Inference Rules for ZK

Sentot Kromodimoeljo

June 2021

0.1 Introduction

0.1.1 ZK versus EVES

EVES[3] is a program verification system whose development started around 1983 at IP Sharp Associates, then Reuters and later at ORA Canada, on behalf of the Government of Canada. Although a technical success, its use was limited primarily to the developers and a few defense-related research organizations. Perhaps better known, the Z/EVES system[5] uses EVES as its inference engine and is a popular tool for the **Z** notation[6].

A novel feature of EVES is its ability to generate checkable proof logs: logical reasoning performed by complex decision procedures and heuristics is translated into a sequence of simple inference steps. A proof checker much simpler than EVES can be programmed to check that a proof indeed follows from a sequence of inference steps. At the time the proof logging mechanism was developed, however, the focus was on being able to generate sequences of simple inferences, without too much concern on what should be the set of inference rules. As a result, the number of inference rules shot up to 139, an unacceptably large number even though each inference rule is simple and appears to be self-evident.

The author had been able to identify a much smaller number of inference rules much later. However, because of licensing concerns, the author could not simply modify EVES to use the reduced set of inference rules. Thus the idea for the development of ZK. By eliminating licensing issues and focusing on the strength of the theorem prover, the hope is ZK could be more widely used.

ZK implements a non-executable subset of Verdi[2], which is the notation for EVES. No executable construct of Verdi is supported: the focus is on theorem proving since hardly anyone was using the executable features of Verdi. Characters, strings, enumerated types, record types and array types are all gone, with the remaining built-in types being `(BOOL)` and `(INT)`. Function groups are also gone along with executable functions and procedures. The resulting language handled by ZK becomes considerably simpler. Although having no effect on the inference rules, the required infrastructure becomes simpler, especially with respect to proof obligation generation.

In addition to doing away with executable features of EVES (which includes a Verdi interpreter), ZK uses a much smaller number of inference rules: 16. The number of inference rules could have been made even smaller, however there would have been trade-offs involved. In the end, the author decided to use the 16 inference rules presented in this report.

The implementation of ZK uses ANSI Common Lisp[1] as a basis. As with EVES, a lexicon is defined which is strictly adhered by the code, except for calls to open source libraries. The Common Lisp pretty printer is used for pretty-printing and open source libraries are used for file compression/decompression. Enhancements to the theorem prover include improved heuristics for dealing with recursive functions.

0.1.2 Proof Logs

A proof log is a sequence of proof log entries. Each entry has an inference name, specifying the inference rule to use, and an index into a subformula of the current formula, on which the inference rule is to apply. An entry may have additional parameters.

An example of a proof log entry is

```
(IF-TRUE (2 3))
```

which, when applied to the formula

```
(ALL (X) (IF (A X) (B X) (IF (TRUE) (C X) (D X))))
```

produces the formula

```
(ALL (X) (IF (A X) (B X) (C X))).
```

The entry specifies that the `IF-TRUE` inference rule is to be used and the index `(2 3)` specifies that the inference step is to be applied to the 3rd of the 2nd subexpression (origin 0 for each index component, although 0 is never used since it always points to the function or operator position).

Extra parameters may be needed to use an inference rule in reverse. For example, to get back to the original formula, we apply the proof log entry

```
(IF-TRUE (2 3) (D X) T).
```

The `T` simply says that the inference rule is to be applied in reverse. Had we instead applied the entry

```
(IF-TRUE (2 3) (E X) T)
```

we would have ended up with the formula

```
(ALL (X) (IF (A X) (B X) (IF (TRUE) (C X) (E X))))
```

which, by the way, is also equivalent to the original formula.

A proof log is a sequence of inference steps that transforms a proposition. For a proof of a proposition, the sequence tranforms the proposition to `(TRUE)`. Typically it uses existing axioms (theorems) so proof checking isn't simply tautology checking. In addition, when "diving" into a subexpression, propositions determined by `IF` tests along the path to the subexpression may be assumed. Thus there is a notion of context, determined globally by earlier axioms and locally by the `IF` structure of the formula being transformed.

Although originally designed to enable the checking of proofs, proof logs can also be used to animate proofs. In ZK this is supported by the proof browsing mechanism. A proof animator can in principle be developed without too much complication, either integrated with ZK or as a separate tool.

