Gödelisation in the untyped lambda calculus

Torben Æ. Mogensen
DIKU, University of Copenhagen, Denmark
email: torbenm@diku.dk

Abstract

It is well-known that one cannot inside the pure untyped lambda calculus determine equivalence. I.e., one cannot determine if two terms are beta-equivalent, even if they both have normal forms. This implies that it is impossible in the pure untyped lambda calculus to do Gödelisation, i.e. to write a function that can convert a term to a representation of (the normal form of) that term, as equivalence of normal-form terms is decidable given their representation. If the lambda calculus is seen as a programming language, this means that you can't from the value of a function find its text.

Things are different for simply typed lambda calculus: Berger and Schwichtenberg showed that, given its type, it is possible to convert a function into a representation of its normal form. This was termed "an inverse to the evaluation function", as it turns values into representations. However, the main purpose was for normalising terms. Similarly, Goldberg has shown that for a subset (proper combinators) of the pure untyped lambda calculus, Gödelisation is possible. However, the Gödeliser itself is not a proper combinator, though it (as all closed lambda terms) can be written by combining proper combinators.

In this paper, we investigate Gödelisation for the full untyped lambda calculus. To overcome the theoretical impossibility of this, we extend the lambda calculus with a feature that allows limited manipulation of extensional aspects: A finite set of labels on lambda terms and a predicate for comparing these. Within this extended lambda calculus, we can convert terms in the subset corresponding to normal form terms in the classical lambda calculus into their representation.

The extension of the lambda calculus (we conjecture) retains the Church-Rosser property. This implies that Gödelisation must yield identical results for beta-equivalent terms. We show only that terms in normal form Gödelise to their representation, but the implication is that any term that has a normal form will Gödelise to a representation of its normal form. Hence, Gödelisation can be used as a tool for normalising lambda terms.

1 Introduction

There are various ways to represent lambda terms as "data" inside the lambda calculus. One is to represent the term by its Gödel number and then represent that number inside the lambda calculus by e.g. a Church-numeral. More tractable representations can also be used, see e.g. Mogensen's papers [6], [7]. The representations used in these papers use the notion of *higher order abstract syntax* [9]. In essence, this means that variable bindings are represented by variable bindings. Given three *constructors*, *VAR*, *APP* and *ABS*, we can represent lambda terms by the following scheme:

$$\begin{array}{lcl} \lceil x \rceil & \equiv & VAR(x) \\ \lceil \lambda x.E \rceil & \equiv & ABS(\lambda x.\lceil E \rceil) \\ \lceil E_1\ E_2 \rceil & \equiv & APP(\lceil E_1 \rceil, \lceil E_2 \rceil) \end{array}$$

The constructors can be expressed in the lambda calculus in a way that allow operations on syntax, including alpha-equivalence testing.

The goal of this paper is to construct a lambda calculus term G such that $G\ E \longrightarrow \lceil E \rceil$ if E is in normal form. Equivalently (due to confluence), we can say that G takes a term and produces the representation of its normal form (if such exist). However, such a term G does provably not exist (see section 6.6 of Barendregts book on the lambda calculus [1]). Hence, we must relax the condition somewhat.

Mayer Goldberg [5] relaxes the condition by restricting the class of terms E that G works for to be the set of proper combinators. Berger and Schwichtenberg [2] relax the condition by requiring E to be in the simply typed lambda calculus and that the type of E is given.

Instead of restricting the set of terms that can be Gödelised, we want our Gödeliser to be able to take any normalizing closed lambda term and return a representation of its normal form. To obtain this we allow G to be written in an extension of the lambda calculus. G can not Gödelise all terms in the extended calculus, but it can do so for all closed normalising terms in the classical lambda calculus.

