29, 31] for Isabelle/HOL implements both lambda-lifting and a simple Schönfinkel's combinatory abstraction algorithm. These methods have been evaluated and compared in [30], but the results were inconclusive. Schönfinkel's algorithm was also used earlier by Hurd [20] in a translation of HOL terms to first-order logic. In [4, Section 6.7.3] a hybrid translation scheme combining combinators and lambda-lifting was evaluated, but the results were inconclusive.

There has been much research on combinatory abstraction algorithms, but it has been mostly focused on designing algorithms suitable for use in the implementation of functional programming languages. Automatically proving problems originating from ITPs is very different from the implementation of functional programs, which is entirely concerned with the reduction of huge lambda-expressions, the target code size and full laziness. In automated theorem proving, besides the size of the translation, issues like the number of introduced combinator equations, the "complexity" of the combinator theory, or the blow-up of the search space also play a significant role.

In this paper we present an evaluation of two versions of lambda-lifting and five combinatory abstraction algorithms which we implemented in HOL(y)Hammer [24, 25] – a hammer tool for HOL Light and HOL4. The evaluation was performed on

those problems from the HOL Light core library and a library for multivariate analysis [16] which contained non-trivial lambda-abstractions in the goal or the dependencies. By non-trivial we mean those lambda-abstractions which were not eliminated by a simple preprocessing described in Section 3. We measured the number of problems various ATPs could reprove from their dependencies within a 30 s time limit. We found that lambda-lifting outperforms the Schönfinkel's combinatory abstraction algorithm with some ATPs, while it performs worse with others. Basing on the algorithm of Abdali [2] and the idea of director strings [26, 27] we managed to develop two algorithms which on average outperform both lambda-lifting and the combinatory abstraction algorithms of Schönfinkel and Turner. These algorithms are new in that, to the author's knowledge, they have not been published before, but they are a combination and adaptation of some ideas already present in the literature. The two new algorithms have similar performance to lambda-lifting for the provers with which lambda-lifting performs well, and similar performance to the abstraction algorithms of Schönfinkel and Turner for the other provers. By integrating these algorithms into HOL(y)Hammer we thus improve automation for the HOL Light and HOL4 proof assistants.

2. Combinatory Abstraction Algorithms

In this section we present a high-level mathematical description of the combinatory abstraction algorithms which we implemented together with some of their theoretical properties. A more detailed description of some choices made in the implementation of the algorithms and how the algorithms were integrated into HOL(y)Hammer is presented in Section 3. The results of the experimental evaluation of the performance of the algorithms are presented in Section 4. We assume basic familiarity with the lambda-calculus [3]. We consider lambda-terms up to α-equivalence and we use the variable convention.

First, we fix some notation and terminology. By Λ we denote the set of all (untyped) lambda-terms, by Λ_0 the set of all closed lambda-terms, and by V the set of variables. Closed lambda-terms are also called *combinators*. Given a set of combinators $B \subseteq \Lambda_0$, by $\mathrm{CL}(B)$ we denote the set of all lambda-terms built from variables and elements of B using only application. A *univariate combinatory abstraction algorithm* A for a basis $B \subseteq \Lambda_0$ is an algorithm computing a function[1] $A : V \times \mathrm{CL}(B) \to \mathrm{CL}(B)$ such that for any $x \in V$ and $t \in \mathrm{CL}(B)$ we have $\mathrm{FV}(A(x,t)) = \mathrm{FV}(t) \setminus \{x\}$ and $A(x,t) =_{\beta\eta} \lambda x.t$. We usually write $[x]_A.t$ instead of $A(x,t)$. For a univariate algorithm A the *induced translation* $\mathrm{H}_A : \Lambda \to \mathrm{CL}(B)$ is defined recursively by

$$\begin{array}{rcl} \mathrm{H}_A(x) & = & x \\ \mathrm{H}_A(st) & = & (\mathrm{H}_A(s))(\mathrm{H}_A(t)) \\ \mathrm{H}_A(\lambda x.t) & = & [x]_A.\mathrm{H}_A(t) \end{array}$$

It follows by straightforward induction that $\mathrm{FV}(\mathrm{H}_A(t)) = \mathrm{FV}(t)$ and $\mathrm{H}_A(t) =_{\beta\eta} t$.

A *multivariate combinatory abstraction algorithm* A for a basis $B \subseteq \Lambda_0$ is a family of algorithms $A_n : V^n \times \Lambda \to \mathrm{CL}(B)$ with $n \in \mathbb{N}$, satisfying the following for any $t \in \Lambda$, $x_1, \ldots, x_n, y \in V$, $n \in \mathbb{N}$:

- $A_n(x_1, \ldots, x_n, t) =_{\beta\eta} \lambda x_1 \ldots x_n.t$,
- $\mathrm{FV}(A_n(x_1, \ldots, x_n, t)) = \mathrm{FV}(t) \setminus \{x_1, \ldots, x_n\}$,
- $A_n(x_1, \ldots, x_n, \lambda y.t) = A_{n+1}(x_1, \ldots, x_n, y, t)$.

We usually write $[x_1, \ldots, x_n]_A.t$ instead of $A_n(x_1, \ldots, x_n, t)$. We denote A_0 by H_A and call it the *translation induced by* A.

[1] Whenever convenient we confuse algorithms with the functions they compute.

Abstraction algorithms are usually presented by a list of recursive equations with side conditions. It is to be understood that the first applicable equation in the list is to be used.

For instance, the (univariate) algorithm (fab) of Curry [10, §6A] for the basis $\{\mathsf{S}, \mathsf{K}, \mathsf{I}\}$ where

$$\begin{array}{rcl} \mathsf{S} & = & \lambda xyz.xz(yz) \\ \mathsf{K} & = & \lambda xy.x \\ \mathsf{I} & = & \lambda x.x \end{array}$$

may be defined by the list of equations

$$\begin{array}{rcl} [x]_{(\mathrm{fab})}.st & = & \mathsf{S}([x]_{(\mathrm{fab})}.s)([x]_{(\mathrm{fab})}.t) \\ [x]_{(\mathrm{fab})}.x & = & \mathsf{I} \\ [x]_{(\mathrm{fab})}.t & = & \mathsf{K}t \end{array}$$

The last equation is thus used only when the previous two cannot be applied. For example $[x]_{(\mathrm{fab})}.yyx = \mathsf{S}(\mathsf{S}(\mathsf{K}y)(\mathsf{K}y))\mathsf{I}$. Note that we have $[x]_{(\mathrm{fab})}.\mathsf{S} = \mathsf{KS}$, $[x]_{(\mathrm{fab})}.\mathsf{K} = \mathsf{KK}$ and $[x]_{(\mathrm{fab})}.\mathsf{I} = \mathsf{KI}$, but $[x]_{(\mathrm{fab})}.\lambda x.t$ is undefined if $\lambda x.t \notin \{\mathsf{S}, \mathsf{K}, \mathsf{I}\}$, because then $\lambda x.t \notin \mathrm{CL}(\{\mathsf{S}, \mathsf{K}, \mathsf{I}\})$. One easily shows by induction on the structure of $t \in \mathrm{CL}(\{\mathsf{S}, \mathsf{K}, \mathsf{I}\})$ that indeed $\mathrm{FV}([x]_{(\mathrm{fab})}.t) = \mathrm{FV}(t) \setminus \{x\}$ and $[x]_{(\mathrm{fab})}.t \to_\beta^* t$, so (fab) is an abstraction algorithm. For all algorithms which we present, their correctness, i.e., that they are abstraction algorithms, follows by straightforward induction, and thus we will avoid mentioning this explicitly every time.