0.2 Framework

0.2.1 The Library Facility

ZK supports theory development *in the large* by providing a library facility. A ZK session can be saved as a `SPEC`, `MODEL` or `FREEZE` library unit using the `MAKE` command. Any session can be saved as a `FREEZE` unit as work in progress, and the work can be continued in a subsequent session using the `EDIT` command.

`SPEC` library units are units that can be *loaded* using the `LOAD` command and are typically specific theory extensions (without proofs). A session with

FUNCTION-STUB declarations can be saved as a SPEC unit. However, there are consistency requirements between a SPEC unit and the corresponding MODEL unit, which include the requirement that each FUNCTION-STUB declaration in the SPEC unit must have a corresponding FUNCTION or ZF-FUNCTION declaration of the same arity in the MODEL unit.

MODEL library units are intended to provide set-theoretic models for their corresponding SPEC units. All proof obligations (to be described in the next subsection) must be discharged before a session can be saved as a MODEL unit. A session with FUNCTION-STUB declarations cannot be saved into a MODEL unit. In addition, if a corresponding SPEC unit already exists and the consistency check fails, the session also cannot be saved as a MODEL unit. Last but not least, a MODEL unit cannot load a (SPEC) unit that causes logical circularity. All of these restrictions are enforced by ZK's library mechanism.

0.2.2 Proof Obligations

In ZK, the user develops a theory *in the small* by introducing declarations and discharging the proof obligations associated with the declarations. Proof obligations need to be discharged to ensure that theory extensions resulting from declarations are *conservative extensions*. The absence of executable features of Verdi greatly simplifies proof obligation generation in ZK.

For the various axiom types of declarations (AXIOM, RULE, FRULE or GRULE), the proof obligation for a declaration is the *body* of the declaration implicitly universally quantified over all variables that occur free in the *body*.

A LOAD command, although not a declaration, extends the current theory. It does not have a proof obligation as all proofs for the loaded unit would be provided by the corresponding MODEL unit. To ensure that a session is a conservative theory extension, in addition to having all proof obligations in the session discharged, all loaded units (including transitive loads) must have corresponding MODEL units.

A FUNCTION-STUB declaration has a proof obligation of (FALSE), meaning it can't be proved directly. As explained in the previous subsection, a session with FUNCTION-STUB declarations cannot be saved as a MODEL library unit. However, it can be saved as a SPEC unit in which case the corresponding MODEL unit ensures that the theory extension is a conservative extension by having an *implementaion* for the declared function.

For the MAP and SELECT variants of ZF-FUNCTION declarations, the proof obligation is (TRUE). For the THAT variant of ZF-FUNCTION declarations, i.e., the body has the form (THAT x P)[1], the proof obligation is

$$\texttt{(SOME (}x'\texttt{) (ALL (}x\texttt{) (= }P\texttt{ (= }x\ x'\texttt{))))},$$

where x' does not occur free in P, and the proof obligation is implicitly universally quantified over the variables that occur free in P other than x. This essentially says there is a unique x satisfying P.

Proof obligation generation for FUNCTION declarations is the most elaborate. Potentially, a FUNCTION declaration defines a recursive function and in ZK, we need to prove that a function application does not recur indefinitely, i.e., it

[1]Italicized symbols are placeholders.

eventually terminates, guaranteeing that the function definition is a conservative theory extension. The proof obligation for a `FUNCTION` declaration with recursive calls is based on the *measure* expression and the *body* of the declaration. Informally, the measure expression specifies a measure function, and the proof obligation states that at each recursive call in the function *body*, the measure function applied to the arguments of the recursive call is smaller according to the well-founded relation `M<` than the application of the measure function to the formal parameters of the `FUNCTION` declaration. More precisely, to generate the proof obligation for a declaration for a recursive function *f* with parameters $(x_1\ \ldots\ x_i)$ and measure function *m*, the following recursive function `po` is applied to the *body* of the function declaration:

