Verified Translation to Directional Combinators

Megan Bailey

May 4, 2008

1 Introduction

It is well-known that lambda calculus terms involving bound variables can be translated into combinator terms without bound variables [2]. In performing these translations, it is valuable to have a set of combinators that produces a smaller, more manageable, more readable output as well as an implementation of a program to determine that output. By implementing the program in a functional programming language that allows for internal type verification, we can verify that the translator preserves the types of all elements throughout a translation. This validates the correctness of the translated expression, and thus validates the combinators themselves along with their functions. Directional Combinators and the algorithm for their translation were developed with these ideas and implemented in the Guru language.

2 The Algorithm

The idea that serves as the foundation of Directional Combinators is that an argument is not passed into a branch of a lambda expression where the corresponding variable is not present. This type of optimization is commonly seen in combinatory logic, from the typical optimization of the combinator K, to the B and C combinators that only send arguments into the left or right branches. This basic idea is seen in Turner's Supercombinators [3]. To simplify implementation and readability, however, Turner's algorithm is expanded to include two lambda functions and additional combinators to accommodate the extra function.

The set is constructed so that an argument can be sent into the left branch, right branch, both, or neither by simply adding a combinator onto the front of a string of combinators preceding the two branches. This idea, however, requires that any input must be of the form $C\ t_1\ t_2$ where C is a chain of combinators followed by the two terms. Thus, Directional Combinators must include an additional lambda function, one that puts any expression into this form and is called first. Other combinators are also needed to handle the inputs of the form

$C\ t_1$. In the end, we have two lambda functions and nine combinators that give the desired type of output.

2.1 Directional Combinators

They are called "directional" because when an argument is passed into an expression, the combinator at the beginning of each chain tells you which direction to go. L, R, B, and N point the argument to the left term, right term, both, or neither respectively. P and Q determine whether to send it into the solitary term or not. The base three, S, K, and I, are used as the building blocks for the primary lambda function in transforming an expression into the correct form. Now each Directional Combinator and its reduction rule will be discussed in more detail.

2.1.1 The SKI Combinators

The set of Directional Combinators includes the original combinators S and K, first developed by Curry and Feys [2]. S takes in three arguments and applies the last argument to each of the first. K takes only two arguments and simply returns the first. Any lambda expression can be translated into an expression using only these two combinators. The combinator I is conveniently used as an identity combinator but is not completely necessary because it can be defined using S and K. It takes in one argument and returns it unaltered. Their reduction rules are

$$\begin{aligned} S\ t_1\ t_2\ a &\longrightarrow (t_1\ a)\ (t_2\ a) \\ K\ t_1\ a &\longrightarrow a \\ I\ a &\longrightarrow a \end{aligned}$$

Although these three combinators are not the most efficient for translations, they can be optimized or serve as a foundation for larger sets as seen in Turner's Supercombinators and other BC optimizations. In a similar manner, additional combinators have been added to them to create the set of Directional Combinators that ultimately produce shorter translated expressions.

2.1.2 The P and Q Combinators

P and Q are used for translating expressions that only have one term after the combinator chain. Because there is only one term, there are only two options, either to pass the argument into the term or to do nothing. P and Q handle those cases respectively. Their reduction rules are

$$\begin{aligned} P\ c\ t\ a &\longrightarrow c\ (t\ a) \\ Q\ c\ t\ a &\longrightarrow c\ t \end{aligned}$$

2.1.3 The L, R, B, and N Combinators

The final four combinators direct an argument when there are two terms applied to each other following the combinator chain. They represent the option of taking the left, right, both, or neither path. The left and right combinators are based on the B and C combinators found in Turner's algorithm, and the "both" combinator could be compared to his S' combinator. The four reduction rules for them are

$$\begin{array}{rcl} L\,c\,t_1\;t_2\;a & \longrightarrow & c\;(t_1\;a)\;t_2 \\ R\,c\,t_1\;t_2\;a & \longrightarrow & c\,t_1\;(t_2\;a) \\ B\;c\,t_1\;t_2\;a & \longrightarrow & c\;(t_1\;a)\;(t_2\;a) \\ N\;c\,t_1\;t_2\;a & \longrightarrow & c\,t_1\;t_2 \end{array}$$