The following algorithm is called (abf) in [10, §6A].

$$\begin{array}{rcll} [x]_{(\mathrm{abf})}.x & = & \mathsf{I} & \\ [x]_{(\mathrm{abf})}.t & = & \mathsf{K}t & \text{if } x \notin \mathrm{FV}(t) \\ [x]_{(\mathrm{abf})}.st & = & \mathsf{S}([x]_{(\mathrm{abf})}.s)([x]_{(\mathrm{abf})}.t) & \end{array}$$

The algorithms (fab) and (abf) are perhaps the most widely known and also the simplest combinatory abstraction algorithms, but they are not particularly efficient.

One measure of the efficiency of an abstraction algorithm A is the *translation size* – the size of $\mathrm{H}_A(t)$ as a function of the size of t. For (fab) the translation size may be exponential, while for (abf) it is $O(n^3)$. See [22, 23] for an analysis of the translation size for various combinatory abstraction algorithms. For a fixed finite basis $\Omega(n \log n)$ is a lower bound on the translation size [22, 23]. This bound is attained in [26] (see also [9] and [23, Section 4]). However, for our purposes we chose certain algorithms for infinite bases and we did not consider the asymptotically optimal ones for finite bases. We present below the algorithms we evaluated.

2.1 Schönfinkel's Algorithm

The basis for Schönfinkel's combinatory abstraction algorithm S is $\{\mathsf{S}, \mathsf{K}, \mathsf{I}, \mathsf{B}, \mathsf{C}\}$ where

$$\begin{array}{rcl} \mathsf{B} & = & \lambda xyz.x(yz) \\ \mathsf{C} & = & \lambda xyz.xzy \end{array}$$

The algorithm S is defined by the equations in Figure 1. Like for (abf), the translation size for S is also $O(n^3)$ but with a smaller constant [23]. The algorithm S is called (abcdef) in [10, §6A]. It is actually the Schönfinkel's algorithm implicit in [33]. The algorithm S is the one implemented by Meng and Paulson [30] and used in Sledgehammer for Isabelle/HOL.

In [30] there seems to be an erroneous claim that the translation size for S is quadratic. This is disproved by the following counterexample: $E_n = \lambda x_1 \ldots x_n.x_n \ldots x_1$. Successive intermediate terms appearing during the translation of E_n look as in Figure 2, where $\mathsf{B}^k\mathsf{C}$ denotes k-time application of B to C, e.g., $\mathsf{B}^3\mathsf{C} = \mathsf{B}(\mathsf{B}(\mathsf{BC}))$. In general

$$\begin{array}{rcl} \mathrm{H}_S(E_1) & = & \mathsf{I} \\ \mathrm{H}_S(E_{n+2}) & = & \Phi_n(\Phi_{n-1}(\Phi_{n-2}(\ldots(\Phi_0(\mathsf{I}))\ldots))) \end{array}$$

$$
\begin{array}{rcll}
[x]_S.t & = & \mathsf{K}t & \text{if } x \notin \mathrm{FV}(t) \\
[x]_S.x & = & \mathsf{I} & \\
[x]_S.sx & = & s & \text{if } x \notin \mathrm{FV}(s) \\
[x]_S.st & = & \mathsf{B}s([x]_S.t) & \text{if } x \notin \mathrm{FV}(s) \\
[x]_S.st & = & \mathsf{C}([x]_S.s)t & \text{if } x \notin \mathrm{FV}(t) \\
[x]_S.st & = & \mathsf{S}([x]_S.s)([x]_S.t) &
\end{array}
$$

Figure 1. Schönfinkel's algorithm S

$$
\begin{array}{rcll}
[x]_T.t & = & \mathsf{K}t & \text{if } x \notin \mathrm{FV}(t) \\
[x]_T.x & = & \mathsf{I} & \\
[x]_T.sx & = & s & \text{if } x \notin \mathrm{FV}(s) \\
[x]_T.uxt & = & \mathsf{C}ut & \text{if } x \notin \mathrm{FV}(ut) \\
[x]_T.uxt & = & \mathsf{S}u([x]_T.t) & \text{if } x \notin \mathrm{FV}(u) \\
[x]_T.ust & = & \mathsf{B}'us([x]_T.t) & \text{if } x \notin \mathrm{FV}(us) \\
[x]_T.ust & = & \mathsf{C}'u([x]_T.s)t & \text{if } x \notin \mathrm{FV}(ut) \\
[x]_T.ust & = & \mathsf{S}'u([x]_T.s)([x]_T.t) & \text{if } x \notin \mathrm{FV}(u) \\
[x]_T.st & = & \mathsf{B}s([x]_T.t) & \text{if } x \notin \mathrm{FV}(s) \\
[x]_T.st & = & \mathsf{C}([x]_T.s)t & \text{if } x \notin \mathrm{FV}(t) \\
[x]_T.st & = & \mathsf{S}([x]_T.s)([x]_T.t) &
\end{array}
$$

Figure 3. Turner's algorithm T

where

$$\Phi_k(X) = \mathsf{C}(\mathsf{BC}(\mathsf{B}^2\mathsf{C}(\mathsf{B}^3\mathsf{C}(\ldots(\mathsf{B}^k\mathsf{C}X)\ldots))))$$

The above may be shown by induction on n, noting that

$$
\begin{array}{c}
[x_1, x_2, \ldots, x_n]_S.x_n \ldots x_2 x_1 = \\
[x_1]_S.\Phi_{n-2}([x_2, \ldots, x_n]_S.x_n \ldots x_2)x_1 = \\
\Phi_{n-2}([x_2, \ldots, x_n]_S.x_n \ldots x_2)
\end{array}
$$

The size of the translation of E_n is thus cubic in n. For a detailed analysis see e.g. [22, Section 2.6].