`po`(e) = `(TRUE)` for e a variable, an integer literal, `(TRUE)` or `(FALSE)`,
`po((ALL(`x`)`P`))` = `(ALL(`x`) po(`P`))`,
`po((ALL(`$x\ y\ \ldots$`)`P`))` = `(ALL(`x`) po((ALL(`$y\ \ldots$`)`P`)))`,
`po((SOME(`x`)`P`))` = `(ALL(`x`) po(`P`))`,
`po((SOME(`$x\ y\ \ldots$`)`P`))` = `(ALL(`x`) po((SOME(`$y\ \ldots$`)`P`)))`,
`po((IF` *test left right*`))` = `(AND po(`*test*`) (IF` *test* `po(`*left*`) po(`*right*`)))`,
`po((`$f\ e_1 \ldots e_i$`))` = `(AND po(`e_1`)...po(`e_i`)(M<` $m(e_1,\ldots,e_i)$ $m(x_1,\ldots,x_i)$`)))`,
`po((`$g\ e_1 \ldots e_j$`))` = `(AND po(`e_1`)...po(`e_j`)(TRUE))`,

where g is a function other than *f*. Note that $m(x_1,\ldots,x_i)$ is the same as the measure expression for the declaration. For a non-recursive `FUNCTION` declaration, the same function is used to generate the proof obligation which then simplifies to `(TRUE)`.

Proof obligations are implicitly universally quantified over its free variables. However, for the purpose of proof logging and checking, the quantification is made explicit. For example, if a proof obligation P has v_1 and v_2 as its free variables, and the first free occurrence of v_1 in P is before the first free occurrence of v_2, then the explicitly quantified proof obligation is

`(ALL (`v_1`) (ALL (`v_2`)` P`))`.

Proof obligations generated would be available to a proof checker along with the declarations. A proof checker can either take a generated proof obligation as correct, or it can check to ensure that it is correct. Note that a proof obligation generated may be simplified before being made available for discharge, in which case the inferences made in the simplification are recorded in a proof log.

0.2.3 Theory Context

For many declarations, an axiom (theorem) is associated with the declaration. For example, a declaration of any of the axiom type (`AXIOM`, `RULE`, `FRULE` or `GRULE`) has its body available as an axiom. A `LOAD` command also may have axioms associated with it.

An axiom associated with a declaration or a `LOAD` command can be used in the proof of later declarations and is referred by its name. The details of how an axiom is used in a proof will be covered in the explanation for the `USE-AXIOM` inference rule in a later section.

Conceptually, the axioms that are accessible for use in a proof provide a context for the proof. We will call it the *theory context*. An axiom is accessible if the declaration or `LOAD` command with which it is associated is accessible. The declarations and `LOAD` commands that are accessible for the discharge of a proof obligation are exactly those prior to the declaration whose proof obligation is being discharged, and include declarations in the *initial theory* of ZK. For a proof of an anonymous proposition, all declarations are accessible. Note that in ZK, a proof obligation need not be discharged before another declaration is entered. The order in which proof obligations are discharged need not be the same as the order in which the declarations that generate them are entered.

As with proof obligations, implicit universal quantification of axiom bodies over their free variables are made explicit for proof logging and checking. For example, the declaration

```
(RULE FOO (X) (= (* X 0) 0))
```

will have

```
(ALL (X) (= (* X 0) 0))
```

available as an axiom named `FOO` for proof logging and proof checking purposes.

A declaration may also generate other axioms that are named by appending a suffix to the name being declared:

- An axiom for the raw proof obligation (suffix: "`.RAWPO`"). The justification for this axiom relies on correctness of proof obligation generation. A formal document on proof obligation generation may be available in the future. A proof checker may want to check that the raw proof obligation is correctly generated.

- An axiom for the proof obligation presented to the user (suffix" "`.PO`"). A proof log for converting `RAWPO` to `PO` is made available for a proof checker.

- An axiom that *defines* the theory extension introduced by the declaration (suffix: "`.DEFINITION`"). The justification for this axiom relies on correctness of the definition as described in the formal desrciption of Verdi[4]. A document on the formal description of the subset of Verdi recognized by ZK may be available in the future. A proof checker may want to check that the definition is correctly produced.

- An axiom that is a *rephrasing* of the `DEFINITION` axiom in terms of more primitive symbols (suffix: "`.INTERNAL`"). A proof log for converting `DEFINITION` to `INTERNAL` is made available for a proof checker.

- For a `FUNCTION` declaration, when ZK can determine that an application of the function always produces a `(BOOL)` value or always an `(INT)` value, an axiom is generated to capture this fact as a theorem (suffix: "`.TYPE-OF-AXIOM`"). A proof log for the proof of the theorem is made available for a proof checker.

