Department of Computer Science
University of Essex, England
Technical Report CSM-343

A Deep Embedding of Z_C in Isabelle/HOL

Norbert Völker

July 25, 2001

Abstract

This report describes a deep embedding of the logic Z_C [HR00] in Isabelle/HOL. The development is based on a general theory of de Bruijn terms. Wellformed terms, propositions and judgements are represented as inductive sets. The embedding is used to prove elementary properties of Z_C such as uniqueness of types, type inhabitation and that elements of judgements are wellformed propositions

1 De Bruijn Terms

The representation of logical syntax in Isabelle/HOL will be based on a polymorphic datatype *dbterm* of de Bruijn terms. This development follows the example of A. Gordon [Gor94] who constructed a similar theory for the HOL system. The datatype *dbterm* is independent of Z_C and can be used as a foundation for deep embeddings in general. For other HOL representations of terms see [Owe95] and [Von95].

The definition of the datatype (α, β, γ) *dbterm* of de Bruijn terms is:

$$\begin{array}{rcl} (\alpha,\beta,\gamma)\ dbterm & = & Const\ \alpha \\ & | & Free\ \beta \\ & | & Bound\ \mathbb{N} \\ & | & dAbs\ ((\alpha,\beta,\gamma)\ dbterm)\ \gamma \\ & | & ((\alpha,\beta,\gamma)\ dbterm)\ \$\ ((\alpha,\beta,\gamma)\ dbterm) \end{array}$$

The five type constructors stand for constants, free variables, bound variables, abstraction and application. The type parameters represent constants (α), free variables (β) and types (γ). By suitable instantiation of these parameters, the datatype *dbterm* can be adapted to the representation of different

logic formalisms. In particular constants and variables can be explicitly typed or not.

In order to make function applications with several arguments more readable, a function *App* is introduced which applies a constant to a list of arguments:

$$\begin{array}{lcl} App & :: & [\alpha, (\alpha,\beta,\gamma)\ dbterm\ list] \ \rightarrow\ (\alpha,\beta,\gamma)\ dbterm \\ App\ a\ [\,] & = & Const\ a \\ App\ a\ (ts\ @\ [t]) & = & App\ a\ ts\ \$\ t \end{array}$$

The infix-operator @ concatenates two lists while # is the cons-operation on lists:

$$\begin{array}{lcl} op\ @ & :: & [\alpha\ list, \alpha\ list] \ \rightarrow\ \alpha\ list \\ [\,]\ @\ y & = & y \\ (a\ \#\ x)\ @\ y & = & a\ \#\ (x\ @\ y) \end{array}$$

An simple example for the use of *App* is:

$$App\ \text{“}f\text{“}\ [u, v] \quad = \quad Const\ \text{“}f\text{“}\ \$\ u\ \$\ v$$

As usual, binary operators such as \$ associate to the left unless explicitly noted otherwise.

1.1 Degree

The definition of de Bruijn terms permits badly formed terms which contain bound variables that do not refer to any abstractions. Well formed terms can be characterised with the help of a a primitive recursive *degree* function on terms:

$$\begin{array}{lcl} degree\ ::\ (\alpha,\beta,\gamma)\ dbterm & \rightarrow & \mathbb{N} \\ degree\ (Const\ c) & = & 0 \\ degree\ (Free\ v) & = & 0 \\ degree\ (Bound\ n) & = & n+1 \\ degree\ (dAbs\ t\ T) & = & degree\ t - 1 \\ degree\ (t\ \$\ t') & = & max\ (degree\ t)\ (degree\ t') \end{array}$$

In [Gor94] it is shown that the terms t with $degree\ t = 0$ are exactly those terms which result from the translation of name-carrying terms to de Bruijn terms. The importance of these terms is stressed by calling them *proper* terms.

1.2 Substitution

The main advantage of de Bruijn terms as compared to name-carrying terms is a simpler definition of substitution. Since bound and free variables are distinguished, there is no danger of variable captures. The definition of substitution is:

$$\begin{array}{lcl} (\_[\_/\_]) & :: & [(\alpha,\beta,\gamma)\ dbterm, \beta, (\alpha,\beta,\gamma)\ dbterm] \ \rightarrow (\alpha,\beta,\gamma) \\ t[v/u] & = & inst\ 0\ (abstract\ 0\ v\ t)\ u \end{array}$$

The two primitive recursive functions *abstract* and *inst* replace a free variable by a bound variable resp. instantiate a bound variable:

$$
\begin{array}{lcl}
\mathit{abstract} & :: & [\mathbb{N}, \beta, (\alpha, \beta, \gamma)\ \mathit{dbterm}] \to (\alpha, \beta, \gamma)\ \mathit{dbterm} \\
\mathit{abstract}\ i\ v\ (\mathit{Const}\ c) & = & \mathit{Const}\ c \\
\mathit{abstract}\ i\ v\ (\mathit{Free}\ w) & = & (\mathit{if}\ v = w\ \mathit{then}\ \mathit{Bound}\ i\ \mathit{else}\ \mathit{Free}\ w) \\
\mathit{abstract}\ i\ v\ (\mathit{Bound}\ n) & = & \mathit{Bound}\ n \\
\mathit{abstract}\ i\ v\ (\mathit{dAbs}\ t\ T) & = & \mathit{dAbs}\ (\mathit{abstract}\ (i+1)\ v\ t)\ T \\
\mathit{abstract}\ i\ v\ (t\ \$\ t') & = & (\mathit{abstract}\ i\ v\ t)\ \$\ (\mathit{abstract}\ i\ v\ t')
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{inst} & :: & [\mathbb{N}, (\alpha, \beta, \gamma)\ \mathit{dbterm}, (\alpha, \beta, \gamma)\ \mathit{dbterm}] \to (\alpha, \beta, \gamma)\ \mathit{dbterm} \\
\mathit{inst}\ i\ (\mathit{Const}\ c)\ u & = & \mathit{Const}\ c \\
\mathit{inst}\ i\ (\mathit{Free}\ w)\ u & = & \mathit{Free}\ w \\
\mathit{inst}\ i\ (\mathit{Bound}\ n)\ u & = & (\mathit{if}\ i = n\ \mathit{then}\ u\ \mathit{else}\ \mathit{Bound}\ n) \\
\mathit{inst}\ i\ (\mathit{dAbs}\ t\ T)\ u & = & \mathit{dAbs}\ (\mathit{inst}\ (i+1)\ t\ u)\ T \\
\mathit{inst}\ i\ (t\ \$\ t')\ u & = & (\mathit{inst}\ i\ t\ u)\ \$\ (\mathit{inst}\ i\ t'\ u)
\end{array}
$$