2.2 Turner's Algorithm

Turner's algorithm [35] is perhaps the most widely known improvement on Schönfinkel's algorithm. The basis for Turner's algorithm is $\{\mathsf{S}, \mathsf{K}, \mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{S}', \mathsf{B}', \mathsf{C}'\}$ where

$$
\begin{array}{rcl}
\mathsf{S}' & = & \lambda kxyz.k(xz)(yz) \\
\mathsf{B}' & = & \lambda kxyz.kx(yz) \\
\mathsf{C}' & = & \lambda kxyz.k(xz)y
\end{array}
$$

Turner's algorithm T is defined by the equations in Figure 3. The translation size of T is worst-case $O(n^2)$ [23] and average-case $O(n^{3/2})$ [17]. Actually, in [35] a different formulation of T is presented by means of "optimisation rules". There is some ambiguity in Turner's original definition, but whatever the interpretation Turner's formulation is *not* equivalent to the algorithm T as presented above. However, the differences are minor. In fact, the algorithm T is equivalent to (a certain interpretation of) Turner's original algorithm as far translating terms in β-normal form is concerned. See [7, 11].

The idea with Turner's combinators $\mathsf{S}', \mathsf{B}', \mathsf{C}'$ is that they allow to leave the structure of the translation of an application st unaltered in the form $\kappa s't'$ where κ is a "tag" composed entirely of combinators. For instance, if $x_1, x_2, x_3 \in \mathrm{FV}(s) \cap \mathrm{FV}(t)$ (and s has a form such that the equations 3-5 in the definition of T are not used) then

$$[x_1, x_2, x_3]_T.st = \mathsf{S}'(\mathsf{S}'\mathsf{S})([x_1, x_2, x_3]_T.s)([x_1, x_2, x_3]_T.s)$$

while

$$
\begin{array}{rcl}
[x_1, x_2, x_3]_S.st & = & \mathsf{S}(\mathsf{BS}(\mathsf{B}(\mathsf{BS})([x_1, x_2, x_3]_S.s))) \\
 & & ([x_1, x_2, x_3]_S.s).
\end{array}
$$

In [21, Chapter 16] it is claimed that using the combinator B^* defined by

$$\mathsf{B}^* = \lambda fxyz.f(x(yz))$$

instead of B' improves Turner's algorithm. We implemented an algorithm we call modified Turner's algorithm T^*, which is defined like algorithm T except that the equation with B' is removed, the following equation added after the third one

$$[x]_{T^*}.st = \mathsf{B}^*st_1t_2 \quad \text{if } x \notin \mathrm{FV}(s) \text{ and } [x]_{T^*}.t = \mathsf{B}t_1t_2$$

and the equation with B is moved after the added one. A variant of T^* was evaluated in [30]. In theory, their formulation is not in general equivalent to ours, or for that matter to the formulation in [21], but the differences are again very minor. See [11].

The experiments described in Section 4 indicate that the difference between the performance of T and T^* is very small, with T^* performing slightly better.

2.3 Lambda-Lifting

Lambda-lifting may be seen as a multivariate combinatory abstraction algorithm for the basis consisting of all supercombinators [19]. A *supercombinator* is a closed lambda-term of the form $\lambda x_1 \ldots x_n.t$ where $n \geq 0$, the term t is not a lambda-abstraction, and each lambda-abstraction appearing in t is a supercombinator. The lambda-lifting algorithm L is defined by the equations in Figure 4, where $m \geq 0$. The advantage of lambda-lifting is that "evaluating" an application of a translated lambda-abstraction to an argument requires just a single substitution. The disadvantages are that a new combinator is used for every lambda-abstraction, that in contrast to the algorithms S and T translations of different lambda-abstractions do not share much "structure", and that the translations are less "complete" in the sense that there is less possibility of the ATPs synthesing new functions not already present in the input.

The algorithms A and D presented in the following sections are in a sense a compromise between lambda-lifting and the algorithms S and T. On the one hand, they produce relatively small encodings and "evaluating" an application of a translated lambda-abstraction to some arguments requires relatively few steps, but more than with lambda-lifting. On the other hand, the algorithms A and D are more complete and the translations of different lambda-abstractions still share some structure, but less so than with the algorithms S and T. The algorithms D, A and $L2$ on average performed best in our experiments.

2.4 Abdali's Algorithm

In [2] Abdali presented a multivariate combinatory abstraction algorithm for an infinite basis. We implemented a certain algorithm similar to the algorithm in [2]. We call our algorithm Abdali's algorithm A in deference to the author of the original idea. The basis for A is $\{\mathsf{B}_n^m, \mathsf{E}_n^m, \mathsf{S}_n^{k,m} \mid n, m, k \in \mathbb{N}\}$ with the combinators defined in Figure 5. The multivariate algorithm A is defined by the equations in Figure 6, where $m > 0$.

For example

$$
\begin{array}{rcl}
[x, y]_A.x(yz)(zxy)z & = & \mathsf{S}_3^{0,1}(\mathsf{B}_0^1(\mathsf{S}_1^{0,0}(\mathsf{B}_0^1z)))z(\mathsf{B}_0^2z) \\
[x, y]_A.x(yz)(zyx)z & = & \mathsf{S}_3^{0,1}(\mathsf{B}_0^1(\mathsf{S}_1^{0,0}(\mathsf{B}_0^1z))) \\
 & & (\mathsf{B}_2^2\mathsf{S}_0^{0,1}\mathsf{S}_0^{1,0})(\mathsf{B}_0^2z) \\
[x, y, z]_A.z(yz)(xy) & = & \mathsf{S}_2^{2,0}\mathsf{S}_0^{1,0}(\mathsf{S}_1^{0,2}\mathsf{S}_0^{1,1})
\end{array}
$$

The idea with algorithm A is to "encode" more structure of a lambda-term in the combinators, thus decreasing the number of

$$\lambda x_1 \ldots x_{n-1}.\mathsf{C}(\mathsf{C}(\mathsf{C}(\ldots(\mathsf{CI}x_{n-1})\ldots)x_3)x_2)x_1$$
$$\lambda x_1 \ldots x_{n-2}.\mathsf{C}(\mathsf{BC}(\mathsf{C}(\mathsf{BC}(\mathsf{C}(\mathsf{BC}(\ldots(\mathsf{C}(\mathsf{BC}(\mathsf{CI}))x_{n-2})\ldots))x_3))x_2))x_1$$
$$\lambda x_1 \ldots x_{n-3}.\mathsf{C}(\mathsf{BC}(\mathsf{B}^2\mathsf{C}(\mathsf{C}(\mathsf{BC}(\mathsf{B}^2\mathsf{C}(\mathsf{C}(\mathsf{BC}(\mathsf{B}^2\mathsf{C}(\ldots(\mathsf{C}(\mathsf{BC}(\mathsf{B}^2\mathsf{C}(\mathsf{C}(\mathsf{BC}(\mathsf{CI}))))))x_{n-3})\ldots)))x_3)))x_2)))x_1$$
$$\ldots$$