2 An extended lambda calculus

We extend the classical lambda calculus with labels: Each lambda abstraction is given a label. The labelling is not unique; different abstractions can share the same label. Indeed, we only need 3 different labels. To make the labelling visible we introduce a way of inspecting labels. The syntax of the extended lambda calculus is

$$
\begin{array}{lcll}
\Lambda^L & \rightarrow & x & \text{variable} \\
 & | & \lambda^l x.\Lambda^L & \text{labelled abstraction} \\
 & | & \Lambda^L_1\ \Lambda^L_2 & \text{application} \\
 & | & l?\Lambda^L_1\ \Lambda^L_2\ \Lambda^L_3 & \text{label inspection}
\end{array}
$$

We have the following reduction rules for the extended lambda calculus:

$$
\begin{array}{lcll}
(\lambda^l.E_1)\ E_2 & \longrightarrow & E_1[x \setminus E_2] & (\beta) \\
l?(\lambda^l x.E_1)\ E_2\ E_3 & \longrightarrow & E_2 & (L1) \\
l?(\lambda^{l'} x.E_1)\ E_2\ E_3 & \longrightarrow & E_3 \ \text{ if } l \neq l' & (L2)
\end{array}
$$

The (β) rule is the usual beta-reduction rule. $(L1)$ and $(L2)$ handle inspection of labels: If the first term is in weak head normal form and its label matches the label that is tested for, the second term is selected. If its label does not match, the third term is selected.

Reduction in the extended lambda calculus is strongly believed to be confluent, but at the moment we have not looked at proving this.

3 Gödelisation

As the basis of our Gödeliser we use the Gödeliser for the typed lambda calculus by Berger and Schwichtenberg [2], but using a notation similar to the extension of this work found in Danvy's type-directed partial evaluators [3]. In this, Gödelisation is defined by a pair of type-indexed functions $\downarrow_t$ and $\uparrow^t$, where $\downarrow_t$ takes a value of type t and produces an expression of type t and $\uparrow^t$ takes an expression of type t and produces a value of type t. $\downarrow_t$ and $\uparrow^t$ are mutually recursively defined by

$$
\begin{array}{lcl}
\downarrow_b\ v & = & v \\
\downarrow_{t_1 \rightarrow t_2}\ v & = & ABS(\lambda\ x : t_1.\ \downarrow_{t_2}\ (v\ (\uparrow^{t_1}\ (VAR(x))))) \\
 & & \text{where } x \text{ is a fresh variable}
\end{array}
$$

$$
\begin{array}{lcl}
\uparrow^b\ e & = & e \\
\uparrow^{t_1 \rightarrow t_2}\ e & = & \lambda\ x : t_1.\ \uparrow^{t_2}\ (APP(e,\ (\downarrow_{t_1}\ x)))
\end{array}
$$

If $\downarrow_t$ is applied to a closed term of type t, the representation of the normal form of that term is produced. The constructors VAR, APP and ABS are like those described in section 1, suitably modified to handle typed terms.

3.1 Untyped Gödelisation

As can be seen, the functions $\downarrow_t$ and $\uparrow^t$ use the type t to select between two actions: Either returning the argument unchanged or doing what can be seen as a two-level eta-expansion of the argument [4][3].

In the untyped world we don't have any type argument to base this selection of actions on. So when do we want to return the argument unchanged and when do we want to eta-expand?

Initially, the $\downarrow$ function will be applied to the term we wish to Gödelise. In this situation, we surely want to eta-expand to get the representation of the top-level abstraction[1]. But we also apply the $\downarrow$ function to the body of the abstraction we build in the representation of the term. This body is obtained by, in the original body, substituting the bound variable by the result of applying $\uparrow$ to the representation of a variable. If the function we Gödelise is $\lambda x.x$, we will hence apply $\downarrow$ to $\uparrow(VAR(x))$. In this situation we want to return $VAR(x)$ directly, in essence letting $\downarrow$ and $\uparrow$ cancel.

This is in fact the general idea: Whenever we apply $\downarrow$ to something produced by $\uparrow$, we let these cancel. Otherwise, we eta-expand.