A `LOAD` command may generate axioms, representing theory extensions introduced by the loading of library units. No proof logs are attached to these axioms.

0.2.4 Formula Context

In the introductory section on proof logs, it was explained that an inference step specified by a proof log entry applies to a subexpression of the current formula. A proof log entry index of `()` means the inference step is to apply to the entire formula. When an inference step is to apply to a proper subexpression within the formula, additional propositions may be available similar to the way axioms are available for use. The details of how these propositions are used will be covered in the explanation for the `LOOK-UP` inference rule.

The additional propositions that are available for use in an inference applied to a subexpression depend on the *if* structure of the formula. As we dive into the subexpression on which an inference is to apply, propositions may be added or removed. We call the set of propositions available for use the *formula context*. Whereas the *theory context* is global in the sense that it remains constant throughout all entries in a proof log, the *formula context* is local in the sense that it is dynamic and may change with each entry in a proof log. In the rest of this subsection, *context* will mean *formula context*.

Starting with an empty *context* when referring to the entire formula, there are 4 rules regarding *context* that apply as we dive into a subexpression:

1. As we dive into `LEFT` of `(IF TEST LEFT RIGHT)`, the propositions `TEST` and `(= TEST (TRUE))` are added to *context*.
2. As we dive into `RIGHT` of `(IF TEST LEFT RIGHT)`, the propositions `(IF TEST (FALSE) (TRUE))` and `(= (IF TEST (FALSE) (TRUE)) (TRUE))` are added to *context*.
3. As we dive into `EXPR` of `(ALL (vars) EXPR)` or `(SOME (vars) EXPR)`, all propositions with free occurrences of any of the variables in `vars` are removed from *context*. Here `vars` is a sequence of one or more variables (separated by spaces).
4. Diving into any other immediate subexpression does not change *context*.

0.2.5 Selected Subexpression

Each proof log entry operates on a subexpression, or more accurately, a triple

$$(\textit{theory context}, \textit{formula context}, \textit{subexpression}).$$

The overall result of applying a proof log entry is the subexpression at the specified position is transformed using the inference rule, with everything else in the formula remaining the same.

0.3 The Inference Rules

There are 16 inference rules used by ZK. All inferences are reversible with `RENAME-UNIVERSAL` and `FLIP-UNIVERSAL` being their own inverses. In addition, the effect of a `LOOK-UP` inference can be reversed by performing a `LOOK-UP` of the original subexpression.

For the rest of this section, it is assumed that *index* correctly selects the subexpression to be transformed. Each of the inference rules will be described along with the corresponding syntax for proof log entries.

0.3.1 IF-TRUE

`IF-TRUE`:

$$(\texttt{IF (TRUE)}\ \textit{left right}) \Leftrightarrow \textit{left}$$

where *left* and *right* are placeholders. The $\Leftrightarrow$ indicates that the transformation can go in either direction.
`(IF-TRUE` *index*`)`: `(IF (TRUE)` *left right*`)` $\Rightarrow$ *left*.
`(IF-TRUE` *index right* `T)`: *left* $\Rightarrow$ `(IF (TRUE)` *left right*`)`.

0.3.2 IF-FALSE

`IF-FALSE`:

$$(\texttt{IF (FALSE)}\ \textit{left right}) \Leftrightarrow \textit{right.}$$

`(IF-FALSE` *index*`)`: `(IF (FALSE)` *left right*`)` $\Rightarrow$ *right*.
`(IF-FALSE` *index left* `T)`: *right* $\Rightarrow$ `(IF (FALSE)` *left right*`)`.

0.3.3 IF-EQUAL

`IF-EQUAL`:

$$(\texttt{IF}\ \textit{test expr expr}) \Leftrightarrow \textit{expr.}$$

`(IF-EQUAL` *index*`)`: `(IF` *test expr expr*`)` $\Rightarrow$ *expr*.
`(IF-EQUAL` *index test* `T)`: *expr* $\Rightarrow$ `(IF` *test expr expr*`)`.

