BRICS RS-95-38 M. Goldberg: Gödelisation in the λ-Calculus

Matematisk Institut
Aarhus Universitet
Trykkeriet

Basic Research in Computer Science

Gödelisation in the λ-Calculus

Mayer Goldberg

BRICS Report Series RS-95-38

ISSN 0909-0878 July 1995

GÖDELISATION IN THE λ-CALCULUS *

Mayer Goldberg
Computer Science Department
Indiana University †
(mayer@cs.indiana.edu)

July 10, 1995

Abstract

Gödelisation is a meta-linguistic encoding of terms in a language. While it is impossible to define an operator in the λ-calculus which encodes all closed λ-expressions, it is possible to construct restricted versions of such an encoding operator. In this paper, we propose such an encoding operator for proper combinators.

1 Prerequisites and Notation

We assume some familiarity with the untyped and simply typed λ-calculi [1, 3]. The set of all terms *generated* by $\{M_1, \ldots, M_n\}$ is $\{M_1, \ldots, M_n\}^+$ [1, Item 8.1.1 (i), Page 165]. The set of all λ-terms is denoted by Λ, the set of all closed λ-terms (combinators) is denoted by Λ^0. When n ranges over the integers, $\ulcorner n \urcorner$ denotes the n-th Church numeral, and when M ranges over all λ-expressions, $\ulcorner M \urcorner$ denotes an encoding of M. The Church successor function is denoted by S^+. The identity combinator is denoted by $\mathbf{I}$. The ordered n-tuple is denoted by $[x_1, \ldots, x_n]$, and the k-th projection function on an n-tuple is denoted by π_k^n. Since the definition of an ordered n-tuple plays a rôle in the proof of Theorem 3.2, we give their definitions below:

$$\begin{array}{rcll} [x_1, \ldots, x_n] & = & \lambda x.(x\ x_1 \cdots x_n) & \text{The ordered } n\text{-tuple.} \\ \pi_k^n & = & \lambda t.(t\ (\lambda x_1 \cdots x_n.x_k)) & \text{The } k\text{-th projection.} \end{array}$$

*This work was completed while visiting BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

†Bloomington, IN 47405, USA.

2 Introduction

Gödelisation[1] is an effective injection that is used to encode terms in a language[1, Item 6.5.6, Page 143] . It is possible to write a combinator **Gödel** in the λ-calculus, such that

$$(\mathbf{G\ddot{o}del}\ \ulcorner M \urcorner) = \ulcorner \ulcorner M \urcorner \urcorner.$$

Gödel is the same as Barendregt's **Num** combinator [1, Item 6.5.9, Page 143]. **Gödel** does not map λ-expressions to their encodings, but rather encodings of λ-expressions to the encodings of their encodings. Indeed, it is impossible to define a combinator that maps λ-expressions to their encodings:

2.1 PROPOSITION: *Gödelisation is necessarily a meta-linguistic notion in the λ-calculus.* There exists no combinator G that satisfies:

$$(G\ M) = \ulcorner M \urcorner, \text{ for any } \lambda\text{-expression } M$$

Proof: $G = \lambda m.G'$. Let $\omega = \lambda x.(x\ x)$. We know that $(\omega\ \omega) \rightarrow (\omega\ \omega)$. Applying G to $(\omega\ \omega)$ we get: $(G\ (\omega\ \omega)) = G'[m := (\omega\ \omega)]$. If m does not occur free in G' then G is not an injection. If m occurs free in G' then $(G\ (\omega\ \omega))$ has no normal form. ■

Since an encoding function for arbitrary λ-terms does not exist, let us consider a weaker notion, that of a *partial Gödeliser:*

2.2 DEFINITION: *A Partial Gödeliser.* Given a set S of combinators, we associate with each $M \in S$ a λ-expression I_M (which is taken to be "information about M"). A λ-expression G_S is said to be a *partial Gödeliser for S* if for each $M \in S$ we have:

$$(G_S\ M\ I_M) = \ulcorner M \urcorner$$

To the best of our knowledge, the only partial Gödeliser in the λ-calculus is due to Berger and Schwichtenberg [2], and encodes simply-typed λ-expressions, given an encoding of their type. Given a simply typed λ-expression M of type $\ulcorner \tau \urcorner$, we have:

$$(G_{\mathrm{ST}}\ M\ \ulcorner \tau \urcorner) = \ulcorner M \urcorner$$

[1]Gödelization takes its name from Gödel's proof technique in his paper "On formally undecidable propositions of Principia Mathematica and related systems" [5].

In the next section we derive a partial Gödeliser for the set of all proper combinators. This result is a part of our Ph.D. thesis [6].

3 Gödelisation of Proper Combinators

3.1 DEFINITION: *Proper Combinators* [1, Page 184, Problem 8.5.15], $\mathbf{PC}(n)$. A *proper combinator* of size n is a λ-expression $\lambda x_1 \cdots x_n.B$ where $B \in \{x_1, \ldots, x_n\}^+$. The set of all proper combinators of size n is $\mathbf{PC}(n)$.