We can use the labels and label testing capability of our extended lambda calculus to facilitate this: We let the results of applying $\uparrow$ use labels different from those used in the term we want to Gödelise. Now, $\downarrow$ can use the label to decide its action: If the label indicates that the argument is the result of applying $\downarrow$, it "undoes" the $\downarrow$ operation (cancelling the $\uparrow$ and $\downarrow$ operations), otherwise it does the eta-expansion. If we assume we use the label 1 as label for the results of applying $\uparrow$, we can write this as

$$
\downarrow\ v\ =\ 1?v\ (cancel\ v)\ (ABS(\lambda^0 x.\downarrow(v\ (\uparrow(VAR(x))))))
$$

The remaining problem is how we can make $\uparrow$ cancelable, i.e. how to program $\uparrow$ and *cancel*. We first look at a normal (not canceled) use of a value returned by $\uparrow$. This is inside the $\downarrow$ function, when it is used as an argument to the original term that we want to Gödelise. The original term might use this as a function or it might return it. We have already covered the latter case. The former case uses the eta-expansion done by $\uparrow$. Since we can not in advance know how the result of $\uparrow$ is used, we must assume that the eta-expansion is necessary and hence let $\uparrow$ do this always, making our first attempt at $\uparrow$ be

$$
\uparrow\ e\ = \lambda^1 x.\uparrow(APP(e,\ (\downarrow x)))
$$

However, this eta-expansion can not in general be undone, as any argument we give to it just produces another eta-expansion and so on *ad infinitum*. However, we can use the argument to the eta-expanded term as a signal that selects between undoing the last eta-expansion and doing another. We can use labels and label testing again for this purpose: We let $(cancel\ v)$ pass v an argument with a special label. The abstraction that is the result of $\uparrow$ will test for this label in its input and when it gets this it will undo the last eta-expansion. If we use 2 as this special signal-label, we get the final versions of $\downarrow$ and $\uparrow$:

$$
\begin{array}{lcl}
\downarrow\ v & = & 1?v\ (v\ \lambda^2 a.a)\ (ABS(\lambda^0 x.\downarrow(v\ (\uparrow(VAR(x)))))) \\
\uparrow\ e & = & \lambda^1 x.2?x\ e\ (\uparrow(APP(e,\ (\downarrow x))))
\end{array}
$$

We can then encode these mutually recursive functions by using Y-combinators:

$$
\begin{array}{lcl}
\downarrow & \equiv & Y\ (\lambda d.(\lambda u.D)\ (Y\ (\lambda u.U))) \\
 & & \text{where} \\
D & \equiv & \lambda v.1?v\ (v\ \lambda^2 a.a)\ (ABS(\lambda^0 x.d\ (v\ (u\ (VAR(x)))))) \\
U & \equiv & \lambda e.\lambda^1 x.2?x\ e\ (u\ (APP(e,\ (d\ x))))
\end{array}
$$

We have omitted the labels for the abstractions used in this encoding. We can use any label for these, as they will never get to a position where they are tested. Hence, we need only a total of three labels: 0 for use in the input term, 1 to designate results of $\uparrow$ and 2 to denote the special signal value. We can e.g. use 0 for all remaining abstractions.

For ease of reading, we will in the following use the mutually recursive definition of the functions.

As an example, figure 1 shows Gödelisation of $\lambda ab.a\ b$.

[1] Since we work with closed terms, we are sure that any normal-form term *will* have a top-level abstraction.

$\downarrow (\lambda^0 a.\lambda^0 b.a\ b)$

$\longrightarrow$ $1?(\lambda^0 a.\lambda^0 b.a\ b)$
$((\lambda^0 a.\lambda^0 b.a\ b)\ \lambda^2 a.a)$
$(ABS(\lambda^0 x.\downarrow((\lambda^0 a.\lambda^0 b.a\ b)\ (\uparrow(VAR(x))))))$

$\longrightarrow$ $ABS(\lambda^0 x.\downarrow((\lambda^0 a.\lambda^0 b.a\ b)\ (\uparrow(VAR(x)))))$