When reasoning about substitution, the set of free variables of a term plays an important role. This set is calculated by the primitive recursive function *frees*:

$$
\begin{array}{lcl}
\mathit{frees} :: (\alpha, \beta, \gamma)\ \mathit{dbterm} & \to & \beta\ \mathit{set} \\
\mathit{frees}\ (\mathit{Const}\ c) & = & \{\,\} \\
\mathit{frees}\ (\mathit{Free}\ v) & = & \{v\} \\
\mathit{frees}\ (\mathit{Bound}\ n) & = & \{\,\} \\
\mathit{frees}\ (\mathit{dAbs}\ t\ T) & = & \mathit{frees}\ t \\
\mathit{frees}\ (t\ \$\ t') & = & \mathit{frees}\ t\ \cup\ \mathit{frees}\ t'
\end{array}
$$

Types without free variables are called *closed*. Substitution of a variable v does not effect a proper term t if v does not occur (free) in t:

$$[|\ v \notin \mathit{frees}\ t;\ \mathit{degree}\ t\ =\ 0\ |] \quad \Longrightarrow \quad t[v/u] = t$$

This theorem is proven easily in HOL using mainly induction over type *dbterm* and simplification.

1.3 Introducing name-carrying syntax

Despite its technical disadvantages, name-carrying syntax has its attractions due to better readability. As shown in [Gor94], named variables can be simulated on top of de Bruijn terms by a constant *Abs*. In view of later applications, it is convenient to restrict the definition of *Abs* to instantiations of *dbterm* which correspond to terms with typed variables:

$$
\begin{array}{lcl}
\mathit{Abs} & :: & [\beta \times \gamma, (\alpha, \beta \times \gamma, \gamma)\ \mathit{dbterm}] \to (\alpha, \beta \times \gamma, \gamma)\ \mathit{dbterm} \\
\mathit{Abs}\ (s, T)\ t & = & \mathit{dAbs}\ (\mathit{abstract}\ 0\ (s, T)\ t)\ T
\end{array}
$$

By definition the term $(\mathit{Abs}\ v\ t)$ is equal to $(\mathit{dAbs}\ t')$ where t' is the abstraction of t over v, i.e. all occurrences of $(\mathit{Free}\ v)$ in t are replaced by $(\mathit{Bound}0)$. Hence, $(\mathit{Abs}\ (s, T)\ t)$ can be interpreted as an abstraction over a variable with name s and type T in the term t.

1.4 Variants

In many logical formalisms, there is a need for "fresh" free variables which do not clash with the free variables of other terms. This is always possible provided the set of variable names is infinite. When working with concrete variables, it is also desirable that the name of the variant bears some resemblance to the original variable name, say by adding a prime or a letter.

In HOL, these concepts can be expressed by an axiomatic type class `variant`. The types in this class feature an overloaded *variant* function which modifies an element x so that it is not member of a finite set A:

$$\begin{array}{l} \mathit{variant} \;::\; [\alpha :: \texttt{variant}, \alpha\ \mathit{set}] \to \alpha \\ \mathit{variant}\ x\ A \neq x \\ \mathit{finite}\ A \implies \mathit{variant}\ x\ A \notin A \end{array}$$

Variable names will be represented as elements of type

$$\mathit{string} \;=\; \mathit{char\ list}$$

A variant function can be defined on type *string* by repeatedly adding a character (say "a") to the end of a string until it is not member of a given finite set. With this definition, it can be proven that type *string* is in class `variant`:

$$\mathit{string} \;::\; \texttt{variant}$$

For product types, the function *variant* is defined in such a way that the second component does not change:

$$\mathit{snd}\ (\mathit{variant}\ (a, b)\ A) \;=\; b$$

Since variables are modelled as pairs consisting of a name and a type, this definition ensures that the variant of a variable has the same type as the original variable. The membership of a product type in class `variant` follows from the membership of the first argument in that class.

2 Type-homogenous Records in HOL

The HOL representation of schemas will be based on a type $(\alpha\ \mathit{rcd})$ of records with strings as labels and field values in some type α. It is defined as an instantiation of a more general type of bindings:

$$\alpha\ \mathit{rcd} \;=\; (\mathit{string}, \alpha)\ \mathit{bdg}$$

Note that these records are type-homogenous: all field values of a record $(r :: \alpha\ \mathit{rcd})$ are elements of the same HOL type α.

Using HOL's "typedef" mechanisms, the binding type $(\alpha, \beta)\ \mathit{bdg}$ was defined as a new type isomorphic to the set of all partial functions from α to β with finite domain.

$$(\alpha, \beta)\ \mathit{bdg} \;\cong\; \{f :: (\alpha \to \beta\ \mathit{option}).\ \mathit{finite}\ (\mathit{dom}\ f)\}$$

Two basic functions on bindings are:

$$\begin{array}{lcl} \mathit{dom\_bdg} & : & (\alpha,\beta)\,\mathit{bdg} \;\rightarrow\; \alpha\;\mathit{set} \\ \_\cdot\_ & :: & [(\alpha,\beta)\,\mathit{bdg},\alpha] \;\rightarrow\; \beta \end{array}$$