2.2 Lambda One and Lambda Star Functions

Lambda and Lambda Star can translate any expression into combinators by using Lambda One for the first variable abstraction and then Lambda Star for all subsequent abstractions. If the expression $(x\;y)\;z$ is submitted as input, a translation setup would be

$$\lambda_*\;x.\;\lambda_*\;y.\;\lambda_1\;z.\;(x\;y)\;z$$

The order that the variables are abstracted out of the expression is not restricted. It is only required that the first abstraction use λ_1.

2.2.1 Control Combinators

The λ_1 function is a way to transform formulas through the first variable abstraction into a form that λ_* accepts. It does this by inserting control combinators. These combinators are S, K, and I, and act as the last element in every chain of combinators during the translation. Having these markers signaling the end of a combinator string creates forms suitable for multiple variable abstractions.

2.2.2 Transfomation Rules for Lam

$$\begin{array}{rcl} \lambda_1\;x.\;x & = & I\;I \\ \lambda_1\;x.\;t & = & K\;t \\ \lambda_1\;x.\;t_1\;t_2 & = & S\;(\lambda_1\;x.\;t_1)\;(\lambda_1\;x.\;t_2) \end{array}$$

These three rules result in expressions of the form $C\;t_1\;t_2$ or $C\;t_1$ with S, K, or I as the control combinator. The λ_* function assumes it is given input that satisfies this property.

2.2.3 Transformation Rules for Lamstar

$$\begin{aligned}
\lambda_* \, x. \, C \, t &= Q \, C \, t \\
\lambda_* \, x. \, C \, x &= P \, C \, I \\
\lambda_* \, x. \, C \, t_1 \, t_2 &= B \, C \, (\lambda_* \, x. \, t_1) \, (\lambda_* \, x. \, t_2) \\
\lambda_* \, x. \, C \, t_1 \, t_2 &= L \, C \, (\lambda_* \, x. \, t_1) \, t_2 \\
\lambda_* \, x. \, C \, t_1 \, t_2 &= R \, C \, t_1 \, (\lambda_* \, x. \, t_2) \\
\lambda_* \, x. \, C \, t_1 \, t_2 &= N \, C \, t_1 \, t_2
\end{aligned}$$

2.3 Translation Size

For a translation of a term of size n with m variables abstractions, the length of the output using our algorithm is $\theta(m*n)$. This is just as efficient as other similar sets of combinators and their abstraction algorithms. Joy, Smith, and Burton [4] provide a clear summary of many common sets of combinators and the upper bounds of their translation lengths. These upper bounds range asymptotically from exponential to linear in terms of n. At the exponential end, they analyze the basic SKI scheme with no optimization techniques, called CL-SKI with an additional Y combinator, showing it has an upper bound of $3^n - (3^m - 1)/2$. On the more efficient end of the range, the set of ten combinators called CL-J also outputs lengths of size $\theta(m * n)$. Although it is not improving on this expression, the set of Directional Combinators remain at the linear end of the range of upper bounds for combinatory logic.

3 Type Preservation

It is useful not only to develop the set of combinators, but also implement the translation algorithm in a way that preserves every term's type throughout the translation. This involves including and updating the formula that the translation proves as each combinator, and its corresponding formula, is applied to the expression. This ensures that combinators are not added to the expression incorrectly. Terms being applied to the combinator must have the correct form to fit into the combinator's proof formula. This type checking also ensures that translation rules along with their implementation are correct.

Also included is a list of assumptions with every term in the input. This list decreases in size as each abstraction occurs, and when every variable abstraction has been performed, every term's list should be empty. The lambda functions are designed to use the first assumption found in the list. Each assumption contains a number corresponding to its position in the list as well as a proof that that number is correct. As assumptions are removed from the list, these numbers decrease until the last item is removed. This data allows the user to know how many variable abstractions are left to perform.