$\longrightarrow$ $ABS(\lambda^0 x.\downarrow(\lambda^0 b.\uparrow(VAR(x))\ b))$

$\longrightarrow$ $ABS(\lambda^0 x.(1?(\lambda^0 b.\uparrow(VAR(x))\ b)$
$((\lambda^0 b.\uparrow(VAR(x))\ b)\ \lambda^2 a.a)$
$(ABS(\lambda^0 y.\downarrow((\lambda^0 b.\uparrow(VAR(x))\ b)\ (\uparrow(VAR(y)))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^0 b.\uparrow(VAR(x))\ b)\ (\uparrow(VAR(y)))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\uparrow(VAR(x)))\ (\uparrow(VAR(y)))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^1 z.2?z\ (VAR(x)))\ (\uparrow(APP(VAR(x),(\downarrow z)))))\ (\uparrow(VAR(y)))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^1 z.2?z\ (VAR(x)))\ (\uparrow(APP(VAR(x),(\downarrow z)))))$
$(\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(2?(\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))$
$(VAR(x))$
$(\uparrow(APP(VAR(x),(\downarrow(\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w))))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),(\downarrow(\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w))))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),$
$(1?(\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))$
$((\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\lambda^2 a.a))$
$(ABS(\lambda p.\downarrow((\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\uparrow(VAR(p))))))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),$
$((\lambda^1 w.2?w\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\lambda^2 a.a))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),$
$(2?(\lambda^2 a.a)\ (VAR(y)))\ (\uparrow(APP(VAR(y),(\downarrow(\lambda^2 a.a))))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),(VAR(y))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.$
$(1?(\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))$
$((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\lambda^2 a.a))$
$(ABS(\lambda^0 z.(\downarrow((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\uparrow(VAR(z))))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.$
$((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\lambda^2 a.a)))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.$
$(2?(\lambda^2 a.a)$
$(APP(VAR(x),(VAR(y))))$
$(\uparrow(APP(VAR(x),(VAR(y))))))))))$

$\longrightarrow$ $ABS(\lambda^0 x.(ABS(\lambda^0 y.(APP(VAR(x),(VAR(y))))))))$

$\equiv$ $\lceil\lambda^0 x.\lambda^0 y.x\ y\rceil$

Figure 1: Example of Gödelisation

We prove lemma 2 by induction over the structure of N and D. The induction hypothesis is the statement of lemma 2: For $N \in \Lambda^N$, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$ and for $D \in \Delta$, $\overline{D} \longrightarrow^* \uparrow (\lceil D \rceil)$.

$N \equiv \lambda^0 x.N_1$:

$$
\begin{array}{lll}
 & \downarrow (\overline{\lambda^0 x.N_1}) & \\
\longrightarrow & 1?(\overline{\lambda^0 x.N_1})\ ((\overline{\lambda^0 x.N_1})\ (\lambda^2 a.a))\ (ABS(\lambda^0 x.\downarrow ((\overline{\lambda^0 x.N_1})\ (\uparrow (VAR(x))))))) & \text{by def. of } \downarrow \\
\longrightarrow & ABS(\lambda^0 x.\downarrow ((\overline{\lambda^0 x.N_1})\ (\uparrow (VAR(x))))) & \text{by } (L2) \\
\equiv & ABS(\lambda^0 x.\downarrow (((\lambda^0 x.N_1)[x_i \backslash \uparrow (VAR(x_i))])\ (\uparrow (VAR(x))))),\ x_i \in FV(\lambda^0 x.N_1) & \text{by def. of } \overline{\ } \\
\longrightarrow & ABS(\lambda^0 x.\downarrow (N_1[x_i \backslash \uparrow (VAR(x_i))][x \backslash \uparrow (VAR(x))])),\ x_i \in FV(\lambda^0 x.N_1) & \text{by } (\beta) \\
\equiv & ABS(\lambda^0 x.\downarrow (N_1[x_i \backslash \uparrow (VAR(x_i))])),\ x_i \in FV(N_1) & \\
\equiv & ABS(\lambda^0 x.\downarrow (\overline{N_1})) & \text{by def. of } \overline{\ } \\
\longrightarrow^* & ABS(\lambda^0 x.\lceil N_1 \rceil) & \text{by induction} \\
\equiv & \lceil \lambda^0 x.N_1 \rceil &
\end{array}
$$

$N \equiv D \in \Delta$:

$$
\begin{array}{lll}
 & \downarrow (\overline{D}) & \\
\longrightarrow^* & \downarrow (\uparrow (\lceil D \rceil)) & \text{by induction} \\
\longrightarrow^* & \lceil D \rceil & \text{by lemma 1}
\end{array}
$$