Figure 2. Successive intermediate steps of Schönfinkel's algorithm S on the term E_n

$$\begin{array}{rcl}
\mathrm{H}_L(x) & = & x \\
\mathrm{H}_L(t_1 t_2) & = & (\mathrm{H}_L(t_1))(\mathrm{H}_L(t_2)) \\
[x_1, \ldots, x_m]_L.\lambda y.t & = & [x_1, \ldots, x_m, y]_L.t \\
[x_1, \ldots, x_m]_L.t & = & (\lambda y_1 \ldots y_n x_1 \ldots x_m.\mathrm{H}_L(t))y_1 \ldots y_n \text{ where } \{y_1, \ldots, y_n\} = \mathrm{FV}(t) \setminus \{x_1, \ldots, x_m\}
\end{array}$$

Figure 4. The lambda-lifting algorithm

$$\begin{array}{rcl}
\mathsf{B}_n^m & = & \lambda x y_1 \ldots y_n z_1 \ldots z_m.x(y_1 z_1 \ldots z_m) \ldots (y_n z_1 \ldots z_m) \\
\mathsf{S}_n^{k,m} & = & \lambda y_1 \ldots y_n z_1 \ldots z_k u v_1 \ldots v_m.u(y_1 z_1 \ldots z_k u v_1 \ldots v_m) \ldots (y_n z_1 \ldots z_k u v_1 \ldots v_m) \\
\mathsf{E}_n^m & = & \lambda x y_1 \ldots y_n z_1 \ldots z_m.x z_1 \ldots z_m (y_1 z_1 \ldots z_m) \ldots (y_n z_1 \ldots z_m)
\end{array}$$

Figure 5. Basis combinators for Abdali's algorithm A

$$\begin{array}{rcl}
\mathrm{H}_A(x) & = & x \\
\mathrm{H}_A(t_1 t_2) & = & (\mathrm{H}_A(t_1))(\mathrm{H}_A(t_2)) \\
\mathrm{H}_A(\lambda x.t) & = & [x]_A.t \\
[x_1, \ldots, x_m]_A.t & = & \mathsf{B}_0^i([x_i, \ldots, x_m]_A.t) \\
 & & \text{if } i > 1, x_i \in \mathrm{FV}(t), x_j \notin \mathrm{FV}(t) \text{ for } j < i \\
[x_1, \ldots, x_m]_A.u x_m & = & [x_1, \ldots, x_{m-1}]_A.u \\
 & & \text{if } x_m \notin \mathrm{FV}(u) \\
[x_1, \ldots, x_m]_A.u x_m t_1 \ldots t_n & = & \mathsf{E}_n^m([x_1, \ldots, x_{m-1}]_A.u)([x_1, \ldots, x_m]_A.t_1) \ldots ([x_1, \ldots, x_m]_A.t_n) \\
 & & \text{if } x_m \notin \mathrm{FV}(u) \\
[x_1, \ldots, x_m]_A.x_i t_1 \ldots t_n & = & \mathsf{S}_n^{i-1,m-i}([x_1, \ldots, x_m]_A.t_1) \ldots ([x_1, \ldots, x_m]_A.t_n) \\
[x_1, \ldots, x_m]_A.s t_1 \ldots t_n & = & \mathsf{B}_n^m(\mathrm{H}_A(s))([x_1, \ldots, x_m]_A.t_1) \ldots ([x_1, \ldots, x_m]_A.t_n) \\
 & & \text{if } \{x_1, \ldots, x_m\} \cap \mathrm{FV}(s) = \emptyset, \{x_1, \ldots, x_m\} \cap \mathrm{FV}(t_1) \neq \emptyset \\
[x_1, \ldots, x_m]_A.\lambda y.s & = & [x_1, \ldots, x_m, y]_A.s \\
[x_1, \ldots, x_m]_A.(\lambda y.s)t_1 \ldots t_n & = & \mathsf{E}_n^m([x_1, \ldots, x_m, y]_A.s)([x_1, \ldots, x_m]_A.t_1) \ldots ([x_1, \ldots, x_m]_A.t_n)
\end{array}$$

Figure 6. Abdali's algorithm A

rewrite steps necessary to "evaluate" an application of a lambda-abstraction to an argument. A β-normal lambda-term has one of two forms: $\lambda x_1 \ldots x_m.x_i t_1 \ldots t_n$ or $\lambda x_1 \ldots x_m.y t_1 \ldots t_n$ where $y \notin \{x_1, \ldots, x_n\}$ (possibly $n = 0$ or $m = 0$). For the first form we use the combinator $\mathsf{S}_n^{i-1,m-i}$, and for the second B_n^m. The E_n^m combinators are used to handle "η-contraction" (equations 5 and 6), and terms not in β-normal form.

From the point of view of translating higher-order formalisms to formats of first-order ATPs, an important feature of the algorithm A is that the combinators B_n^m, E_n^m and $\mathsf{S}_n^{k,m}$ always occur with at least n arguments. This makes it possible to represent them in first-order logic as n-ary functions instead of nullary constants, using an optimisation described in [30, Section 2.7]. This happens to have an impact on ATPs' performance by reducing the size of the generated expressions.

The algorithm A combines the advantages of lambda-lifting with the advantages of the abstraction algorithms S and T: "evaluating" an application of a translated lambda-abstraction to an argument requires comparatively few rewrite steps while representations of different lambda-abstractions still share some "structure". The smaller number of steps, compared to algorithms S and T, required to "evaluate" an application of a translated lambda-abstraction to an argument is due to the fact that more "structure" of the lambda-abstraction is encoded directly in the combinators. Thus the steps "pushing down" the argument are fewer and each step does more "work". For instance, for a term $t = \lambda xy.y(xy)(xy)$ we have $\mathrm{H}_S(t) = \mathsf{S}(\mathsf{BS}(\mathsf{SI}))\mathsf{I}$ and $\mathrm{H}_A(t) = \mathsf{S}_2^{1,0}\mathsf{S}_0^{0,0}\mathsf{S}_0^{0,0}$. Then we have

$$\begin{array}{rcl}
\mathsf{S}(\mathsf{BS}(\mathsf{SI}))\mathsf{I}ab & \to & (\mathsf{BS}(\mathsf{SI})a)(\mathsf{I}a)b \\
 & \to & \mathsf{S}(\mathsf{SI}a)(\mathsf{I}a)b \\
 & \to & \mathsf{S}(\mathsf{SI}a)ab \\
 & \to & \mathsf{SI}ab(ab) \\
 & \to & \mathsf{I}b(ab)(ab) \\
 & \to & b(ab)(ab)
\end{array}$$

and

$$\begin{array}{rcl}
\mathsf{S}_2^{1,0}\mathsf{S}_0^{0,0}\mathsf{S}_0^{0,0}ab & \to & b(\mathsf{S}_0^{0,0}ab)(\mathsf{S}_0^{0,0}ab) \\
 & \to & b(ab)(\mathsf{S}_0^{0,0}ab) \\
 & \to & b(ab)(ab)
\end{array}$$