3.1 Formulas Proven by Combinators

$$\begin{array}{rcl}
S & : & (P \to Q \to R) \to (P \to Q) \to P \to R \\
K & : & P \to Q \to P \\
I & : & P \to P \\
P & : & (Q \to S) \to (P \to Q) \to (P \to S) \\
Q & : & (Q \to S) \to Q \to (P \to S) \\
L & : & (Q \to R \to S) \to (P \to Q) \to R \to P \to S \\
R & : & (Q \to R \to S) \to Q \to (P \to R) \to P \to S \\
B & : & (Q \to R \to S) \to (P \to Q) \to (P \to R) \to P \to S \\
N & : & (Q \to R \to S) \to Q \to R \to P \to S
\end{array}$$

4 Guru Implementation

Guru is a functional programming language that allows the desired type of implementation to be created. Programs written using Guru can contain proofs, so verification of properties of data types occurs internally. Data types are built using the keyword 'Inductive' and type constructors. These constructors have the option of taking in arguments using the 'fun' command, indicating that we are defining something that is a function of the following parameters. The last expression following a period, or a colon if no arguments are declared, specifies the return type. Functions are created using the 'Define' instruction and use similar syntax to the 'Inductive' command. More information about the Guru programming language can be found at *http://cl.cse.wustl.edu/*.

4.1 Key Data Types

```
Inductive form:type:=
         imp:Fun(p q:form).form
        |var:Fun(x:nat).form
.
```

The basis of any input given to the algorithm is the data type 'form'. It includes many common formula operators such as 'and', 'or', and implication. The rest of the data types are built from these as well as from the list type which is defined in the Guru library.

```
Inductive pf:Fun(f:form)(l:<list form>).type:=
        Mp:Fun(f f':form)(l:<list form>)(p:<pf(imp f f') l>)
              (p':<pf f l>).<pf f' l>
        |Assump:Fun(p:form)(l:<list form>)(n:nat)
              (u:{(nth form n l)=p}).<pf p l>
.
```

The 'pf' data type takes in a form and a list of forms, the list of assumptions that is kept up-to-date throughout the algorithm. There are only two types of proofs, Mp and Assump. Mp stands for modus ponens and indicates and an application of one argument with type f onto another with type $(imp\ f\ f')$. Therefore, Mp is a proof of formula f'. Assump is simply an assumption of some argument along with the previously discussed proof that it is where it claims to be in the master list of assumptions.

```
Inductive control:Fun(f:form).type:=
        I:Fun(p:form).<control (imp p p) >
        |s:Fun(p q r:form).<control (imp (imp p (imp q r))
              (imp (imp p q)(imp p r)))>
        |K:Fun(p q:form).<control (imp p (imp q p))>
        |P:Fun(p q s:form)(c:<control (imp q s)>).
              <control (imp (imp p q) (imp p s))>
        |Q:Fun(p q r:form)(c:<control (imp q r)>).
              <control (imp q (imp p r))>
        |L:Fun(p q r s:form)(c:<control (imp q (imp r s))>).
              <control (imp (imp p q) (imp r (imp p s)))>
        |R:Fun(p q r s:form)(c:<control (imp q (imp r s))>).
              <control (imp q (imp (imp p r) (imp p s)))>
        |B:Fun(p q r s:form)(c:<control (imp q (imp r s))>).
              <control (imp (imp p q) (imp (imp p r) (imp p s)))>
        |N:Fun(p q r s:form)(c:<control (imp q (imp r s))>).
              <control (imp q (imp r (imp p s)))>
        |Mp2:Fun(p q:form)(c:<control (imp p q)>).<control q>
.
```

The 'control' data type includes the nine previously defined directional combinators. S, K, and I take forms and return a proof of their respective formula using those forms. The rest of the combinators must be added onto the front of a list so in addition to the forms, they require the first term of their proof formula and return a proof of the second half of the outermost implication.

```
Inductive pf2:Fun(f:form)(l:<list form>).type:=
        CtrlMp:Fun(p q r:form)(l:<list form>)
              (c:<control (imp p (imp q r))>)(t1:<pf2 p l>)
        (t2:<pf2 q l>).<pf2 r l>
    |CtrlI:Fun(p r:form)(l:<list form>)(c:<control (imp p r)>).
        <pf2 r l>
    |CtrlVar:Fun(p r:form)(l:<list form>)(c:<control (imp p r)>)
        (n:nat).<pf2 r l>
.
```