$D \equiv x$:

$$
\begin{array}{lll}
 & \overline{x} & \\
\equiv & \uparrow (VAR(x)) & \text{by def. of } \overline{\ } \\
\equiv & \uparrow (\lceil x \rceil) &
\end{array}
$$

$D \equiv D_1\ N_1$:

$$
\begin{array}{lll}
 & \overline{D_1\ N_1} & \\
\equiv & \overline{D_1}\ \overline{N_1} & \\
\longrightarrow^* & \uparrow (\lceil D_1 \rceil)\ \overline{N_1} & \text{by induction} \\
\longrightarrow & 2?\overline{N_1}\ \lceil D_1 \rceil\ (\uparrow (APP(\lceil D_1 \rceil, \downarrow (\overline{N_1})))) & \text{by def. of } \uparrow \\
\longrightarrow & \uparrow (APP(\lceil D_1 \rceil, \downarrow (\overline{N_1}))) & \text{by } (L2) \\
\longrightarrow^* & \uparrow (APP(\lceil D_1 \rceil, \lceil N_1 \rceil)) & \text{by induction} \\
\equiv & \uparrow (\lceil D_1\ N_1 \rceil) &
\end{array}
$$

□

Figure 2: Proof of lemma 2

4 Proof of correctness

In this section we will prove the correctness of the Gödeliser.

We start by proving that $\downarrow$ and $\uparrow$ cancel in the expected way, which we state in

Lemma 1 *For all $E \in \Lambda^L$, $\downarrow (\uparrow E) \longrightarrow^* E$.*

This is simple to prove:

$$
\begin{array}{lll}
\downarrow (\uparrow E) & \longrightarrow & \downarrow (\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x)))) \\
 & \longrightarrow & 1?(\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x)))) \\
 & & ((\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x))))\ (\lambda^2 a.a)) \\
 & & (ABS(\lambda^0 y.\downarrow (\cdots))) \\
 & \longrightarrow & ((\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x))))\ (\lambda^2 a.a)) \\
 & \longrightarrow & 2?(\lambda^2 a.a)\ E\ (\uparrow (APP(E, \downarrow (\lambda^2 a.a)))) \\
 & \longrightarrow & E
\end{array}
$$

□

We next define the input to the Gödeliser: Lambda terms in normal form with label 0 on all abstractions and not containing label tests:

$$
\begin{array}{lll}
\Lambda^N & \rightarrow & \lambda^0 x.\Lambda^N \\
 & | & \Delta \\
 & & \\
\Delta & \rightarrow & x \\
 & | & \Delta\ \Lambda^N
\end{array}
$$

Note that this includes open terms. We will need to handle open terms in a lemma below, even though the input to the Gödeliser is assumed to be closed.

We now define

$$\overline{N} \equiv N[x_i \backslash \uparrow (VAR(x_i))],\ x_i \in FV(N)$$

where $FV(N)$ is the set of free variables of N. Hence, $\overline{N}$ replaces all free variables of N by $\uparrow$ applied to the representations of the variables. Note that for closed N, $\overline{N} = N$.

We continue with the central lemma of our proof:

Lemma 2 *For $N \in \Lambda^N$, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$ and for $D \in \Delta$, $\overline{D} \longrightarrow^* \uparrow (\lceil D \rceil)$.*

The proof of lemma 2 can be found in figure 2.

We can now state the correctness theorem

Theorem 3 *If $N \in \Lambda^N$ and N is closed, then $\downarrow N \longrightarrow^* \lceil N \rceil$.*

The proof is simple: Since N is closed, $N \equiv \overline{N}$ and by lemma 2, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$.

□

5 Implementation

The Gödeliser has been implemented in Scheme, where it has been used to "decompile" functions. Scheme doesn't have labels and label testing, but it does have pointer equality tests. While not quite equivalent to label testing, it has in conjunction with some of Scheme's non-functional features been sufficient to emulate the label testing needed in the Gödeliser. We will in this paper just show the program text (in figure 3) of the Scheme implementation and refer to another paper [8] for more details. Note that we have extended the $\downarrow$-part of the Gödeliser to work with base-type values. Hence, terms containing base-type values can be reified.

The call-by-value nature of Scheme makes the implementation unable to Gödelise terms that do not reduce to normal form under call-by-value reduction.