One sees that the steps for $\mathrm{H}_S(t)$ are more "detailed" and thus there are more of them. The use of more specialised combinators by the

algorithm A essentially allows to combine some of the steps into a single step. The downside is that we usually need more distinct combinators if a problem contains several lambda-abstractions.

We did not evaluate the original algorithm of Abdali. Based on preliminary experiments and a heuristic understanding of the desirable properties of the translations, we do not expect it to perform better than the algorithm A. The original algorithm of Abdali uses a slightly more complex encoding for terms of the form $\lambda x_1 \ldots x_m.x_i t_1 \ldots t_n$, and it performs "$\eta$-contractions" less often (an analogon of equation 6 is missing) thus producing larger encodings. However, whether this really affects the performance of the ATPs on translated problems remains to be experimentally verified.

2.5 Director Strings

We also implemented an algorithm D which is essentially a combination of the algorithm A with the idea of director strings [21, 26, 27]. We use the notation $\langle \nu_1, \ldots, \nu_n \rangle_m$ for a list of binary strings $\nu_1, \ldots, \nu_k \in \{0,1\}^m$ all of the same length m. We denote the set of all such lists by L_m. We use the notation $\nu(i)$ for the ith element of the binary string ν. The basis for D is $\{\mathsf{D}_l, \mathsf{U}_l^k \mid k \in \mathbb{N}, l \in L_m, m \in \mathbb{N}\}$ with the definitions of the combinators shown in Figure 7. The notation $\{t\}^b$ should be treated as t if $b = 1$, or omitted if $b = 0$. For example

$$\mathsf{D}_{\langle 010,101 \rangle_3} = \lambda x y z_1 z_2 z_3.x z_2 (y z_1 z_3).$$

The idea is that the list l in D_l and U_l^k is a list of "directors" which specify how the arguments $z_1, \ldots, z_m$ are to be distributed into $x, y_1, \ldots, y_n$.

The algorithm D is defined by the equations in Figure 8, where $m > 0$ and the notation $\{x\}^b$ denotes either x if $b = 1$, or should be omitted if $b = 0$.

For example

$$\begin{array}{lcl} [x,y]_D.x(yz)(zxy)z & = & \mathsf{U}^1_{\langle 01,11,00 \rangle_2}(\mathsf{U}^1_{\langle 0 \rangle_1} z)zz \\ [x,y]_D.x(yz)(zyx)z & = & \mathsf{U}^1_{\langle 01,11,00 \rangle_2}(\mathsf{U}^1_{\langle 0 \rangle_1} z) \\ & & \quad (\mathsf{D}_{\langle 01,10 \rangle_2} z \mathsf{I})z \\ [x,y,z]_D.z(yz)(xy) & = & \mathsf{U}^2_{\langle 011,110 \rangle_3} \mathsf{II} \end{array}$$

where we write I instead of $\mathsf{U}^1_{\langle\rangle_1}$.

The algorithm D generally produces shorter encodings than A which also require fewer rewrite steps for the "evaluation" of an application of a translation of a term to some arguments. This is at the cost of using more "specialised" combinators and thus reducing "sharing" between translations of different lambda-abstractions.

2.6 Typability

In the presentation of abstraction algorithms we have ignored the issues of typability, working in the untyped lambda-calculus. However, the terms of higher-order logic as implemented in HOL Light are simply typed, with support for rank-1 polymorphism. It is a simple exercise to find the principal type for every base combinator we introduced (for lambda-lifting we need to restrict the base to typable supercombinators), and show the following proposition for each of the abstraction algorithms (where $m = 1$ for univariate algorithms, and $m \in \mathbb{N}$ arbitrary for multivariate ones), by induction on the structure of t.

Proposition 2.1. *If* $\Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash t : \tau$ *then* $\Gamma \vdash [x_1, \ldots, x_m].t : \tau_1 \to \ldots \to \tau_m \to \tau$.

Here $\vdash$ denotes derivability in the Curry-style system of simple types [18], and Γ is a type context such that $x_1, \ldots, x_m$ are not present in Γ. The proposition implies that the translations preserve the (simple) types of terms. Analogously, one verifies that the algorithms may be implemented to operate on the Church-style typed terms of the HOL logic.

3. Implementation

We integrated our lambda-abstraction elimination methods into the existing implementation of HOL(y)Hammer. In this section we describe how this integration was done and what choices were made regarding some details of the algorithms left unspecified in the previous sections.

HOL(y)Hammer uses a sound fully typed translation originally due to Hurd [20]. For every method, before the elimination of lambda-abstractions the preprocessing outlined in [24] (based on [30]) is performed. In particular, the following is performed.

- Any equation between a constant and a lambda-expression is translated directly to another equation. For example, the formula $h = \lambda x.fx(gx)$ is translated to $\forall x.hx = fx(gx)$.
- All terms are reduced to $\beta\eta$-normal form.

The preprocessing eliminates the majority of lambda-abstractions, so our abstraction algorithms are actually used only with more complicated lambda-expressions. Note that because of the last point above we only ever translate terms in β-normal form, so the last equations in the definitions of the algorithms A and D are never actually used. It actually rarely occurs that theorems have any β-redexes, and when they occur they are usually simple and the reduction to β-normal form does not cause any blow-up in the term size.

In the previous section, for each abstraction algorithm we have defined the base combinators as certain closed lambda-terms. In the implementation of the translation each combinator is replaced by a constant polymorphically typed with the principal type of the combinator, and an axiom is added for the constant expressing the definition of the combinator as a universally quantified equation. Of course, all occurences of the same combinator are replaced by the same constant. For instance, the combinator $\mathsf{K} = \lambda xy.x$ is replaced with a constant $\mathsf{K}^{\alpha \to \beta \to \alpha}$ (with the superscript indicating the polymorphic type) and the following defining axiom is added:

$$\forall x : \alpha. \forall y : \beta. (\mathsf{K}^{\alpha \to \beta \to \alpha} x y = x).$$

The removal of combinators occurs still within higher-order logic. It is essential for the performance of ATPs that the defining axioms are added only for those combinators which actually occur in the translated terms.