0.3.4 IF-TEST

`IF-TEST`:

$$(\texttt{IF}\ \textit{test left right}) \Leftrightarrow (\texttt{IF}\ (=\ \textit{test}\ \texttt{(TRUE)})\ \textit{left right}).$$

`(IF-TEST` *index*`)`: `(IF` *test left right*`)` $\Rightarrow$ `(IF` $(=$ *test* `(TRUE)`$)$ *left right*`)`.
`(IF-TEST` *index* `T)`: `(IF` $(=$ *test* `(TRUE)`$)$ *left right*`)` $\Rightarrow$ `(IF` *test left right*`)`.

0.3.5 IS-BOOLEAN

`IS-BOOLEAN`:

$$\textit{expr} \Leftrightarrow (\texttt{IF}\ \textit{expr}\ \texttt{(TRUE) (FALSE)})$$

with the condition that *expr* is known to be in `(BOOL)`. An expression is known to be in `(BOOL)` if it is an application of one of the following functions and operators:
`TRUE, FALSE, AND, OR, IMPLIES, NOT, =, >=, <=, >, <, M<, IN, SUBSET, ALL, SOME`,
or it is of the form `(IF` *test left right*`)` with *left* and *right* both known to be in `(BOOL)`.
`(IS-BOOLEAN` *index*`)`: *expr* $\Rightarrow$ `(IF` *expr* `(TRUE) (FALSE))`.
`(IS-BOOLEAN` *index* `T)`: `(IF` *expr* `(TRUE) (FALSE))` $\Rightarrow$ *expr*.

0.3.6 LOOK-UP

`LOOK-UP`:

$$\textit{expr} \Leftrightarrow \textit{result}$$

with the condition that `(=` *expr result*`)` or `(=` *result expr*`)` appears in *formula context*.
`(LOOK-UP` *index result*`)`: *expr* $\Rightarrow$ *result*.

An inverse operation is not needed since `LOOK-UP` is perfectly symmetrical:
`(LOOK-UP` *index expr*`)`: *result* $\Rightarrow$ *expr*
given the same *formula context*.

0.3.7 CASE-ANALYSIS

`CASE-ANALYSIS`

$$(f\,(\texttt{IF}\ \textit{test}\ l_1\ r_1)\ (\texttt{IF}\ \textit{test}\ l_2\ r_2)\ \ldots) \Leftrightarrow (\texttt{IF}\ \textit{test}\ (f\,l_1\ l_2\ \ldots)\ (f\,r_1\ r_2\ \ldots))$$

where *f* is any function of arity ≥ 1.
`(CASE-ANALYSIS` *index* `0)`:
$(f\,(\texttt{IF}\ \textit{test}\ l_1\ r_1)\ (\texttt{IF}\ \textit{test}\ l_2\ r_2)\ \ldots) \Rightarrow (\texttt{IF}\ \textit{test}\ (f\,l_1\ l_2\ \ldots)\ (f\,r_1\ r_2\ \ldots))$.
`(CASE-ANALYSIS` *index* `0 T)`:
$(\texttt{IF}\ \textit{test}\ (f\,l_1\ l_2\ \ldots)\ (f\,r_1\ r_2\ \ldots)) \Rightarrow (f\,(\texttt{IF}\ \textit{test}\ l_1\ r_1)\ (\texttt{IF}\ \textit{test}\ l_2\ r_2)\ \ldots)$.
The 0 is needed because the proof logging mechanism provides a shortcut:

$$(f\,l_1\ \ldots\ (\texttt{IF}\ \textit{test}\ l_i\ r_i)\ l_{i+1}\ \ldots) \Leftrightarrow$$
$$(\texttt{IF}\ \textit{test}\ (f\,l_1\ \ldots\ l_i\ l_{i+1}\ \ldots)\ (f\,l_1\ \ldots\ r_i\ l_{i+1}\ \ldots))$$

for $1 \leq i \leq$ arity of *f*, which combines the inference rules `CASE-ANALYSIS` and `IF-EQUAL`. The proof log entry for the shortcut uses the same format as that for the unadulterated `CASE-ANALYSIS` inference rule (with $1 \leq i \leq$ arity of *f*):
`(CASE-ANALYSIS` *index i*`)` and
`(CASE-ANALYSIS` *index i* `T)`.
The shortcut is used extensively by ZK, especially for `IF` normalization and unnormalization where *f* is `IF` and $i = 1$.