Note that some proper combinators are not simply typed. For example $(\lambda x.xx)$ has no simple type.

3.2 THEOREM: *There exists G_{PC} such that for any $n \geq 1$ and proper combinator $P \in \mathbf{PC}(n)$ we have:*

$$(G_{\mathrm{PC}}\ P\ \ulcorner n \urcorner) = \ulcorner P \urcorner$$

Proof: We assume the existence of combinators **Var**, **Abs**, and **App** for encoding variables, abstractions and applications. Specifically:

$$\begin{aligned}(\mathbf{Var}\ \ulcorner n \urcorner) &= \ulcorner x_n \urcorner \\ (\mathbf{Abs}\ \ulcorner x_n \urcorner\ \ulcorner M \urcorner) &= \ulcorner (\lambda x_n.M) \urcorner \\ (\mathbf{App}\ \ulcorner M \urcorner\ \ulcorner N \urcorner) &= \ulcorner (M\ N) \urcorner\end{aligned}$$

By defining **Var**, **Abs**, and **App** appropriately, we can obtain encodings of λ-expressions in terms of integers, lists, strings, or any other data structure we might want to work with. For example, in Appendix A, we use S-expressions to encode variables, abstractions and applications in Scheme.

Given the representation of ordered pairs described in Section 1, let us note that for any λ-expressions R, a, b, we have

$$\begin{aligned}([R,a]\ [R,b]) &= ((\lambda x.(x\ R\ a))\ (\lambda x.(x\ R\ b))) \\ &= ((\lambda x.(x\ R\ b))\ R\ a) \\ &= (R\ R\ b\ a).\end{aligned}$$

In particular, if $R = \lambda rba.[r, (\mathbf{App}\ a\ b)]$,

$$([R, \ulcorner M \urcorner]\ [R, \ulcorner N \urcorner]) = [R, \ulcorner (M\ N) \urcorner].$$

Now pick a proper combinator in $\mathbf{PC}(n)$, $P = \lambda x_1 \cdots x_n.B$, where $B \in \{x_1, \ldots, x_n\}^+$. We obtain $\ulcorner B \urcorner$ as follows:

$$(B\ [(\lambda rba.[r, (\mathbf{App}\ a\ b)])], (\mathbf{Var}\ \ulcorner 1 \urcorner)]$$
$$\vdots$$
$$[(\lambda rba.[r, (\mathbf{App}\ a\ b)])], (\mathbf{Var}\ \ulcorner n \urcorner)]) = \ulcorner B \urcorner$$

This solves the main problem in defining G_{PC}, i.e., the construction of the body of a proper combinator of size n. What remains is to wrap encodings of abstractions of the n variables around the encoding of the body. The technique we use is similar to that by Church to derive a definition for P^- [3, Chapter III, §9, Page 31]. Thus:

$$\begin{aligned} G_{\mathrm{PC}} = \lambda pm.((\lambda t.(\pi_3^3\ t\ (\pi_2^2\ (\pi_2^3\ t)))) \\ (n\ (\lambda t.[(S^+\ (\pi_1^3\ t)) \\ (\pi_2^3\ t\ [(\lambda vnm.[v, (\mathbf{App}\ m\ n)]), \\ (\mathbf{Var}\ (\pi_1^3\ t))]), \\ (\lambda x.(\pi_3^3\ t\ (\mathbf{Abs}\ (\mathbf{Var}\ (\pi_1^3\ t))\ x))))]) \\ [\ulcorner 1 \urcorner, p, \mathbf{I}])) \end{aligned}$$

■

The above definition for G_{PC} can be translated directly into the programming language Scheme [4], as is done in Appendix A. A sample run is given in Appendix B.

Note: It is a simple exercise to replace Church numerals in the definition of G_{PC} to use the built-in representation of integers in Scheme. Our Scheme program, however, is faithful to the definition of G_{PC} in the λ-calculus.

4 Conclusion

Proposition 2.1, shows that no Gödeliser for Λ^0 exists, that takes no additional information about the expression it is encoding. Consequently, We consider *partial Gödelisers*, operating on specific subsets of Λ^0 and taking some information about the expressions they are encoding. Berger and Schwichtenberg [2] have constructed a partial Gödeliser which encodes simply-typed λ-expressions, given an encoding of their type. We have shown that a partial Gödeliser also exists for proper combinators, given their size.

Acknowledgements

I am grateful to BRICS[2] for hosting me this summer and for providing a stimulating environment. Thanks are also due to Olivier Danvy, Daniel P. Friedman, Julia L. Lawall, and Larry Moss for their comments and encouragement.

References

[1] Hendrik P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1984.