The 'pf2' data type is used for the λ_* function and is similar to the first proof type. It includes three terms called CtrlMp, CtrlI, and CtrlVar. These differ based on what will be found after the chain of combinators whether it is an application, a term without any free variables, or a term with free variables. They require the typical forms and list of assumptions along with a control argument indicating the first combinator chain and the terms applied to it.

```
Inductive lamstar_result:Fun(f1 f2:form)(l:<list form>).type :=
        result1:Fun(f1 f2:form)(l:<list form>)(p:<pf2 (imp f2 f1) l>).
              <lamstar_result f1 f2 l>
        |result2:Fun(f1 f2:form)(l:<list form>)(p:<pf2 f1 l>).
              <lamstar_result f1 f2 l>
.
```

Finally, the 'lamstar_result' data type is an indicator of whether the current variable being abstracted, $f2$, was found in the branch that was traversed, $f1$. The result1 instance of this means that it was found, and thus, requires a proof that the assumption implies the translated expression. Result2 on the other hand requires only a proof of the expression, unchanged because the variable was absent so no combinators were needed.

4.2 Types of Functions

The implementation includes two main functions, one for λ_1 and one for λ_*. The lamstar function also requires the use of two helper functions called case1 and case2.

```
Define lam :=
        fun lam(f1 f2:form)(l:<list form>)(p:<pf f1 (cons form f2 l)>):
        <pf2 (imp f2 f1) l>.
```

Lam takes in the variable being abstracted (form f2), the expression being translated (form f1), and a proof of that expression under an assumption of f2. After translating it into combinators using the rules in section 2.2.2, that assumption does not appear in the corresponding list, and the function outputs a proof2 that f2 implies f1.

```
Define lamstar :=
        fun lamstar(f1 f2:form)(l:<list form>)(p:<pf2 f1 (cons form f2 l)>):
        <lamstar_result f1 f2 l>.
```

Lamstar implements the translation rules found in section 2.2.3. It requires almost identical inputs as lam but instead of taking in a proof, it takes in a proof2, which by construction is in the correct form to run correctly in the λ_* translation. Instead of returning a proof2, this function returns a lamstar_result which actually holds the proof2 inside of it while indicating whether the variable was ever used in the expression.

```
Define case1 :=
        fun case1(f1 f2:form)(l:<list form>)(p:<pf2 f1 (cons form f2 l)>):
        <lamstar_result f1 f2 l>.
```

Within the λ_* function, two helper functions are used called case1 and case2. They have the same signature and handle the two main types of expressions given to λ_*. Case1 deals with input of the form $C\ t_1$ while case 2 deals with the form $C\ t_1\ t_2$ where C is a string of combinators. In the first case, P and Q combinators must be added to the front whether the input variable is present or not. This is due to the fact that there is only one term so as each variable is put back into the equation, each one needs to know whether to be passed into the one term or not. The second case does not add Q combinators anywhere. When the translation gets deep enough to be in the case where there is one term and it does not find it there, a chain of combinators outside this one term will reflect that fact, so no Q is needed.

5 Conclusion

This paper has developed a verified translation to Directional Combinators. This approach may have applications to more general verified compilation.

References

[1] Bunder, M.W. Some Improvements to Turner's Algorithm for Bracket Abstraction. *The Journal of Symbolic Logic,* 55 (1990), 656-669.

[2] Currey, H.B., Feys R. "Combinatory Logic." Vol. 1. North-Holland, Amsterdam, 1958.

[3] Jones, Simon Peyton. "SK Combinators." *The Implementation of Functional Programming Languages.* Prentice Hall, 1987. 260-280.

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

[5] Noshita, Koohei, and Hikita, Teruo. The BC-Chain Method for Representing Combinators in Linear Space. *Lecture Notes in Computer Science,* 220 (1986), 292-306.