0.3.8 ALL-CASE-ANALYSIS

`ALL-CASE-ANALYSIS`

$$(\texttt{ALL}\ (\textit{vars})\ (\texttt{IF}\ P\ Q\ (\texttt{FALSE}))) \Leftrightarrow (\texttt{IF}\ (\texttt{ALL}\ (\textit{vars})\ P)\ (\texttt{ALL}\ (\textit{vars})\ Q)\ (\texttt{FALSE}))$$

where *vars* is a sequence of one or more variables (e.g., `X Y`).
`(ALL-CASE-ANALYSIS` *index*`)`:
$(\texttt{ALL}\ (\textit{vars})\ (\texttt{IF}\ P\ Q\ (\texttt{FALSE}))) \Rightarrow (\texttt{IF}\ (\texttt{ALL}\ (\textit{vars})\ P)\ (\texttt{ALL}\ (\textit{vars})\ Q)\ (\texttt{FALSE}))$
`(ALL-CASE-ANALYSIS` *index* `T)`:
$(\texttt{IF}\ (\texttt{ALL}\ (\textit{vars})\ P)\ (\texttt{ALL}\ (\textit{vars})\ Q)\ (\texttt{FALSE})) \Rightarrow (\texttt{ALL}\ (\textit{vars})\ (\texttt{IF}\ P\ Q\ (\texttt{FALSE})))$.

0.3.9 USE-AXIOM

`USE-AXIOM`

$$\texttt{(TRUE)} \Leftrightarrow \textit{axiom}$$

where *axiom* is in *theory context.* `USE-AXIOM` is to *theory context* what `LOOK-UP` is to *formula context.*
`(USE-AXIOM` *index axiom-name*`)`: `(TRUE)` $\Rightarrow$ *axiom.*
`(USE-AXIOM` *index axiom-name* `T)`: *axiom* $\Rightarrow$ `(TRUE)`.
Note that the *axiom* is in closed form (no free variables).

0.3.10 FLIP-UNIVERSALS

`FLIP-UNIVERSALS`

$$\texttt{(ALL}\ (vars_1)\ \texttt{(ALL}\ (vars_2)\ \textit{expr})) \Leftrightarrow \texttt{(ALL}\ (vars_2)\ \texttt{(ALL}\ (vars_1)\ \textit{expr}))$$

where $vars_1$ and $vars_2$ are each a sequence of one or more variables.
`(FLIP-UNIVERSALS` *index*`)`:
$\texttt{(ALL}\ (vars_1)\ \texttt{(ALL}\ (vars_2)\ \textit{expr})) \Rightarrow \texttt{(ALL}\ (vars_2)\ \texttt{(ALL}\ (vars_1)\ \textit{expr}))$.
No syntax for inverse is provided since the inference rule is its own inverse.

0.3.11 RENAME-UNIVERSAL

`RENAME-UNIVERSAL`

$$\texttt{(ALL}\ (x)\ \textit{expr}) \Leftrightarrow \texttt{(ALL}\ (y)\ \textit{expr}[x \leftarrow y])$$

where x and y are variables and $\textit{expr}[x \leftarrow y]$ is *expr* with all free occurrences of x in it replaced by y.
`(RENAME-UNIVERSAL` *index x y*`)`: $\texttt{(ALL}\ (x)\ \textit{expr}) \Rightarrow \texttt{(ALL}\ (y)\ \textit{expr}[x \leftarrow y])$.
No syntax for inverse is provided since the inference rule is its own inverse.

0.3.12 INSTANTIATE-UNIVERSAL

`INSTANTIATE-UNIVERSAL`

$$\texttt{(ALL}\ (\mathrm{x})\ P) \Leftrightarrow \texttt{(IF}\ P[x \leftarrow e]\ \texttt{(ALL}\ (\mathrm{x})\ P)\ \texttt{(FALSE))}$$

where x is a variable, e is an expression, and $P[x \leftarrow e]$ is P with all free occurrences of x in it replaced by e.
`(INSTANTIATE-UNIVERSAL` *index* `(=` *x e*`))`:
$\texttt{(ALL}\ (\mathrm{x})\ P) \Leftrightarrow \texttt{(IF}\ P[x \leftarrow e]\ \texttt{(ALL}\ (\mathrm{x})\ P)\ \texttt{(FALSE))}$.
`(INSTANTIATE-UNIVERSAL` *index* `(=` *x e*`) T)`:
$\texttt{(IF}\ P[x \leftarrow e]\ \texttt{(ALL}\ (\mathrm{x})\ P)\ \texttt{(FALSE))} \Rightarrow \texttt{(ALL}\ (\mathrm{x})\ P)$.