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[3] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[4] William Clinger and Jonathan Rees (editors). Revised[4] report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[6] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus.* PhD thesis, Department of Computer Science, Indiana University, December, 1995.

A Scheme Code

```
;;; The Identity combinator:
(define I (lambda (x) x))
```

[2] Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

```
;;; Routines to facilitate Church-numeral arithmetic:

(define Church-zero (lambda (x) (lambda (y) y)))
(define Church-S+
  (lambda (cn)
    (lambda (x)
      (lambda (y)
        (x ((cn x) y))))))
(define Church-one (Church-S+ Church-zero))
(define integer->Church
  (lambda (n)
    (if (zero? n)
        Church-zero
        (Church-S+ (integer->Church (sub1 n))))))
(define Church->integer (lambda (cn) ((cn add1) 0)))

;;; The definition of Var, Abs, and App using S-expressions for
;;; encoding proper combinators:

(define Var
  (lambda (n)
    (string->symbol (format "x~a" (Church->integer n)))))
(define Abs (lambda (v e) (list 'lambda (list v) e)))
(define App (lambda (f x) (list f x)))

;;; Support for ordered pairs:

(define make-pair (lambda (a b) (lambda (s) ((s a) b))))
(define pair->1 (lambda (p) (p (lambda (a) (lambda (b) a)))))
(define pair->2 (lambda (p) (p (lambda (a) (lambda (b) b)))))

;;; Support for ordered triples:

(define make-triple (lambda (a b c) (lambda (s) (((s a) b) c))))
(define triple->1
  (lambda (t) (t (lambda (a) (lambda (b) (lambda (c) a))))))
(define triple->2
  (lambda (t) (t (lambda (a) (lambda (b) (lambda (c) b))))))
(define triple->3
  (lambda (t) (t (lambda (a) (lambda (b) (lambda (c) c))))))
```



```
;;; The Gödeliser for proper combinators, from Theorem 3.2

(define Gpc
  (lambda (p n)
    ((lambda (t) ((triple->3 t) (pair->2 (triple->2 t))))
     ((n (lambda (t)
           (make-triple
             (Church-S+ (triple->1 t))
             ((triple->2 t) (make-pair
                              (lambda (v)
                                (lambda (n)
                                  (lambda (m)
                                    (make-pair v (App m n)))))
                              (Var (triple->1 t))))
             (lambda (x)
               ((triple->3 t) (Abs (Var (triple->1 t)) x))))))
      (make-triple Church-one p I)))))
```

B Scheme Session

```
> (load "pc.scm")

;;; Defining the proper combinator λxyz.(x x (y y)(y y (z z))),
;;; which is not simply typed:

> (define foo
    (lambda (x)
      (lambda (y)
        (lambda (z)
          (((x x) (y y)) ((y y) (z z)))))))

;;; foo denotes a procedure:

> foo
#<procedure foo>

;;; Encoding foo into a list:

> (Gpc foo (integer->Church 3))
(lambda (x1)
  (lambda (x2)
    (lambda (x3)
      (((x1 x1) (x2 x2)) ((x2 x2) (x3 x3))))))
>
```

Recent Publications in the BRICS Report Series

RS-95-38 Mayer Goldberg. *Gödelisation in the λ-Calculus.* July 1995. 7 pp.

RS-95-37 Sten Agerholm and Mike Gordon. *Experiments with ZF Set Theory in HOL and Isabelle.* July 1995. 14 pp. To appear in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS, 1995.

RS-95-36 Sten Agerholm. *Non-primitive Recursive Function Definitions.* July 1995. 15 pp. To appear in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS, 1995.

RS-95-35 Mayer Goldberg. *Constructing Fixed-Point Combinators Using Application Survival.* June 1995. 14 pp.

RS-95-34 Jens Palsberg. *Type Inference with Selftype.* June 1995. 22 pp.

RS-95-33 Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. *Type Inference with Non-structural Subtyping.* June 1995. 22 pp.

RS-95-32 Jens Palsberg. *Efficient Inference of Object Types.* June 1995. 32 pp. To appear in *Information and Computation.* Preliminary version appears in *Ninth Annual IEEE Symposium on Logic in Computer Science*, LICS '94 Proceedings, pages 186–195.

RS-95-31 Jens Palsberg and Peter Ørbæk. *Trust in the λ-calculus.* June 1995. 32 pp. To appear in *Static Analysis: 2nd International Symposium*, SAS '95 Proceedings, 1995.

RS-95-30 Franck van Breugel. *From Branching to Linear Metric Domains (and back).* June 1995. 30 pp. Abstract appeared in Engberg, Larsen, and Mosses, editors, *6th Nordic Workshop on Programming Theory*, NWPT '6 Proceedings, 1994, pages 444-447.

RS-95-29 Nils Klarlund. *An $n \log n$ Algorithm for Online BDD Refinement.* May 1995. 20 pp.