For lambda-lifting the supercombinators may be nested – they are replaced inside-out using the method outlined above. Actually, we evaluated two variants of lambda-lifting: $L1$ and $L2$. The lambda-lifitng algorithm $L2$ proceeds as described above, replacing supercombinators with new constants and adding defining axioms. The lambda-lifitng algorithm $L1$, which is actually the original algorithm implemented in HOL(y)Hammer, replaces the supercombinators with variables instead, adding their definitions as guards. For example, the goal $p(\lambda xy.x)$ is replaced by the algorithm $L1$ with (we omit the types)

$$\forall f.(\forall xy.fxy = x) \to pf.$$

As revealed by the experimental evidence of the next section, these two variations differ dramatically in terms of the resulting ATPs' performance. One of the reasons is that translating supercombinators to constants allows us to subsequently perform the following optimisation (originally by Meng and Paulson [30]) for translating higher-order constants: for each such constant c create a first-order function that has the minimum arity with which c is used in the particular set of HOL formulas that is used to create the first-order ATP problem. In particular, this implies that if the lambda-abstraction has n free variables then it will always be translated into an n-ary function (see Section 2.3).

$$\begin{array}{rcl}
\mathsf{D}_{\langle\nu_0,\ldots,\nu_n\rangle_m} & = & \lambda x y_1 \ldots y_n z_1 \ldots z_m . x \{z_1\}^{\nu_0(1)} \ldots \{z_m\}^{\nu_0(m)} (y_1 \{z_1\}^{\nu_1(1)} \ldots \{z_m\}^{\nu_1(m)}) \ldots (y_n \{z_1\}^{\nu_n(1)} \ldots \{z_m\}^{\nu_n(m)}) \\
\mathsf{U}^k_{\langle\nu_1,\ldots,\nu_n\rangle_m} & = & \lambda y_1 \ldots y_n z_1 \ldots z_m . z_k (y_1 \{z_1\}^{\nu_1(1)} \ldots \{z_m\}^{\nu_1(m)}) \ldots (y_n \{z_1\}^{\nu_n(1)} \ldots \{z_m\}^{\nu_n(m)})
\end{array}$$

Figure 7. Basis combinators for algorithm D

$$\begin{array}{rcl}
\mathrm{H}_D(x) & = & x \\
\mathrm{H}_D(t_1 t_2) & = & (\mathrm{H}_D(t_1))(\mathrm{H}_D(t_2)) \\
\mathrm{H}_D(\lambda x.t) & = & [x]_D.t \\
[x_1,\ldots,x_m]_D.t & = & \mathsf{D}_{\langle 0^m\rangle_m}([x_i,\ldots,x_m]_D.t) \\
 & & \text{if } i>1, x_i \in \mathrm{FV}(t), x_j \notin \mathrm{FV}(t) \text{ for } j<i \\
[x_1,\ldots,x_m]_D.u x_m & = & [x_1,\ldots,x_{m-1}]_D.u \\
 & & \text{if } x_m \notin \mathrm{FV}(u) \\
[x_1,\ldots,x_m]_D.u x_m t_1 \ldots t_n & = & \mathsf{D}_{\langle\nu_0,\nu_1,\ldots,\nu_n\rangle_m} u' t'_1 \ldots t'_n \\
 & & \text{where } x_m \notin \mathrm{FV}(u), \\
 & & u' = [\{x_1\}^{\nu_0(1)},\ldots,\{x_{m-1}\}^{\nu_0(m-1)}]_D.u, \\
 & & t'_i = [\{x_1\}^{\nu_i(1)},\ldots,\{x_m\}^{\nu_i(m)}]_D.t_i \text{ for } i=1,\ldots,n \\
 & & \nu_0(m)=1, \nu_0(j)=1 \text{ iff } x_j \in \mathrm{FV}(u), \text{ for } j=1,\ldots,m-1 \\
 & & \nu_i(j)=1 \text{ iff } x_j \in \mathrm{FV}(t_i), \text{ for } i=1,\ldots,n, j=1,\ldots,m, \\
[x_1,\ldots,x_m]_D.x_i t_1 \ldots t_n & = & \mathsf{U}^i_{\langle\nu_1,\ldots,\nu_n\rangle_m} t'_1 \ldots t'_n \\
 & & \text{where } t'_i = [\{x_1\}^{\nu_i(1)},\ldots,\{x_m\}^{\nu_i(m)}]_D.t_i \text{ for } i=1,\ldots,n \\
 & & \nu_i(j)=1 \text{ iff } x_j \in \mathrm{FV}(t_i), \text{ for } i=1,\ldots,n, j=1,\ldots,m, \\
[x_1,\ldots,x_m]_D.s t_1 \ldots t_n & = & \mathsf{D}_{\langle 0^m,\nu_1,\ldots,\nu_n\rangle_m}(\mathrm{H}_D(s)) t'_1 \ldots t'_m \\
 & & \text{where } \{x_1,\ldots,x_m\} \cap \mathrm{FV}(s) = \emptyset, \\
 & & \{x_1,\ldots,x_m\} \cap \mathrm{FV}(t_1) \neq \emptyset, \\
 & & t'_i = [\{x_1\}^{\nu_i(1)},\ldots,\{x_m\}^{\nu_i(m)}]_D.t_i \text{ for } i=1,\ldots,n \\
 & & \nu_i(j)=1 \text{ iff } x_j \in \mathrm{FV}(t_i), \text{ for } i=1,\ldots,n, j=1,\ldots,m, \\
[x_1,\ldots,x_m]_D.\lambda y.s & = & [x_1,\ldots,x_m,y]_D.s \\
[x_1,\ldots,x_m]_D.(\lambda y.s) t_1 \ldots t_n & = & \mathsf{D}_{\langle\nu_0,\ldots,\nu_n\rangle_m} s' t'_1 \ldots t'_n \\
 & & \text{where } s' = ([\{x_1\}^{\nu_0(1)},\ldots,\{x_m\}^{\nu_0(m)},y]_D.s) \\
 & & t'_i = [\{x_1\}^{\nu_i(1)},\ldots,\{x_m\}^{\nu_i(m)}]_D.t_i \text{ for } i=1,\ldots,n \\
 & & \nu_0(j)=1 \text{ iff } x_j \in \mathrm{FV}(\lambda y.s), \text{ for } j=1,\ldots,m \\
 & & \nu_i(j)=1 \text{ iff } x_j \in \mathrm{FV}(t_i), \text{ for } i=1,\ldots,n, j=1,\ldots,m
\end{array}$$

Figure 8. The algorithm D

Another possible reason for the poor performance of $L1$ versus $L2$ is that it always adds universal quantification over variables replacing the supercombinators, not only in the goal but also in the axioms where existential quantification would seem more appropriate. We evaluated the $L1$ version of lambda-lifting simply because it is the "baseline" algorithm originally implemented in HOL(y)Hammer. In other words, the translation using $L1$ is exactly the translation originally done by HOL(y)Hammer as described in [24].