0.3.13 REMOVE-UNIVERSAL

`REMOVE-UNIVERSAL`

$$\texttt{(ALL}\ (vars)\ P) \Leftrightarrow \texttt{(IF}\ P\ \texttt{(TRUE)}\ \texttt{(FALSE))}$$

where *vars* is a sequence of variables that do not occur free in P.
`(REMOVE-UNIVERSAL` *index*`)`: `(ALL` (*vars*) P`)` $\Rightarrow$ `(IF` P `(TRUE) (FALSE))`.
`(REMOVE-UNIVERSAL` *index* `(`*vars*`) T)`:
`(IF` P `(TRUE) (FALSE))` $\Rightarrow$ `(ALL (`*vars*`)` P`)`.
One might call the inverse operation *universal introduction*.

0.3.14 INDUCT

`INDUCT`

$$\texttt{(ALL}\ (x)\ P\texttt{)} \Leftrightarrow$$
$$\texttt{(ALL}\ (x)\ \texttt{(IMPLIES (ALL}\ (y)\ \texttt{(IMPLIES (M<}\ y\ x\texttt{)}\ P[x \leftarrow y]\texttt{))}\ P\texttt{))}$$

where $P[x \leftarrow y]$ is P with all free occurrences of x replaced by y.
`(INDUCT` *index* y`)`:
`(ALL` (x) P`)` $\Rightarrow$
`(ALL` (x) `(IMPLIES (ALL` (y) `(IMPLIES (M<` y x`)` $P[x \leftarrow y]$`))` P`))`.

ZK never uses the `INDUCT` inference rule in reverse, although it is perfectly sound to do so.

0.3.15 COMPUTE

`COMPUTE`

$$\mathit{expr} \Leftrightarrow \mathit{compute}(\mathit{expr})$$

where *expr* must be one of the forms:

`(+` i_1 i_2`)`, `(-` i_1 i_2`)`, `(*` i_1 i_2`)`, `(negate` i_1`)`, `(div` i_1 nz`)`, `(mod` i_1 nz`)`, `(rem` i_1 nz`)`, `(ord` i_1`)`, `(type-of` i_1`)`, `(type-of` b_1`)`, `(>=` i_1 i_2`)`, `(=` i_1 i_2`)`, `(=` b_1 b_2`)`, or `(=` x x`)`,

i_1 and i_2 must be integer literals, nz must be a non-zero integer literal, b_1 and b_2 must be Boolean literals (`(TRUE)` or `(FALSE)`), and x can be any expression.
`(COMPUTE` *index*`)`: `(+` 10 5`)` $\Rightarrow$ 15.
`(COMPUTE` *index* `(+ 10 5) T)`: 15 $\Rightarrow$ `(+` 10 5`)`.

0.3.16 SYNTAX

`SYNTAX`

$$\begin{gathered}
\texttt{(AND)} \Leftrightarrow \texttt{(AND (TRUE) (TRUE))}\\
\texttt{(AND}\ x\texttt{)} \Leftrightarrow \texttt{(AND}\ x\ \texttt{(TRUE))}\\
\texttt{(AND}\ x\ y\texttt{)} \Leftrightarrow \texttt{(IF}\ x\ \texttt{(IF}\ y\ \texttt{(TRUE) (FALSE)) (FALSE))}\\
\texttt{(AND}\ x\ y\ z\ \ldots\texttt{)} \Leftrightarrow \texttt{(AND}\ x\ \texttt{(AND}\ y\ z\ \ldots\texttt{))}\\
\texttt{(OR)} \Leftrightarrow \texttt{(OR (FALSE) (FALSE))}\\
\texttt{(OR}\ x\texttt{)} \Leftrightarrow \texttt{(OR}\ x\ \texttt{(FALSE))}\\
\texttt{(OR}\ x\ y\texttt{)} \Leftrightarrow \texttt{(IF}\ x\ \texttt{(TRUE) (IF}\ y\ \texttt{(TRUE) (FALSE)))}\\
\texttt{(OR}\ x\ y\ z\ \ldots\texttt{)} \Leftrightarrow \texttt{(OR}\ x\ \texttt{(OR}\ y\ z\ \ldots\texttt{))}\\
\texttt{(IMPLIES}\ x\ y\texttt{)} \Leftrightarrow \texttt{(IF}\ x\ \texttt{(IF}\ y\ \texttt{(TRUE) (FALSE)) (TRUE))}\\
\texttt{(NOT}\ x\texttt{)} \Leftrightarrow \texttt{(IF}\ x\ \texttt{(FALSE) (TRUE))}\\
\texttt{(*)} \Leftrightarrow \texttt{(* 1 1)}\\
\texttt{(*}\ x\texttt{)} \Leftrightarrow \texttt{(*}\ x\ \texttt{1)}\\
\texttt{(*}\ x\ y\ z\ \ldots\texttt{)} \Leftrightarrow \texttt{(*}\ x\ \texttt{(*}\ y\ z\ \ldots\texttt{))}
\end{gathered}$$