For a binding f, $(\mathit{dom\_bdg}\ f)$ denotes the (finite) set of all elements in the domain of the binding. Given an element $(s \in \mathit{dom\_bdg}\ f)$, the value of $(f \cdot s)$ is the unique element in the range of f which is bound to s. For $(s \notin \mathit{dom\_bdg}\ f)$, the value of $(f \cdot s)$ is not specified. Every binding f is characterised by its domain $(\mathit{dom\_bdg}\ f)$ and the value of $(f \cdot s)$ for $(s \in \mathit{dom\_bdg}\ f)$:

$$(f = g) \quad = \quad (\mathit{dom\_bdg}\ f = \mathit{dom\_bdg}\ g \wedge (\forall s \in \mathit{dom\_bdg}\ f.\ f \cdot s = g \cdot s))$$

Operations on records are defined by restricting the type of general operations on bindings. In Isabelle/HOL, this can be done on a purely syntactical level. Rather than introducing separate logical constants, this means that the functions on records are abbreviations which are translated implicitly to type-restricted versions of functions on bindings. For example, the function *labels* which returns the set of all labels of a record is defined by translating it to a type restricted version of function *dom_bdg*:

$$\begin{array}{lllll} \texttt{syntax} & \mathit{labels} & :: & \alpha\ \mathit{rcd} \rightarrow \mathit{string}\ \mathit{set} \\ \texttt{translations} & \text{“}\mathit{labels}\text{“} & \Rightarrow & \text{“}\mathit{dom\_bdg}\ ::\ \_\ \mathit{rcd} \rightarrow \mathit{string}\ \mathit{set}\text{“} \end{array}$$

Compared to the introduction of new constants, the use of the translation mechanism has the technical advantage that it makes it unnecessary to explicitly fold/unfold definitions. Thus, theorems derived for “general binding functions” apply directly to the functions on records.

Since only records and no general bindings occur in the main paper, the remainder of this section will concentrate on records for convenience.

Records can be constructed from the empty record (undef) by successive addition of further elements (update):

$$\begin{array}{lcl} \mathit{undef} & :: & \alpha\ \mathit{rcd} \\ \mathit{labels}\ \mathit{undef} & = & \{\,\} \\ \mathit{update} & :: & [\alpha\ \mathit{rcd}, \mathit{string}, \alpha] \;\rightarrow\; \alpha\ \mathit{rcd} \\ \mathit{labels}\,(\mathit{update}\ r\ l\ a) & = & \mathit{labels}\ r \cup \{l\} \\ (\mathit{update}\ r\ l\ a) \cdot m & = & \mathit{if}\ l = m\ \mathit{then}\ a\ \mathit{else}\ r \cdot m \end{array}$$

The fact that every record can be constructed in this way is expressed by an induction theorem:

$$\begin{array}{l} [|\ P\ \mathit{undef}; \\ \quad \forall\ r\ l\ a.\ (P\ r \wedge l \notin \mathit{labels}\ r) \implies P\ (\mathit{update}\ r\ l\ a)\ |] \implies P\ r \end{array}$$

The usual syntax for records can be imitated using Isabelle's parsing and pretty-printing tools. Here is the representatio of the record $(< x = 1, y = 2 >:: \mathbb{N}\ \mathit{rcd})$:

Isabelle/HOL Syntax	Translated to
$< \text{“}x\text{“} = 1, \text{“}y\text{“} = 2 >$	$\mathit{update}\ (\mathit{update}\ \mathit{undef}\ \text{“}x\text{“}\ 1)\ \text{“}y\text{“}\ 2)$

The definition of function *rcd_term* in Section 3 employs a function *sort_rcd* which sorts the fields of a record into a list of label/value pairs. The sorting is done with respect to the lexicographical ordering on labels. The behaviour of function *sort_rcd* can be described inductively with the help of a function *ins* which inserts an element into its right position in a sorted list.

$$\begin{array}{lcl} \mathit{sort\_rcd} & :: & \alpha\ \mathit{rcd} \to (\mathit{string} \times \alpha)\ \mathit{list} \\ \mathit{sort\_rcd}\ \mathit{undef} & = & [\,] \\ \mathit{sort\_rcd}\ (\mathit{update}\ r\ l\ a) & = & \mathit{ins}\ (\lambda\ x\ y.\ \mathit{fst}\ x \leq \mathit{fst}\ y)\ (l, a)\ (\mathit{sort\_rcd}\ r) \end{array}$$

It can be proven that the resulting list contains no duplicate labels and is sorted with respect to the lexicographical ordering on labels. Furthermore, the function *sort_rcd* is injective.

$$\begin{array}{l} \mathit{sorted}\ (\lambda\ x\ y.\ (\mathit{fst}\ x \leq \mathit{fst}\ y))\ (\mathit{sort\_rcd}\ r) \\ \mathit{nodups}\ (\mathit{map}\ \mathit{fst}\ \mathit{sort\_rcd}\ r) \\ \mathit{set}\ (\mathit{sort\_rcd}\ r)\ =\ (\lambda l.(l, r \cdot l))\ `\ (\mathit{labels}\ r) \\ \mathit{inj}\ \mathit{sort\_rcd} \end{array}$$

In addition to the function *zip_rcd* which was already described in the main text, the definition of Z_C (tables 4 and 6) uses two further functions on records:

$$\begin{array}{lcl} \mathit{map\_rcd}\ ::\ [\alpha \to \beta, \alpha\ \mathit{rcd}] & \to & \beta\ \mathit{rcd} \\ \mathit{labels}\ (\mathit{map\_rcd}\ f\ r) & = & \mathit{labels}\ r \\ l \in \mathit{dom\_bdg}\ r & \Longrightarrow & \mathit{map\_rcd}\ f\ r \cdot l\ =\ f\ (r \cdot l) \end{array}$$

$$\begin{array}{lcl} \mathit{id\_rcd}\ ::\ \mathit{string\ set} & \to & \mathit{string\ rcd} \\ \mathit{finite}\ A & \Longrightarrow & \mathit{labels}\ (\mathit{id\_rcd}\ A) = A \\ [|\ \mathit{finite}\ A;\ a \in A\ |] & \Longrightarrow & \mathit{id\_rcd}\ A \cdot a = a \end{array}$$

The result of $(\mathit{map\_rcd}\ f\ r)$ is a record with the same labels as r but where function f has been applied to every field value. For a finite set A of strings, the result of $(\mathit{id\_rcd}\ A)$ is a record where every element $(a \in A)$ is bound to itself.