For the algorithms A and D the equations for the base combinators may be expressed recursively instead of directly using the equations from Sections 2.4–2.5. For instance, for $m > 0$ we have

$$\mathsf{B}^{m+1}_n x y_1 \ldots y_n z =_\beta \mathsf{B}^m_n x (y_1 z) \ldots (y_n z).$$

Of course, when using the recursive equations, the axioms for combinators appearing in the right-hand sides must be (recursively) added, even if these combinators do not occur in any of the translated terms. We implemented the recursive equations. They yield a small but consistent and noticeable improvement in the performance of the translations, presumably by enabling partial applications and thus introducing more "sharing" between translations of different lambda-abstractions.

4. Experimental Evaluation

We evaluated the algorithms described in the previous sections on those problems from the HOL Light core library and a library for multivariate analysis [16] which contained non-trivial lambda-abstractions in the goal or the dependencies, i.e. lambda-abstractions that could not be eliminated by the preprocessing described in Section 3. Note that these tend to be the more difficult problems, which explains the low success rate. We used the following ATPs: CVC4 version 1.3, E version 1.8, Vampire version 3.0, Z3 version 4.0 and SPASS version 3.5. The methodology was to measure the number of theorems that the ATP could reprove from their dependencies within a time limit of 30 s for each problem. We removed from the dependencies the definitions of logical connectives, which harmed the success rate and are useless because logical connectives are translated directly. The evaluation was performed on a workstation with 48 2.1GHz AMD cores and 320GB RAM, where the ATPs were run in parallel to occupy all the cores. Table 1 shows the cumulative results, i.e., the cumulative number and percentage of problems that were solved by any of the provers using a given method for eliminating lambda-abstractions. The column "Solved%" denotes the percentage (rounded to the first decimal place) of the problems solved, and "Solved" the number of problems solved out of the total number of 6605 problems. The table "Greedy sequence" presents a greedy sequence constructed as follows: first take the best performing algorithm, then take the algo-

Algorithm	Solved%	Solved
Lambda-lifting $L2$	23.8	1569
Director strings D	23.4	1545
Abdali A	22.9	1511
Turner T^*	22.9	1511
Turner T	22.6	1493
Schönfinkel S	21.6	1417
Lambda-lifting $L1$	10.6	699
any	28.2	1862

Greedy sequence

Algorithm	Sum%	Sum
Lambda-lifting $L2$	23.8	1569
Director strings D	27.8	1834
Abdali A	28.0	1848
Turner T^*	28.1	1854
Lambda-lifting $L1$	28.2	1860
Schönfinkel S	28.2	1862

Table 1. Cumulative results

Algorithm	Solved%	Solved
Director strings D	17.2	1139
Turner T^*	17.2	1136
Lambda-lifting $L2$	17.2	1135
Turner T	17.0	1123
Abdali A	16.9	1114
Schönfinkel S	16.1	1061
Lambda-lifting $L1$	7.8	518
any	21.7	1435

Greedy sequence

Algorithm	Sum%	Sum
Director strings D	17.3	1140
Lambda-lifting $L2$	21.0	1390
Abdali A	21.3	1410
Lambda-lifting $L1$	21.6	1424
Turner T^*	21.7	1434
Turner T	21.7	1435

Table 2. Results for CVC4 1.3

Algorithm	Solved%	Solved
Lambda-lifting $L2$	10.8	715
Schönfinkel S	10.8	713
Director strings D	10.7	708
Turner T	10.5	691
Turner T^*	10.5	691
Abdali A	10.4	690
Lambda-lifting $L1$	6.2	412
any	15.5	1463

Greedy sequence

Algorithm	Sum%	Sum
Lambda-lifting $L2$	10.8	715
Schönfinkel S	14.8	974
Abdali A	15.0	992
Lambda-lifiting $L1$	15.3	1008
Director strings D	15.4	1014
Turner T	15.4	1017
Turner T^*	15.5	1019

Table 3. Results for E 1.8

Algorithm	Solved%	Solved
Director strings D	19.0	1252
Turner T^*	18.6	1225
Abdali A	18.5	1224
Lambda-lifting $L2$	18.3	1210
Turner T	18.3	1209
Schönfinkel S	17.1	1129
Lambda-lifting $L1$	7.8	512
any	23.4	1546

Greedy sequence

Algorithm	Sum%	Sum
Director strings D	19.0	1252
Lambda-lifting $L2$	22.8	1508
Abdali A	23.1	1524
Lambda-lifting $L1$	23.3	1536
Turner T^*	23.4	1543
Schönfinkel S	23.4	1546

Table 4. Results for Z3 4.0

Algorithm	Solved%	Solved
Lambda-lifting $L2$	9.9	657
Abdali A	9.8	648
Director strings D	9.8	648
Turner T^*	9.6	631
Turner T	9.5	630
Schönfinkel S	9.2	608
Lambda-lifting $L1$	6.7	442
any	13.8	909

Greedy sequence

Algorithm	Sum%	Sum
Lambda-lifting $L2$	9.9	657
Director strings D	13.0	859
Lambda-lifting $L1$	13.5	890
Abdali A	13.6	900
Schönfinkel S	13.8	909

Table 5. Results for Vampire 3.0

Algorithm	Solved%	Solved
Abdali A	5.1	336
Schönfinkel S	5.1	334
Director strings D	5.0	333
Turner T	5.0	328
Turner T^*	5.0	327
Lambda-lifting $L2$	3.9	257
Lambda-lifting $L1$	2.9	188
any	7.2	472

Greedy sequence

Algorithm	Sum%	Sum
Abdali A	5.1	336
Lambda-lifting $L2$	6.6	437
Lambda-lifting $L1$	7.0	460
Schönfinkel S	7.2	472

Table 6. Results for SPASS 3.5

rithm that increases the number of solved problems the most, and so on. As mentioned in Section 3, most lambda-abstractions were eliminated by the preprocessing, so we preselected those problems which contained some lambda-abstractions in the goal or the dependencies that could not be eliminated by the preprocessing – only on these problems the output of the different algorithms actually differs. This explains the generally low success rate, because the problems containing non-trivial lambda-abstractions, either in the goal or the dependencies, tend to be the more difficult ones.