$$\begin{gathered}
\texttt{(+)} \Leftrightarrow \texttt{(+ 0 0)} \\
\texttt{(+ } x\texttt{)} \Leftrightarrow \texttt{(+ } x \texttt{ 0)} \\
\texttt{(+ } x\ y\ z \ldots\texttt{)} \Leftrightarrow \texttt{(+ } x \texttt{ (+ } y\ z \ldots\texttt{))} \\
\texttt{(MAKE-SET)} \Leftrightarrow \texttt{(NULLSET)} \\
\texttt{(MAKE-SET } a\texttt{)} \Leftrightarrow \texttt{(SETADD } a \texttt{ (NULLSET))} \\
\texttt{(MAKE-SET } a\ b \ldots\texttt{)} \Leftrightarrow \texttt{(SETADD } a \texttt{ (SETADD } b \ldots\texttt{))} \\
\texttt{(SOME (}\ldots\texttt{) } P\texttt{)} \Leftrightarrow \texttt{(NOT (ALL (}\ldots\texttt{) (NOT } P\texttt{)))} \\
\texttt{(ALL (}x\ y \ldots\texttt{) } P\texttt{)} \Leftrightarrow \texttt{(ALL (}x\texttt{) (ALL (}y \ldots\texttt{) } P\texttt{))}.
\end{gathered}$$

`(SYNTAX` *index*`)`: `(AND)` $\Rightarrow$ `(AND (TRUE) (TRUE))`
`(SYNTAX` *index* `T)`: `(AND (TRUE) (TRUE))` $\Rightarrow$ `(AND)`.

0.4 Proof Checker Implementation Considerations

Although not a requirement, implementing a Proof Checker in Lisp or Scheme would be a good choice, since declarations and proof logs generated in ZK are stored as s-expressions in compressed files (in gzip format).

A proof checker would need to be able to process declarations in addition to proof logs. Although it also needs to process `LOAD` commands, the proof checker need not implement a library facility since all loaded declarations are made available by ZK. Of course, if correctness of the ZK library facility is of concern, then some kind of library mechanism needs to be implemented.

As it is now, a proof checker would need to know the *initial theory* of ZK. In the future, the initial theory may be made available by ZK, perhaps as a compressed flle.

The extent to which a proof checker ought to check the result of a theory development in ZK depends on the degree of trust we place on various components of ZK and the level of assurance we want to obtain. There are always tradeoffs involved: extra checking means more code which makes it harder to ensure the proof checker is correct. At the very least, a proof checker ought to check the result of applying entries in a proof log to a formula.

Bibliography

[1] ANSI. Programming Language Common Lisp. Technical Report INCITS 226-1994, ANSI, 1994.

[2] Dan Craigen. Reference Manual for the Language Verdi. Technical Report TR-96-5482-07, ORA Canada, February 1996.

[3] Sentot Kromodimoeljo, Bill Pase, Mark Saaltink, Dan Craigen, and Irwin Meisels. The EVES System. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 349–373. Springer, 1993.

[4] Mark Saaltink. A Formal Description of Verdi. Technical Report TR-90-5429-10b, ORA Canada, 1994.

[5] Mark Saaltink. The Z/EVES System. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

[6] J. Michael Spivey. *Z Notation - A Reference Manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.