3 Representation of Z_C Types

Typed logics distinguish terms according to their types. Simple type systems can be modelled directly as HOL datatypes. In the case of the logic Z_C[HR00, HR99a, HR99b] types are build from the natural number type using the type operators power set, product and record. This leads to the datatype:

$$\begin{array}{lcl} \mathit{zty} & = & \mathit{NatT} \\ & | & \mathit{SetT}\ \mathit{zty} \\ & | & \mathit{PrdT}\ \mathit{zty}\ \mathit{zty} \\ & | & \mathit{RcdT}\ (\mathit{zty}\ \mathit{rcd}) \end{array}$$

The datatype *zty* branches recursively over *rcd*. From a semantic point of view, this is unproblematic because type $(\alpha\ \mathit{rcd})$ is isomorphic to a subset of type $(\mathit{string} \to \alpha\ \mathit{option})$. Unfortunately, while branching over the latter

$$
\begin{array}{lcl}
FALSE & :: & term \\
FALSE & = & Const\ \text{“}FALSE\text{“} \\
NOT & :: & term \rightarrow term \\
NOT\ p & = & App\ \text{“}NOT\text{“}\ [p] \\
op\ \ EQ & :: & [term, term] \rightarrow term \\
t\ EQ\ u & = & App\ \text{“}EQ\text{“}\ [t, u] \\
op\ \ OR & :: & [term, term] \rightarrow term \\
p\ OR\ q & = & App\ \text{“}OR\text{“}\ [p, q] \\
EX & :: & [variable, term, term] \rightarrow term \\
EX\ v\ A\ p & = & App\ \text{“}EX\text{“}\ [A, Abs\ v\ p]
\end{array}
$$

Table 1: Representation of Z_C formulae in HOL

type is supported by the datatype package of Isabelle99-2/HOL, there is no automated support for branching over "subtypes" as yet. Hence the type *zty* was defined by an explicit axiomatisation.

4 Representation of Z_C Formulae and Terms

Both formulae and terms of Z_C will be represented as elements of type

$$term \quad = \quad (string, variable, zty)\ dbterm$$

where the type *variable* is an abbreviation for the cartesian product

$$variable \quad = \quad string \times zty$$

The membership of type *variable* in class `variant`

$$variable \quad :: \quad \texttt{variant}$$

follows from the membership of type *string* in this class, see Section 1.4. Since type *term* is simply an instantiation of datatype *dbterm*, the technical framework of Section 1 applies.

The syntactic representation of Z_C logical constants on top of type *term* is straightforward, see Table 1. In fact, there is nothing specific to Z_C about these constants - they are simply a minimal set of constants for first-order predicate logic. The representation of further, derived logical constants is entirely analogous.

The HOL representation of Z_C terms is complicated by two aspects:

1. terms can be constructed from records of other terms and
2. terms can be restricted to types

$ZERO$	$=$	$Const$ "0"
$SUC\ x$	$=$	App "Suc" $[x]$
$PAIR\ x\ y$	$=$	App "$PAIR$" $[x, y]$
$RCD\ ts$	$=$	App "RCD" $(rcd\_term\ ts)$
NAT_SET	$=$	$Const$ "NAT_SET"
$POW\_SET\ x$	$=$	App "POW_SET" $[x]$
$PRD\_SET\ x\ y$	$=$	App "$PROD\_SET$" $[x, y]$
$RCD\_SET\ ts$	$=$	App "RCD_SET" $(rcd\_term\ ts)$
$COMPREH\ v\ x\ p$	$=$	App "$COMPREH$" $[x,\ Abs\ v\ p]$
$x\ DOT\ s$	$=$	App "DOT" $[x,\ Const\ s]$
$x\ FILTER\ T$	$=$	App "$FILTER$" $[x,\ rep\_typ\ T]$
$FST\ x$	$=$	App "FST" $[x]$
$SND\ x$	$=$	App "SND" $[x]$

Table 2: Representation of Z_C terms in HOL

The first problem is solved by the introduction of an injective function *rcd_term* which represents a record of terms as a list of terms. Its definition utilises a function *sort_rcd* which transforms a record to a list of pairs sorted by their labels and a representation of pairs within type *term*.

$$\begin{array}{lcl} rcd\_term & :: & term\ rcd\ \rightarrow\ term\ list \\ rcd\_term & = & map\ (\lambda(x, y).App\ \text{"}PAIR\text{"}\ [Const\ x,\ y]) \circ sort\_rcd \end{array}$$

The representation of terms which contain references to types requires a representation of types as elements of type *term*. This is provided by the following primitive recursive function:

$$\begin{array}{lcl} rep\_typ & :: & zty\ \rightarrow\ term \\ rep\_typ\ NatT & = & Const\ \text{"}NatT\text{"} \\ rep\_typ\ (SetT\ T) & = & App\ \text{"}SetT\text{"}\ [rep\_typ\ T] \\ rep\_typ\ (PrdT\ U\ V) & = & App\ \text{"}PrdT\text{"}\ [rep\_typ\ U, rep\_typ\ V] \\ rep\_typ\ (RcdT\ S) & = & App\ \text{"}RcdT\text{"}\ (rcd\_term\ (map\_rcd\ rep\_typ\ S)) \end{array}$$

Injectivity of function *rep_typ* can be proven by structural induction on the datatype *zty*. Table 2 shows the representation of Z_C terms as elements of type *term*.

As an example, here is the translation of the term $\{x \in \mathbb{N} \mid suc\ (x) = 0\}$ to HOL:

COMPREH ("x",NatT) NAT_SET (SUC (Free("x",NatT)) EQ ZERO)

A fundamental property of the HOL representation of Z_C syntax is its faithfulness - different Z_C terms are mapped to different HOL terms. Technically this amounts to freeness properties of the HOL constants which represent the

syntax, i.e. they are injective functions and their ranges are disjoint. For example:

$$\begin{array}{rcl}((x\ EQ\ y) = (x'\ EQ\ y')) & = & ((x = x') \wedge (y = y')) \\ (x\ EQ\ y) & \neq & NOT\ p\end{array}$$

An explicit formulation of the freeness properties would lead to a number of theorems quadratic in the number of constants. Fortunately, the Isabelle/HOL simplifier can establish these properties on the fly whenever they are needed in proofs.