6 Discussion

While the extended lambda calculus is able to Gödelise classical lambda terms, it is not self-Gödelisable. For example, it is not possible inside the calculus to distinguish $(\lambda^0 x.x)$ from $(\lambda^0 x.1?x\ x\ x)$. It will be interesting to study what extensions are needed to the lambda calculus to make it fully self-Gödelisable, short of adding Gödelisation as a primitive operation. On a related issue, can we make smaller extensions of the classical lambda calculus than we have in this paper and still get Gödelisation of the classical fragment of this calculus? In other words, how many of the properties of the classical lambda calculus can we retain while allowing Gödelisation of the classical fragment?

While the extended lambda calculus inherits many properties of the classical lambda calculus, for example (we conjecture) confluence, it does not inherit all of them. As an example, eta-reduction is not valid in the extended calculus. Indeed, $(\lambda^0 x.x)$ and $(\lambda^0 x.(\lambda^0 y.x\,y))$ are Gödelised to representations that can be distinguished even in the classical lambda calculus.

In [10], it is shown that adding explicit Gödelisation (by reification) to the lambda calculus makes textual identity the only valid equivalence. By retaining beta-equivalence, we feel that our extension is less disruptive and more useful than explicit Gödelisation. In particular, it allows the Gödeliser to be used for normalisation of terms.

The Gödeliser has some similarities to Mogensen's self-reducer for the lambda calculus [6], and was indeed partly derived from this. The $\downarrow$ function in the Gödeliser corresponds to the R' function in the self-reducer while the $\uparrow$ function corresponds to the P function in the self-reducer. Where the P function in the self-reducer builds a pair of two values, the $\uparrow$ function in the Gödeliser builds a function that selects between two values based on the form of the argument. This isn't too far from how pairs are traditionally represented in the lambda calculus.

```
(define (downarrow v)
  (cond
   ((number? v) v)
   ((boolean? v) v)
   ((char? v) v)
   ((string? v) v)
   ((vector? v) v)
   ((symbol? v) (list 'quote v))
   ((null? v) v)
   ((pair? v)
     (list 'cons (downarrow (car v))
                 (downarrow (cdr v))))
   ((procedure? v)
     (if (memq v registered) (v special)
        (let ((x (gensym)))
          (list 'lambda (list x)
                 (downarrow (v (uparrow x))))))))))

(define (uparrow e)
  (let
    ((f (lambda (v)
          (if (eq? v special) e
              (uparrow (list e (downarrow v)))))))
    (set! registered (cons f registered))
    f))

(define registered '())

(define special '(special))

(define count 0)

(define (gensym)
  (set! count (+ 1 count))
  (string->symbol
    (string-append "x" (number->string count))))

(define (goedelise v)
  (set! count 0)
  (set! registered '())
  (downarrow v))
```

Figure 3: Scheme implementation of Gödeliser

7 Conclusion

We have presented an extension to the lambda calculus which allows Gödelisation of terms from the subset that corresponds to the classical lambda calculus. We have shown this by developing a Gödeliser and proving it correct.

Since the extensions can be modeled by the standard non-functional features of Scheme, the result can be used to make a decompiler (and partial evaluator) for a fragment of Scheme that includes the classical lambda calculus. When used as a partial evaluator or normaliser, the Scheme implementation of the Gödeliser performs call-by-value reduction to normal form, which is not a complete reduction strategy. However, this is a small limitation compared to the requirement that the residual programs must have normal forms.

References

[1] H. P. Barendregt. *The lambda Calculus, its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, New York, Oxford, 2 edition, 1984.

[2] U. Berger and H. 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. IEEE Computer Society Press, 1991.

[3] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257. ACM, 1996.

[4] O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–228, September 1995.

[5] Mayer Goldberg. Gödelisation in the λ-calculus (extended version). Technical Report RS-96-5, BRICS, 1996.

[6] T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[7] T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.

[8] T. Æ. Mogensen. Normalization for a subset of scheme. In O. Danvy and P. Dybjer, editors, *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation*. to appear, 1998.

[9] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM, ACM Press, 1988.

[10] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, (10):189–199, 1998.