In Tables 2–6 we show the evaluation results for each prover separately. The results show that lambda-lifting performs better with the provers CVC4, E, Z3 and Vampire, while Schönfinkel's and Turner's abstraction algorithms S and T perform better with SPASS. The algorithms A and D perform similarly to lambda-lifting with CVC4, Z3 and Vampire, and similarly to algorithms S and T with the other provers. The modified Turner's algorithm T^* always either outperforms T or gives very similar results. Except for the provers E and SPASS, the algorithm T^* performs better than S. Cumulatively, the improvement is significant. This is in contrast to [30] where a version of the algorithm T^* was found to yield no significant improvement over Schönfinkel's algorithm.

A raw data package for our evaluation, which includes the original problems, their translations using each of the algorithms, and the output of the provers on the translated problems, is available at `http://www.mimuw.edu.pl/~lukaszcz/cpp2016.tar.bz2`.

5. Conclusions and Future Work

Our work shows that it is possible to improve automation in proof assistants by enhancing the translations of lambda-abstractions in state-of-the-art hammers. Guided by heuristic considerations and basing on existing literature we devised two algorithms that on average outperform the standard combinatory abstraction algorithms. There are many abstraction algorithms in the published literature and they may be tweaked and modified in a great number of ways. We have considered only a small fraction of published methods.

We now outline two directions we have not pursued in this paper which seem to merit further investigation. First, our evaluation was performed on "small" (but difficult) problems with human-selected dependencies. A more realistic scenario is an evaluation on "large" problems with a large number of dependencies selected by an automatic relevance filter. This evaluation is yet to be performed. Secondly, the work of Broda and Damas [6, 8] seems interesting as their combinatory abstraction algorithm produces compact representations which moreover guarantee that the length of a reduction to the normal form of a translated term is not greater than the length of the leftmost reduction of the original term to a lambda-free normal form.

Aside of combinatory abstraction algorithms, a different but closely related approach to translating lambda-abstractions into a first-order format are explicit substitution calculi [28, 32]. In fact, the method of director strings is presented as a calculus of explicit substitutions in [34]. Some other calculi of explicit substitutions from the literature [13, 14] are also based on specialised combinators. As far as the more typical calculi of explicit substitutions are concerned, like e.g. $\lambda\sigma$ [1], the author does not think that they are well-suited for the purpose of translating higher-order logic to first-order logic. They have been designed with the intention of mimicking β-reduction while making substitutions and operations on them an explicit part of the calculus instead of meta-level notions. The reduction steps are thus quite "small". In particular, some operations are required to push a substitution below a lambda – this does not occur at all even with the combinatory abstraction algorithms S and T, because all lambdas are simply eliminated. We believe that our algorithms A and D owe their success partly to the fact that they strike a balance between "large" reduction steps (as with lambda-lifting) and a lot of "sharing" between translations of different lambda-abstractions (as with algorithms S and T). However, these conclusions are based on a heuristic understanding of the problem and have not been fully backed up by experimental evidence, i.e., we have not implemented or evaluated any methods based on calculi of explicit substitutions. Whether explicit substitution calculi are fit for our purposes remains to be experimentally verified.

Acknowledgments

The author would like to thank Cezary Kaliszyk and the anonymous referees for helpful comments on earlier versions of this paper.

References

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

[2] S. K. Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41(1):222–224, 1976.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[4] J. C. Blanchette. *Automatic Proofs and Refutations for Higher-Order Logic*. PhD thesis, Technische Universität München, 2012.

[5] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 2015. (to appear).

[6] S. Broda and L. Damas. Compact bracket abstraction in combinatory logic. *Journal of Symbolic Logic*, 62(3):729–740, 1997.

[7] M. W. Bunder. Some improvements to Turner's algorithm for bracket abstraction. *Journal of Symbolic Logic*, 55(2):656–669, 1990.

[8] M. W. Bunder. Expedited Broda-Damas bracket abstraction. *Journal of Symbolic Logic*, 65(4):1850–1857, 2000.

[9] F. W. Burton. A linear space translation of functional programs to Turner combinators. *Information Processing Letters*, 14(5):201–204, 1982.

[10] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. North-Holland, 1958.

[11] Ł. Czajka. On the equivalence of different presentations of Turner's bracket abstraction algorithm. *CoRR*, abs/1510.03794, 2015.

[12] T. Gauthier and C. Kaliszyk. Premise selection and external provers for HOL4. In X. Leroy and A. Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 49–57. ACM, 2015.

[13] H. Goguen and J. Goubault-Larrecq. Sequent combinators: a Hilbert system for the lambda calculus. *Mathematical Structures in Computer Science*, 10(1):1–79, 2000.

[14] J. Goubault-Larrecq. Conjunctive types and SKInT. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, March 27-31, 1998, Selected Papers*, volume 1657 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 1998.

[15] T. C. Hales. Developments in formal proofs. *CoRR*, abs/1408.6474, 2014.

[16] J. Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50(2):173–190, 2013.

[17] T. Hikita. On the average size of Turner's translation to combinator programs. *Journal of Information Processing*, 7(3):164–169, 1984.

[18] J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[19] R. J. M. Hughes. Supercombinators: A new implementation method for applicative languages. In *LISP and Functional Programming*. ACM Press, 1982.

[20] J. Hurd. An LCF-style interface between HOL and first-order logic. In *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, pages 134–138, 2002.

[21] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[22] M. S. Joy. *On the efficient implementation of combinators as an object code for functional programs*. PhD thesis, University of East Anglia, 1984.

[23] M. S. Joy, V. J. Rayward-Smith, and F. W. Burton. Efficient combinator code. *Computer Languages*, 10(3/4):211–224, 1985.

[24] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

[25] C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[26] R. Kennaway and M. R. Sleep. Variable abstraction in $O(n \log n)$ space. *Information Processing Letters*, 24(5):343–349, 1987.

[27] R. Kennaway and M. R. Sleep. Director strings as combinators. *ACM Trans. Program. Lang. Syst.*, 10(4):602–626, 1988.

[28] D. Kesner. The theory of calculi with explicit substitutions revisited. In J. Duparc and T. A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2007.

[29] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban. Mash: Machine learning for Sledgehammer. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2013.

[30] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[31] J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[32] K. H. Rose. Explicit substitution: tutorial & survey. Technical report, BRICS, Department of Computer Science, University of Aarhus, 1996.

[33] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[34] F. Sinot, M. Fernández, and I. Mackie. Efficient reductions with director strings. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.

[35] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, 1979.