The free variables of a Z_C term or proposition can be calculated simply by unfolding definitions.

$$\begin{array}{lcl}frees\ FALSE & = & \{\} \\ frees\ (NOT\ t) & = & frees\ t \\ frees\ (t\ EQ\ u) & = & frees\ t\ \cup\ frees\ u \\ frees\ (t\ OR\ u) & = & frees\ t\ \cup\ frees\ u \\ frees\ (t\ IN\ u) & = & frees\ t\ \cup\ frees\ u \\ frees\ (EX\ v\ x\ p) & = & frees\ x\ \cup\ (frees\ p - \{v\})\end{array}$$

The effect of substitution on Z_C propositions other than quantifications is straightforward:

$$\begin{array}{lcl}FALSE\,[v/t] & = & FALSE \\ (NOT\ p)\,[v/t] & = & NOT\ (p\,[v/t]) \\ (r\ EQ\ s)\,[v/t] & = & r\,[v/t]\ EQ\ s\,[v/t] \\ (r\ IN\ s)\,[v/t] & = & r\,[v/t]\ IN\ s\,[v/t] \\ (r\ OR\ s)\,[v/t] & = & r\,[v/t]\ OR\ s\,[v/t]\end{array}$$

Substitution of an existentially quantified proposition requires variable renaming:

$$\begin{array}{l}[|degree\ p = 0;\ degree\ t = 0\ |] \implies \\ \quad (EX\ v\ x\ p)\,[w/t] = (let\ z = variant\ v\ (\{w\}\ \cup\ frees\ p\ \cup\ frees\ t) \\ \qquad\qquad in\ EX\ z\ (x\,[w/t])\ (p\,[v/Free\ z]\,[w/t]))\end{array}$$

Similar rules can be derived for substitution applied to wellformed Z_C terms.

The definition of derived logical connectives or term operations in Z_C is mirrored in HOL by completely analogous definitions of constants operating on type *term*. For example, the HOL representation of the Z_C subset relationship is defined as:

$$\begin{array}{lcl}op\ SUBSET & :: & [term, term] \to term \\ A\ SUBSET\ B & = & A\ IN\ POW\_SET\ B\end{array}$$

5 Representation of Typing and Wellformedness

The typed terms and wellformed propositions of Z_C are represented in HOL by two sets *tterm* and *prop*:

$$\begin{array}{lcl}tterm & :: & (term \times zty)\ set \\ prop & :: & term\ set\end{array}$$

$$
\begin{array}{lcl}
 & & FALSE \in prop \\
[|\ t : T;\ u : T\ |] & \Longrightarrow & (t\ EQ\ u) \in prop \\
[|\ t : T;\ s : SetT\ T\ |] & \Longrightarrow & (t\ IN\ s) \in prop \\
p \in prop & \Longrightarrow & NOT\ p \in prop \\
[|\ p \in prop;\ q \in prop\ |] & \Longrightarrow & (p\ OR\ q) \in prop \\
[|\ x : SetT\ T;\ p \in prop\ |] & \Longrightarrow & EX\ (s, T)\ x\ p \in prop
\end{array}
$$

Table 3: Z_C formation rules in HOL

Membership of a pair (t, T) in the set *tterm* is written as $t : T$. Typeable elements of *term* are also called *wellformed terms*. Since variables are explicitly typed, there is no need for separate typing contexts.

The sets *tterm* and *prop* are defined inductively where the introduction rules correspond directly to the Z_C typing and proposition formation rules of [HR00]. The only difficulty was the translation of the "ellipses" which occur in rules dealing with records. In order to express these succinctly, a HOL constant *zip_rcd* was introduced:

$$
\begin{array}{lcl}
zip\_rcd :: (\alpha \times \beta)\ set & \rightarrow & (\alpha\ rcd \times \beta\ rcd)\ set \\
zip\_rcd\ r & = & \{(x, y).\ labels\ x = labels\ y \\
 & & \quad \wedge\ (\forall s \in labels\ x.(x.s, y.s) : r)\}
\end{array}
$$

By definition, two records x and y are elements of the set $(zip\_rcd\ r)$ if and only if x and y have the same set of labels and if for every label, the values attached with that label in x and y are in the relationship r.

Tables 3 and 4 show the introduction rules of the sets *tterm* and *prop*. Note that these rules are mutually recursive. The wellformedness of terms containing derived Z_C constants such as *SUBSET* can be derived by unfolding the definition of such constants.

Inductive sets are defined in Isabelle/HOL to be least fixed points of a monotonic set valued function [Pau94]. This made it necessary to prove monotonicity of the function *zip_rcd* before the definition of the sets *tterm* and *prop* could be processed by the inductive set package:

$$
A \subseteq B \quad \Longrightarrow \quad zip\_rcd\ A \subseteq zip\_rcd\ B
$$

Isabelle/HOL automatically proves several theorems about inductively defined sets. Unfortunately in its raw form, the automatically generated mutual induction theorem for wellformed terms and propositions turned out to be unsuitable for the proof of some Z_C properties. The problem is caused by variable binding and will be illustrated with existential quantification. Trying to prove properties $(P\ p)$ and $(Q\ t\ T)$ for all propositions p and wellformed terms $t : T$ using the automatically generated mutual induction theorem leads (amongst others) to the following proof obligation:

$$
\begin{array}{l}
\forall\ p\ s\ x.\ [|\quad x : SetT\ T;\ p \in prop; \\
\qquad\qquad P\ p;\ Q\ x\ (SetT\ T) \qquad |] \Longrightarrow\ P\ (EX\ (s, T)\ x\ p)
\end{array}
$$

$$
\begin{array}{lcl}
 & & \mathit{Free}\ (s,T) : T \\
 & & \mathit{ZERO} : \mathit{NatT} \\
t : \mathit{NatT} & \Longrightarrow & \mathit{SUC}\ t : \mathit{NatT} \\
[|\ t : T;\ u : U\ |] & \Longrightarrow & \mathit{PAIR}\ t\ u : \mathit{PrdT}\ T\ U \\
(ts, Ts) \in \mathit{zip\_rcd}\ \mathit{tterm} & \Longrightarrow & \mathit{RCD}\ ts : \mathit{RcdT}\ Ts \\
\mathit{NAT\_SET} : \mathit{SetT}\ \mathit{NatT} & & \\
x : \mathit{SetT}\ T & \Longrightarrow & \mathit{POW\_SET}\ x : \mathit{SetT}\ (\mathit{SetT}\ T) \\
[|\ t : \mathit{SetT}\ T;\ u : \mathit{SetT}\ U\ |] & \Longrightarrow & \mathit{PRD\_SET}\ t\ u : \mathit{SetT}\ (\mathit{PrdT}\ T\ U) \\
(ts, \mathit{map\_rcd}\ \mathit{SetT}\ Ts) \in \mathit{zip\_rcd}\ \mathit{tterm} & & \\
 & \Longrightarrow & \mathit{RCD\_SET}\ ts : \mathit{SetT}\ (\mathit{RcdT}\ Ts) \\
[|\ p \in \mathit{prop};\ x : \mathit{SetT}\ T\ |] & \Longrightarrow & \mathit{COMPREH}\ (s,T)\ x\ p : \mathit{SetT}\ T \\
p : \mathit{PrdT}\ T\ U & \Longrightarrow & \mathit{FST}\ p : T \\
p : \mathit{PrdT}\ T\ U & \Longrightarrow & \mathit{SND}\ p : U \\
[|\ t : \mathit{RcdT}\ Ts;\ Us \subseteq Ts;\ U = \mathit{RcdT}\ Us\ |] & & \\
 & \Longrightarrow & (t\ \mathit{FILTER}\ U) : U \\
[|\ t : \mathit{RcdT}\ Ts;\ l \in \mathit{labels}\ Ts\ |] & \Longrightarrow & (t\ \mathit{DOT}\ l) : (Ts \cdot l)
\end{array}
$$

Table 4: Z_C typing rules in HOL

This proof obligation is problematic because due to possible variable renaming, the proposition $(EX\ (s,T)\ x\ p)$ does in general not contain p as a subterm. Hence the induction hypothesis $(P\ p)$ is often not sufficient in order to establish $(P\ (EX\ (s,T)\ x\ p))$.

There are several ways to solve this problem by strengthening the induction theorem. Our approach was based on applying wellfounded induction where the generic *size*-function on the HOL datatype *dbterm* was taken as the measure function. For the existential quantifier, this leads to the following, more amenable proof obligation:

$$
\begin{array}{lll}
\forall\ p\ s\ x. & [| & x : \mathit{SetT}\ T;\ p \in \mathit{prop}; \\
 & & \forall p' \in \mathit{prop}.\ \mathit{size}\ p' \leq \mathit{size}\ p \Longrightarrow P\ p'; \\
 & & \forall x'\ T'.\ x' : \mathit{SetT}\ T' \wedge \mathit{size}\ x' \leq \mathit{size}\ x \Longrightarrow Q\ x'\ (\mathit{SetT}\ T') \\
 & |] & \Longrightarrow P\ (EX\ (s,T)\ x\ p)
\end{array}
$$

Using the improved induction theorem, several basic properties of the HOL representation of typed terms and propositions were established. In particular, typed terms and wellformed propositions are disjoint and consist of proper terms only:

$$
\begin{array}{lcl}
\{t.\ \exists\ T.t : T\}\ \cap\ \mathit{prop} & = & \{\,\} \\
p : \mathit{prop} & \Longrightarrow & \mathit{degree}\ p = 0 \\
t : T & \Longrightarrow & \mathit{degree}\ t = 0
\end{array}
$$

A fundamental result about Z_C is that typing is unique and that all types are

inhabited by a closed, wellformed term:

$$[|\ t : T;\ t : U\ |] \quad \Longrightarrow \quad T = U$$
$$\forall T.\ \exists t.\ \mathit{degree}\ t = 0\ \wedge\ \mathit{frees}\ t = \{\}\ \wedge\ t : T$$

The Z_C formation and typing rules in tables 3 and 4 are in form of implications. Because of freeness properties of the formula and term representation, it is possible to extend most of these implications to equivalences. This result is sometimes called the "generation lemma". For example:

$$\begin{array}{lcl} (\mathit{NOT}\ p \in \mathit{prop}) & = & (p \in \mathit{prop}) \\ (\mathit{POW\_SET}\ C : T) & = & (\exists\ U.\ T = \mathit{SetT}\ (\mathit{SetT}\ U) \wedge C : \mathit{SetT}\ U) \end{array}$$

6 Representation of Judgements

The formulation of Z_C in [HR00] employs judgements which relate a set of propositions (the assumptions) with another proposition (the conclusion). Because propositions are modelled as elements of type *term*, the set *zc* which represents all valid Z_C judgements in HOL is of type:

$$\mathit{zc} \quad :: \quad (\mathit{term}\ \mathit{set} \times \mathit{term})\ \mathit{set}$$

Membership of a pair (ps, q) in *zc* is written as $ps \vdash q$:

$$\begin{array}{lcl} \mathit{op}\ \vdash & :: & [\mathit{term}\ \mathit{set}, \mathit{term}] \rightarrow \mathit{bool} \\ ps \vdash q & = & (ps, q) \in \mathit{zc} \end{array}$$

In analogy to the definition of the sets *prop* and *tterm*, the definition of *zc* takes the form of an inductive set definition where the introduction rules are HOL translations of Z_C inference rules. Table 5 shows rules dealing with the logical quantifiers and equality. Table 6 contains inference rules dealing with other Z_C constants.

Several Z_C inference rules contain assumptions about typing resp. membership in *prop*. These assumptions were chosen carefully in order to guarantee that all elements of a judgement are wellformed propositions:

$$ps\ \vdash q \quad \Longrightarrow \quad (ps \cup \{q\})\ \subseteq\ \mathit{prop}$$

Of course, from the basic Z_C inference rules, further Z_C judgements can be derived in Isabelle/HOL. As an example, we proved some basic monotonicity theorems:

$$\begin{array}{l} x \vdash A\ \mathit{SUBSET}\ B \quad \Longrightarrow \quad x \vdash \mathit{POW\_SET}\ A\ \mathit{SUBSET}\ \mathit{POW\_SET}\ B \\ [|\ x \vdash A\ \mathit{SUBSET}\ B; x \vdash C\ \mathit{SUBSET}\ D\ |] \\ \qquad\qquad \Longrightarrow \quad x \vdash \mathit{PRD\_SET}A\ C\ \mathit{SUBSET}\ \mathit{PRD\_SET}B\ D \\ [|\ x \subseteq\ \mathit{prop};\ \ (\mathit{xs}, \mathit{ys}) : \mathit{zip\_rcd}\ \{(a, b).\ x \vdash a\ \mathit{SUBSET}\ b\}\ |] \\ \qquad\qquad \Longrightarrow \quad x \vdash \mathit{RCD\_SET}\ \mathit{xs}\ \mathit{SUBSET}\ \mathit{RCD\_SET}\ \mathit{ys} \end{array}$$

The derivation of these Z_C theorems on top of HOL involved explicit proofs of syntactic properties such as freeness of variables, typing of terms and calculating the result of substitutions. While the proofs in themselves were not that hard, it has to be said that the mixture of "logical" proof goals" with "syntactic side condition" proof goals led to clutter in some proof states.

$$
\begin{array}{rcl}
P \in \mathit{prop} & \Longrightarrow & \{P\} \vdash P \\
[\!|\ x \vdash P;\ x \subseteq y;\ y \subseteq \mathit{prop}\ |\!] & \Longrightarrow & y \vdash P \\
[\!|\ Q \in \mathit{prop};\ x \vdash P\ |\!] & \Longrightarrow & x \vdash P\ \mathit{OR}\ Q \\
[\!|\ P \in \mathit{prop};\ x \vdash Q\ |\!] & \Longrightarrow & x \vdash P\ \mathit{OR}\ Q \\
[\!|\ x \vdash P\ \mathit{OR}\ Q;\ x \cup \{P\} \vdash R;\ x \cup \{Q\} \vdash R\ |\!] & \Longrightarrow & x \vdash R \\
\{P\} \cup x \vdash \mathit{FALSE} & \Longrightarrow & x \vdash \mathit{NOT}\ P \\
[\!|\ x \vdash P;\ x \vdash \mathit{NOT}\ P\ |\!] & \Longrightarrow & x \vdash \mathit{FALSE} \\
\vdash \mathit{NOT}\,(\mathit{NOT}\ P) & \Longrightarrow & x \vdash P \\
P \in \mathit{prop} & \Longrightarrow & \{\mathit{FALSE}\} \vdash P \\
[\!|\ x \vdash P\,[(s,T)/t];\ x \vdash t\ \mathit{IN}\ C;\ t : T;\ P \in \mathit{prop}\ |\!] & \Longrightarrow & x \vdash \mathit{EX}\ (s,T)\ C\ P \\
[\!|\ x \vdash \mathit{EX}\ z\ C\ P;\ y \notin (\mathit{frees}\ Q \cup (\bigcup a \in x.\, \mathit{frees}\ a)); & & \\
\quad x \cup \{\mathit{Free}\ y\ \mathit{IN}\ C, P\,[z/\mathit{Free}\ y]\} \vdash Q\ |\!] & \Longrightarrow & x \vdash Q \\
t : T & \Longrightarrow & \{\,\} \vdash t\ \mathit{EQ}\ t \\
x \vdash t\ \mathit{EQ}\ u & \Longrightarrow & x \vdash u\ \mathit{EQ}\ t \\
[\!|\ x \vdash t\ \mathit{EQ}\ u;\ x \vdash P[(s,T)/t];\ P \in \mathit{prop};\ t : T\ |\!] & \Longrightarrow & x \vdash P[(s,T)/u]
\end{array}
$$

Table 5: Z_C inference rules in HOL (I)

7 Carrier Sets

In Z_C, the set of elements of a type plays an important role. This concept can be formalised in HOL by the introduction of a function *carrier* which takes types to sets. This function is defined by primitive recursion over the datatype *zty* of Z_C types:

$$
\begin{array}{lcl}
\mathit{carrier} & :: & \mathit{zty} \rightarrow \mathit{term} \\
\mathit{carrier}\ \mathit{NatT} & = & \mathit{NAT\_SET} \\
\mathit{carrier}\ (\mathit{SetT}\ T) & = & \mathit{POW\_SET}\ (\mathit{carrier}\ T) \\
\mathit{carrier}\ (\mathit{PrdT}\ U\ V) & = & \mathit{PRD\_SET}\ (\mathit{carrier}\ U)\ (\mathit{carrier}\ V) \\
\mathit{carrier}\ (\mathit{RcdT}\ \mathit{Ts}) & = & \mathit{RCD\_SET}\ (\mathit{map\_rcd}\ \mathit{carrier}\ \mathit{Ts})
\end{array}
$$

By induction over *zty*, one can prove easily two basic properties of (*carrier* T):

$$
\begin{array}{l}
\mathit{carrier}\ T : \mathit{SetT}\ T \\
\mathit{frees}\ (\mathit{carrier}\ T) = \{\,\}
\end{array}
$$

Finally, it can be proven that a closed term of type T is a member of the carrier set of T. Notice that this membership is a judgement of Z_C:

$$
[\!|\ t : T;\ \mathit{frees}\ t = \{\,\};\ x \subseteq \mathit{prop}\ |\!] \Longrightarrow x \vdash (t\ \ \mathit{IN}\ \mathit{carrier}\ T)
$$

The proof was carried out by induction over the term t.

$$[|\ RCD\ ts : RcdT\ Ts;\ s \in labels\ ts\ |] \implies \{\} \vdash (RCD\ ts\ DOT\ s)\ EQ\ (ts\ \cdot s)$$

$$t : RcdT\ Ts \implies \{\} \vdash t\ EQ\ RCD\,(map\_rcd\,(\lambda s.t\ DOT\ s)\,(id\_rcd\,(labels\ Ts)))$$

$$[|\ t : T;\ u : U\ |] \implies \{\} \vdash FST\ (PAIR\ t\ u)\ EQ\ t$$

$$[|\ t : T;\ u : U\ |] \implies \{\} \vdash SND\ (PAIR\ t\ u)\ EQ\ u$$

$$PAIR\ t : PrdT\ U\ V \implies \{\} \vdash PAIR\ (FST\ t)\ (SND\ t)\ EQ\ t$$

$$[|\ x \vdash P[(s,T)/t];\ x \vdash t\ IN\ C;\ P \in prop;\ t : T\ |] \implies x \vdash t\ IN\ COMPREH\ (s,T)\ C\ P$$

$$x \vdash t\ IN\ COMPREH\ z\ C\ P \implies x \vdash t\ IN\ C$$

$$[|\ x \vdash t\ IN\ COMPREH\ (s,T)\ C\ P;\ P \in prop;\ t : T\ |] \implies x \vdash P[(s,T)/t]$$

$$\{\} \vdash ZERO\ IN\ NAT\_SET$$

$$x \vdash n\ IN\ NAT\_SET \implies x \vdash SUC\ n\ IN\ NAT\_SET$$

$$n : NatT \implies \{\} \vdash NOT\ (ZERO\ EQ\ SUC\ n)$$

$$x \vdash SUC\ n\ EQ\ SUC\ m \implies x \vdash n\ EQ\ m$$

$$[|\ x \vdash P[z/ZERO];\ z = (s, NatT)\ x \cup \{P\} \vdash P[z/SUC\ (Free\ z)]\ |] \implies x \vdash P$$

$$[|\ x \vdash t\ IN\ T;\ x \vdash u\ IN\ U\ |] \implies x \vdash PAIR\ t\ u\ IN\ PRD\_SET\ T\ U$$

$$x \vdash a\ IN\ PRD\_SET\ T\ U \implies x \vdash FST\ a\ IN\ T$$

$$x \vdash a\ IN\ PRD\_SET\ T\ U \implies x \vdash SND\ a\ IN\ U$$

$$[|\ (\{Free\ z\ IN\ C\} \cup x) \vdash Free\ z\ IN\ D;\ z \notin frees\ C \cup frees\ D|] \implies x \vdash C\ IN\ POW\_SET\ D$$

$$[|\ x \vdash C\ IN\ POW\_SET\ D;\ x \vdash t\ IN\ C\ |] \implies x \vdash t\ IN\ D$$

$$[|\ (ts, cs) \in zip\_rcd\{(a,b).\, x \vdash a\ IN\ b\};\ x \subseteq prop|] \implies x \vdash RCD\ ts\ IN\ RCD\_SET\ cs$$

$$[|\ x \vdash t\ IN\ RCD\_SET\ cs;\ s \in labels\ cs\ |] \implies x \vdash (t\ DOT\ s)\ IN\ (cs\ \cdot\ s)$$

$$[|\ x \vdash A\ SUBSET\ B;\ x \vdash B\ SUBSET\ A\ |] \implies x \vdash A\ EQ\ B$$

$$[|\ t : RcdT\ Ts;\ Us \leq Ts;\ s \in labels\ Us\ |] \implies \{\} \vdash ((t\ FILTER\ RcdT\ Us)\ DOT\ s)\ EQ\ (t\ DOT\ s)$$

Table 6: Z_C inference rules in HOL (II)

8 Conclusions

This paper presents a deep embedding of the logic Z_C in Isabelle/HOL. The embedding is based on a general HOL theory of de Bruijn terms. This theory is highly reusable and could provide a useful foundation for other logic embeddings. Another characteristic feature of our approach is the use of inductive set definitions for wellformed terms, propositions and judgements. Main results are mechanical proofs of the uniqueness of types, type inhabitation and that elements of judgements are wellformed propositions. Furthermore, a carrier set function is defined and the membership of closed typed terms in the corresponding carrier set is established.

A strong point of the deep embedding is that it allows explicit reasoning in Isabelle about concepts such as substitution, freeness of variables or typing. Unfortunately, this also is one of the weaknesses of the deep embedding - because such syntactic matters are represented explicitly, they cause proof obligations about syntactic side conditions when performing Z_C reasoning on top of HOL. Even with Isabelle/HOL's powerful proof tactics, this tends to introduce a lot of clutter during Z_C derivations. This suggests that for the purpose of actually performing Z_C reasoning on top of HOL, it is preferable to use more semantic embeddings. In contrast to the deep embedding, the similarity of Z_C and HOL should allow a considerable reuse of HOL theories in such semantic embeddings.

References

[Gor94] A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 414–427. Springer-Verlag, 1994.

[HR99a] Henson and Reeves. Revising Z: Part i - logic and semantics. *Formal Apects of Computing*, 11:359–380, 1999.

[HR99b] Henson and Reeves. Revising Z: Part ii - logical development. *Formal Apects of Computing*, 11:381–401, 1999.

[HR00] Henson and Reeves. Investigating Z. *JLC: Journal of Logic and Computation*, 10:43–73, 2000.

[Owe95] Chris Owens. Coding binding and substitution explicitly in isabelle. In L.C. Paulson, editor, *Proceedings of the First Isabelle Users Workshop*, Technical Report 379, pages 36–52. Computer Laboratory, University of Cambridge, September 1995.

[Pau94] L.C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction — CADE-12*, LNAI 814, pages 148–161. Springer-Verlag, 1994.

[Von95] J. Von Wright. Representing higher-order logic proofs in HOL. *The Computer Journal*, 38(2):171–179, 1995.